Meet recursive session types*

Roberto Bruni[1] and Leonardo Gaetano Mezzina[2]

[1] Computer Science Department, University of Pisa, Italy
[2] IMT Lucca, Institute for Advanced Studies, Italy

Abstract. Session types have been introduced to govern long running interaction protocols between two parties. We focus on regular session types, which can account for the sequence of typed I/O actions and internal/external choices to be carried on by (possibly recursive) protocols. They allow studying the degree of compatibility between two communicating entities and they have been applied to many process calculi, ranging from π-calculus inspired ones, to object-oriented and, more recently, to service-oriented calculi. This paper addresses the problem of computing the intersection (meet) of recursive session types, which is useful to define a deterministic procedure for the typing of conditional constructs, where different branches can follow different protocols, or the typing of different calls to the same service, when session types are seen as service contracts. We first introduce meet algebraically, as the greatest lower bound according to Gay and Hole's subtyping relation, then give a co-inductive characterization for the meet in terms of non-deterministic inference rules and finally we distill an algorithm to infer the meet.

1 Introduction

In this paper we face the problem of how to compute the intersection of, possibly infinite, regular session types. Our interest in computing the meet emerged while developing type inference algorithms for service-oriented and session-based process calculi. The semantic gap (in terms of representation distance) between such calculi and lambda-calculus motivates the direct study of session types. In fact, session types are the most accepted paradigm to type sessions. A dyadic session is a private communication between two parties which have associated a dual session types [12]. Since the communication is private the type system has also to worry about the linear usage of a session; that is, the partner of each session side must always be uniquely determined. In the course of this paper we assume the linearity check is achieved separately, when needed, without giving the details. Consequently, each sequent of a type judgment is separated in three parts: a standard typing environment, the process and the linear typing environment. The standard typing environment contains assumptions about types of non linear names; for example the standard π-calculus channels [16] (see [11]) or names used to instantiate new fresh sessions (often called *ports*

* Research supported by the EU within the FET-GC II Integrated Project IST-2005-016004 SENSORIA and by the Italian FIRB Project TOCAI.IT.

[12, 19] or *services* [4]). The linear typing environment tracks the sequence of actions in each session. Each action can be an input/output action or an internal/external choice. Accordingly, each action has a respective co-action and it is easy to compute the syntactic dual type $\overline{T}$ of a session type T simply turning the various actions in the respective co-actions.

The external choice is a `switch`-like construct (in the sense of traditional programming languages) which is used to offer a finite set of options for the other party to select from. The internal choice concerns the potential selections a process may perform. For example, different branches of a conditional statement can carry different choices.

This philosophy is followed in [12] where the types of two dual sessions match, up to exchanging actions with co-actions. However, this choice introduces a certain degree of non determinism, especially in the rule for typing the external choice. Since types must match syntactically their duals it can happen that additional (fictitious) internal choices must be introduced nondeterministically during the typing. As an example consider the process described in the syntax of [19]:

```
accept db(k) in k▷{ query:k?(sqlvar)in P1
                  || update:P2
                  || add:P3 }
request db(k) in if test then k◁query ; k!(sqlstr) ; P
                         else k◁update ; Q
```

A database service is offered by means of an `accept` on the shared name `db`. After the client `request` is issued a new private session k is created that is shared between the server and the client. The new session ensures the client and the service have a private conversation when communicating on k. The example service offers three options to choose from: `query`, `update` and `add`. The client depending on the outcome of the expression $test$ (representing its internal state) chooses either `query` or `update`. If the client requests the `query` option, then the service expects a SQL query which will be bound in the string variable $sqlvar$ in the body process P. Suppose each P_i has type T_i, P has type $\overline{T_1}$ and Q has type $\overline{T_2}$ (for $\overline{T_i}$ denoting the syntactic dual of T_i), then the process is well typed when k has type, say, $\&\{\texttt{query}{:}?(string).T_1,\ \texttt{update}{:}T_2,\ \texttt{add}{:}T_3\}$ on service side but k has type $\oplus\{\texttt{query}{:}!(string).\overline{T_1},\ \texttt{update}{:}\overline{T_2},\ \texttt{add}{:}\overline{T_3}\}$ on client side, because these two types are syntactically equivalent if we exchange $\&$ (the symbol for external choice) with $\oplus$ (the symbol for internal choice) and ? (the symbol for input) with ! (the symbol for output). Notice however that the typing rules on the client side must guess locally the type of the external choice provided by the service; i.e., the typing rule of a process of the form $k \triangleleft l; P$ must guess the entire set of choices offered by the counterpart. In turn, this will also force the two branches of the conditional statement to take into account the same set of alternatives.

This is a common method, especially for simplifying the various proofs that involve session type systems, as witnessed by the numbers of useful properties proved for the initial proposal and its variants [7,4,19,3]. However, a non tricky procedure that wishes to deduce the type of an internal choice has to insert a set of choices besides the selected one and then ask the rest of the system if the type matches its dual. If not, then it must retry until the duality is achieved, i.e. the type of a process depends on the context where it is used. The obvious step is to relax the notion of duality permitting branches in external choices to be a subset of branches in an external choice. This solves one problem, but what if different choices are nested inside an if-then-else construct, like in the example above? The solution we propose is that the typing rule for `if` e `then` P `else` Q computes the intersection between the types of P and Q, instead of requiring P and Q to have exactly the same type. This is an effective method which allows a simple solution and supports the inference of session types.

The operation of intersection is built on a partial order, namely the intersection is the greatest lower bound of two elements. In the following we shall use the terms *intersection* and *meet* interchangeably. The pre-order we exploit is due to Gay and Hole [11] which proposed the notion of subtyping $\leq$ for session types. From the algebraic definition of the greatest lower bound, we extract the meet relation (defined on triples) written as $\_\overset{c}{\wedge}\_ = \_$. The fact that the meet of two regular infinite session types, when it exists, can be represented by a regular infinite session type is maybe not obvious and contributed here.

Finally, we guess that usages of meet are not restricted only on static type-checking but can be useful, for example, in a context where session types are considered as contracts in the sense of [15,5] or a multi party scenario, in the sense of [13,2], to determine the smallest common behavior of a set of contracts.

Outline of the paper. Section 2 introduces the basic notation for session types. Section 3 presents the algebraic notion of intersection in terms of the greatest lower bound of a partial order. Section 4 presents two co-inductive relations: one on triples that captures two session types and their respective meet and one that captures all pairs of session types on which the meet is defined. Section 5 shows an algorithm to construct the meet of two session types and proves its correctness w.r.t. the two previous defined relations. Section 6 summarizes the results and points out directions for further work. To convince the reviewers about the the applicability and effectiveness of our result, in Appendix B we show how to apply the meet to two well-known typed calculi present in the literature and enriched here with a conditional construct, as well as to a more recent type system proposed by the authors themselves so to derive a (deterministic and terminating) type inference algorithm.

2 Session Types

We assume an infinite collection $\alpha, \beta, \ldots$ of type variables and an infinite collection $l, \ldots$ of labels. We distinguish session types σ and types for message contents ϕ and we let T, U, V, S range over the least fixed point of the function induced by the grammar in Fig. 1. The types $?(S_1, \ldots, S_n).T$ and $!(S_1, \ldots, S_n).T$

$$\begin{array}{rll}
\sigma ::= & \mathtt{end} & \text{(no action)} \\
| & ?(\phi_1, \ldots, \phi_n).\sigma & \text{(input of a tuple)} \\
| & !(\phi_1, \ldots, \phi_n).\sigma & \text{(output of a tuple)} \\
| & \&\{l_1 : \sigma_1, \ldots, l_n : \sigma_n\} & \text{(external choice)} \\
| & \oplus\{l_1 : \sigma_1, \ldots, l_n : \sigma_n\} & \text{(internal choice)} \\
| & \mu\alpha.\sigma & \text{(recursive behavior)} \\
| & \alpha & \text{(type variable)}
\end{array}
\qquad
\begin{array}{rll}
\phi ::= & \tau & \text{(basic type)} \\
| & \sigma & \text{(session delegation)} \\
| & [\sigma] & \text{(session reference)}
\end{array}$$

Fig. 1. Syntax of types

represent respectively an input and an output of tuple of type $S_1, \ldots, S_n$ respectively, followed by the continuation T. The types $\&\{l_1 : T_1, \ldots, l_n : T_n\}$ and $\oplus\{l_1 : T_1, \ldots, l_n : T_n\}$ denote respectively an external and an internal choice. The external choice is used to offer a set of options to the partner while the internal choice expresses the selections a partner may wish to perform. The recursive expression $\mu\alpha.T$ represents the recursive behaviors and μ is a binder that gives rise, in the standard way, to notions of bound (bv) and free variables (fv), closed μ-types, and equivalence of μ-types up to renaming of bound variables. In the proofs we also use the function $\mathsf{unfold}(T)$ defined by recursion on the structure of T: $\mathsf{unfold}(\mu\alpha.T) = \mathsf{unfold}(T[^{\mu\alpha.T}/_{\alpha}])$ and in all other cases $\mathsf{unfold}(T) = T$. Finally, τ represents a basic data value (we shall rely on int and $string$) and $[T]$ is the type of a non linear name, i.e.; $[T]$ allows to send and receive ports, services or channels which can be used according to T.

We often write $\tilde{T}$ for a sequence $T_1, \ldots, T_n$ of types and $\diamondsuit\{l_i : T_i\}_{i\in I}$ for $\diamondsuit\{l_{i_1} : T_{i_1}, \ldots, l_{i_n} : T_{i_n}\}$ where $I = \{i_1, \ldots, i_n\}$ and $\diamondsuit \in \{\oplus, \&\}$. We shall use sometimes also the (partial) function **;** defined for both kinds of choice as:

$$\diamondsuit\{l : T\} \,;\, \diamondsuit\{l_1 : T_1, \ldots, l_n : T_n\} = \begin{cases} \diamondsuit\{l_1 : T_1, \ldots, l : T, \ldots, l_n : T_n\} & \text{if } l \notin \{l_1, \ldots, l_n\} \\ \diamondsuit\{l_1 : T_1, \ldots, l_n : T_n\} & \text{if } l = l_i \text{and } T = T_i \text{ for some } i \\ undefined & \text{otherwise} \end{cases}$$

inductively extended as expected when the first operator is a general choice.

Definition 1 (Well formedness). *A type T is well formed if it is contractive (i.e., in any subexpression $\mu\alpha.\mu\alpha_1 \ldots \mu\alpha_n.T'$ the body T' is not α, see [8]) and if the same label does not appear twice in a choice (i.e., in any $\diamondsuit\{l_i : T_i\}_{i\in I}$ if $i \neq j$ then $l_i \neq l_j$).*

For simplicity we call types only well formed types and denote with $TYPE$ the set of all types. Each session type T has a syntactic dual type $\overline{T}$, recursively defined by the equations in Fig. 2.

We now present a few session types that will serve as a running example:

$$T = \mu\alpha.\&\{\mathbf{quit} : \mathtt{end}, \; \mathbf{play} : \mu\beta.\&\{\mathbf{prev} : \alpha, \; \mathbf{vol} : ?(int).\beta\}\} \qquad S = \mu\alpha.\&\{\mathbf{quit} : \mathtt{end}, \; \mathbf{play} : \mu\beta.\&\{\mathbf{prev} : \alpha, \; \mathbf{bright} : ?(int).\beta\}\}$$

$$\overline{[T]} = [\overline{T}] \qquad \overline{\tau} = \tau \qquad \overline{\texttt{end}} = \texttt{end} \qquad \overline{?(\tilde{S}).T} = !(\tilde{S}).\overline{T} \qquad \overline{!(\tilde{S}).T'} = ?(\tilde{S}).\overline{T'}$$
$$\overline{\&\{l_1 : T_1, \ldots, l_n : T_n\}} = \oplus\{l_1 : \overline{T_1}, \ldots, l_n : \overline{T_n}\}$$
$$\overline{\oplus\{l_1 : T_1, \ldots, l_n : T_n\}} = \&\{l_1 : \overline{T_1}, \ldots, l_n : \overline{T_n}\} \qquad \overline{\mu\alpha.T} = \mu\alpha.\overline{T} \qquad \overline{\alpha} = \alpha$$

Fig. 2. The syntactic dual of a session types

$$U = \mu\alpha.\&\{\mathbf{quit} : \texttt{end}, \mathbf{play} : \&\{\mathbf{prev} : \alpha\}\} \qquad U' = \mu\alpha.\&\{\mathbf{play} : \&\{\mathbf{prev} : \alpha\}\}$$

Type T represents the type of a menu that offers two choices: **quit** which allows to exit from the menu and **play** which allows accessing a submenu. The submenu is composed of two choices: **prev** permits returning to the main menu and **vol** that after the communication of an integer returns to the submenu. Type S is similar but for the branch **bright** in place of **vol**. Now it is clear that U has the common behaviors of both menus T and S and it is a possible candidate for their meet. In fact, U can offer either the option **quit** or the two consecutive options **play** and **prev**. U' it is also a possible candidate for the meet since it has the common behaviors of T and S, but it also lacks of **quit**. Intuitively we want U to be the meet of T and S, not U'.

3 Algebraic meet of session types

In this section we discuss the algebraic notion of intersection with respect to the pre-order $\leq$ given in [11].

Definition 2. *A relation $R \subseteq TYPE \times TYPE$ is a type simulation relation if $(T, V) \in R$ implies the following conditions:*

1. *if* $\mathsf{unfold}(T) = [T']$ *then* $\mathsf{unfold}(V) = [V']$ *and* $(T', V') \in R$ *and* $(V', T') \in R$
2. *if* $\mathsf{unfold}(T) = !(S'_1, \ldots, S'_n).T'$ *then* $\mathsf{unfold}(V) = !(S''_1, \ldots, S''_n).V'$ *and for all* $i \in \{1, \ldots, n\}$, $(S''_i, S'_i) \in R$ *and* $(T', V') \in R$
3. *if* $\mathsf{unfold}(T) = ?(S'_1, \ldots, S'_n).T'$ *then* $\mathsf{unfold}(V) = ?(S''_1, \ldots, S''_n).V'$ *and for all* $i \in \{1, \ldots, n\}$, $(S'_i, S''_i) \in R$ *and* $(T', V') \in R$
4. *if* $\mathsf{unfold}(T) = \&\{l_1 : T_1, \ldots, l_m : T_m\}$ *then* $\mathsf{unfold}(V) = \&\{l'_1 : V_1, \ldots, l'_n : V_n\}$ *and* $m \leq n$ *and for all* $i \in \{1, \ldots, m\}$ *exists one* $j \in \{1, \ldots, n\}$ *such that* $l_i = l'_j$ *and* $(T_i, V_j) \in R$
5. *if* $\mathsf{unfold}(T) = \oplus\{l_1 : T_1, \ldots, l_m : T_m\}$ *then* $\mathsf{unfold}(V) = \oplus\{l'_1 : V_1, \ldots, l'_n : V_n\}$ *and* $n \leq m$ *and for all* $i \in \{1, \ldots, n\}$ *exists one* $j \in \{1, \ldots, m\}$ *such that* $l'_i = l_j$ *and* $(T_i, V_j) \in R$
6. *if* $\mathsf{unfold}(T) = \texttt{end}$ *then* $\mathsf{unfold}(V) = \texttt{end}$
7. *if* $\mathsf{unfold}(T) = \tau$ *then* $\mathsf{unfold}(V) = \tau$

The co-inductive $\leq$ relation is defined as $T \leq V$ if there exists a type simulation relation R such that $(T, V) \in R$.

The relation $\leq$ is essentially the subtyping relation introduced in [11] with the only difference that branches are considered not ordered.

Proposition 1. *$\leq$ is a pre-order.*

We consider the equivalence relation $\stackrel{\mu}{=}$ induced by a $\leq$,

Definition 3. *We write $T \stackrel{\mu}{=} V$ iff $T \leq V$ and $V \leq T$ hold.*

Consequently, $\leq$ can be considered a partial order up to $\stackrel{\mu}{=}$. We write $\mathsf{greatest}(T, U, V)$ for the predicate $\forall\ V'.V' \leq T \wedge V' \leq U \Rightarrow\ V' \leq V$. Since the relation $\leq$ is a partial order up to $\stackrel{\mu}{=}$, defining the meet of two types requires a bit of attention. The problem is due to the fact that given two types there exists a (possibly infinite) set of greatest lower bounds. In particular all the elements of such a set belong to the equivalence class generated by the relation $\stackrel{\mu}{=}$. In a few words, if V is the greatest lower bound of T and U then all the elements of $[V]_{\stackrel{\mu}{=}}$ (the equivalence class of V w.r.t. $\stackrel{\mu}{=}$) also are. Thereby the uniqueness of the meet is intended with respect to the notion of $\stackrel{\mu}{=}$.

Definition 4 (Type Intersection). *The intersection of two types $T \wedge U$ is the (unique w.r.t. $\stackrel{\mu}{=}$) type V such that $V \leq T$ and $V \leq U$ and $\mathsf{greatest}(T, U, V)$.*

It is also easy to see that the following holds:

Lemma 1. *For any T, U, V:*

- *Idempotency:* $(T \wedge T) \stackrel{\mu}{=} T$
- *Commutativity:* $(T \wedge U) \stackrel{\mu}{=} (U \wedge T)$
- *Associativity:* $T \wedge (U \wedge V) \stackrel{\mu}{=} (T \wedge U) \wedge V$

Example 1. Consider types given in Section 1. We have both $U \leq T$, $U \leq S$ and $U' \leq T$, $U' \leq S$ but since $U' \leq U$ then U is more suitable than U' to be the meet of T and S, but for now we do not have an automatic manner to verify that $\mathsf{greatest}(T, S, U)$ holds since we have an universal quantification over types.

We shall need an additional proposition which relates the $\leq$ relation with the duality relation:

Proposition 2. *If $T \leq V$ then $\overline{V} \leq \overline{T}$.*

Proof. It suffices to prove that $R = \{(\overline{V}, \overline{T}) | T \leq V\}$ is a type simulation relation. We sketch the input and external choice cases. If $\mathsf{unfold}(\overline{V}) = ?(S_1, \ldots, S_n).\overline{V'}$ then by definition of $\leq$, $\mathsf{unfold}(\overline{T}) = ?(S'_1, \ldots, S'_n).\overline{T'}$ and we conclude since $(S_i, S'_i) \in R$ because $S'_i \leq S_i$ (by the contravariance of output [17] and the fact that the duality relation is not computed on the input tuples) and $(\overline{T'}, \overline{V'}) \in R$ because $V' \leq T'$.

If $\mathsf{unfold}(\overline{V}) = \&\{l_1 : \overline{V_1}, \ldots, l_m : \overline{V_m}\}$ then $\mathsf{unfold}(\overline{T}) = \&\{l'_1 : \overline{T_1}, \ldots, l'_n : \overline{T_n}\}$ and $m \leq n$ and for all $i \in \{1, \ldots, m\}$ exists one $j \in \{1, \ldots, n\}$ such that $l_i = l'_j$ and $(\overline{T_j}, \overline{V_i}) \in R$ since $(V_i \leq T_j)$

Corollary 1. *$T \leq V$ iff $\overline{V} \leq \overline{T}$*

Proof. By the previous proposition and by the fact that $\overline{\overline{T}} = T$, i.e. the syntactic duality relation is involutive.

4 Inference Relations for the Meet

In this section we overcome the problem introduced by the end of Section 2 relatively to the universal quantification in the definition of the predicate greatest. To this aim we introduce two new relations: one that checks whether a type is effectively the meet of two given types and another one that checks the existence of the meet for two given types.

Definition 5. *A relation $R \subseteq$ TYPE $\times$ TYPE $\times$ TYPE is a meet relation if $(T, U, V) \in R$ implies the following conditions:*

1. *if* $\mathsf{unfold}(T) = [T']$ *then* $\mathsf{unfold}(U) = [U']$ *and* $\mathsf{unfold}(V) = [V']$ *and* $T' \stackrel{\mu}{=} U'$ *and* $U' \stackrel{\mu}{=} V'$
2. *if* $\mathsf{unfold}(T) = !(S'_1, \ldots, S'_n).T'$ *then* $\mathsf{unfold}(U) = !(S''_1, \ldots, S''_n).U'$ *and* $\mathsf{unfold}(V) = !(S'''_1, \ldots, S'''_n).V'$ *and for all* $i \in \{1, \ldots, n\}$, $(\overline{S'_i}, \overline{S''_i}, \overline{S'''_i}) \in R$ *and* $(T', U', V') \in R$
3. *if* $\mathsf{unfold}(T) = ?(S'_1, \ldots, S'_n).T'$ *then* $\mathsf{unfold}(U) = ?(S''_1, \ldots, S''_n).U'$ *and* $\mathsf{unfold}(V) = ?(S'''_1, \ldots, S'''_n).V'$ *and for all* $i \in \{1, \ldots, n\}$, $(S'_i, S''_i, S'''_i) \in R$ *and* $(T', U', V') \in R$
4. *if* $\mathsf{unfold}(T) = \&\{l_i : T_i\}_{i \in I}$ *then* $\mathsf{unfold}(U) = \&\{l'_j : T'_j\}_{j \in J}$ *and letting* $K = \{(i,j) \mid l_i = l'_j\}$ *with* $|K| \geq 1$ *we have* $\mathsf{unfold}(V) = \&\{l_i : U_{(i,j)}\}_{(i,j) \in K}$ *and* $\forall i \forall j (i,j) \in K \Rightarrow (T_i, T'_j, U_{(i,j)}) \in R$
5. *if* $\mathsf{unfold}(T) = \oplus\{l_i : T_i\}_{i \in I}$ *then* $\mathsf{unfold}(U) = \oplus\{l'_j : T'_j\}_{j \in J}$ *and letting* $K = \{(i,j) \mid l_i = l'_j\}$, $I' = \{(i \in I) \mid \forall i (i,j) \notin K\}$,$J' = \{(j \in J) \mid \forall j (i,j) \notin K\}$ *we have* $\mathsf{unfold}(V) = \oplus\{l_i : U_{(i,j)}\}_{(i,j) \in K}$; $\oplus\{l_i : T_i\}_{i \in I'}$; $\oplus\{l'_j : T'_j\}_{j \in J'}$ *and* $\forall i \forall j (i,j) \in K \Rightarrow (T_i, T'_j, U_{(i,j)}) \in R$
6. *if* $\mathsf{unfold}(T) = \mathtt{end}$ *then* $\mathsf{unfold}(U) = \mathtt{end}$ *and* $\mathsf{unfold}(V) = \mathtt{end}$
7. *if* $\mathsf{unfold}(T) = \tau$ *then* $\mathsf{unfold}(U) = \tau$ *and* $\mathsf{unfold}(V) = \tau$

The co-inductive $\_ \stackrel{c}{\wedge} \_ = \_$ *relation is defined as* $T \stackrel{c}{\wedge} U = V$ *if there exists a meet relation* R *such that* $(T, U, V) \in R$.

Since the output is contravariant [17] the respective rule requires the syntactic dual of each sent type to be co-inductively in the relation. The duality relation is exploited here to use the same algorithm but to compute the union (or the join). The rule in fact would require to compute the least upper type of $\tilde{S}'$ and $\tilde{S}''$. Rule for choices use the intersection set K which contains the pairs of indexes with equal labels in each choice. Remember that we are considering only well formed types, hence labels in choices are distinct and consequently each component in a pair belonging to K is distinct too. Accordingly, the type $\diamondsuit\{l_i : U_{(i,j)}\}_{(i,j) \in k}$ has all the labels in the intersection where l_i are some of the labels indexed by I (here we used l_i the labels of the first type, but l'_j would do too). Moreover, the rule for external choice uses the **;** operation to exhibit the type U. The resulting type contains branches in the intersection (indexed by K) which are constrained to be co-inductively in the relation together with the remaining branches of the type T (indexed by I') and the remaining branches of the type V (indexed by J'). There is a subtlety in the rule for external choice, as the relation is defined only if K contains at least one element. A similar check is not necessary for the internal choice since the requirement is automatically fulfilled since T and V contain at least one branch each.

Next theorem shows that indeed $\_ \overset{c}{\wedge} \_ = \_$ relates two types with their respective meet.

Theorem 1. *$T \wedge U \overset{\mu}{=} V$ iff $T \overset{c}{\wedge} U = V$*

Proof. $\Rightarrow$) We show that $R = \{(T, U, V) | V \leq T$ and $V \leq U$ and $\mathsf{greatest}(T, U, V)\}$ is a meet relation. Notice that the first case where we check the membership of $([T'], [V'], [U'])$ comes directly from the definition of $\overset{\mu}{=}$. We sketch instead the case where $\mathsf{unfold}(V)$ is an input. If $\mathsf{unfold}(V) = !(S'''_1, \ldots, S'''_n).V'$ by definition of $\leq$ we must have $\mathsf{unfold}(T) = !(S'_1, \ldots, S'_n).T'$ with $S'_i \leq S'''_i$ for all $i \in \{1, \ldots, n\}$ and $V' \leq T'$. Similarly for $U = !(S''_1, \ldots, S''_n).U'$ we have $S''_i \leq S'''_i$ and $V' \leq U'$. By Proposition 2 we have $\overline{S'''_i} \leq \overline{S'_i}$ and $\overline{S''_i} \leq \overline{S'_i}$. To conclude the proof, note that $\mathsf{greatest}(\overline{S'_i}, \overline{S''_i}, \overline{S'''_i})$ and $\mathsf{greatest}(T', U', V')$ trivially hold by $\mathsf{greatest}(T, U, V)$. In fact, by definition of $\mathsf{greatest}(T, U, V)$ we have $\forall V_1 = !(W_1, \ldots, W_n).V'_1$ such that $V_1 \leq !(S'_1, \ldots, S'_n).T'$ and $V_1 \leq !(S''_1, \ldots, S''_n).U' \Rightarrow V_1 \leq !(S'''_1, \ldots, S'''_n).V'$. By definition of $\leq$ we have $V'_1 \leq T'$ and $V'_1 \leq U' \Rightarrow V'_1 \leq V'$ or $\mathsf{greatest}(T', U', V')$. In the same manner we have $\overline{W_i} \leq \overline{S'_i}$ and $\overline{W_i} \leq \overline{S''_i} \Rightarrow \overline{W_i} \leq \overline{S'''_i}$.
$\Leftarrow$) It suffices to show that:

1. $R_T = \{(V, T) | \exists U \; T \overset{c}{\wedge} U = V\}$ is a type simulation relation
2. $R_U = \{(V, U) | \exists T \; T \overset{c}{\wedge} U = V\}$ is a type simulation relation
3. $T \overset{c}{\wedge} U = V$ implies $\mathsf{greatest}(T, U, V)$

We sketch case 1 where $\mathsf{unfold}(T) = \&\{l_i : T_i\}_{i \in I}$. By definition of $\_ \overset{c}{\wedge} \_ = \_$ if $\mathsf{unfold}(U) = \&\{l'_j : T'_j\}_{j \in J}$ then $\mathsf{unfold}(V) = \&\{l_i : V_{(i,j)}\}_{(i,j) \in K}$. The result follows since $|K| \leq |I|$ and the labels of V are a subset of the labels in T and by definition, $\_ \overset{c}{\wedge} \_ = \_$ co-inductively holds on branches indexed by K.
Case 3 is an immediate consequence of cases 1 and 2 together with the fact that each time $\_ \overset{c}{\wedge} \_ = \_$ builds the greatest subtype. As an example consider again the case where $\mathsf{unfold}(T) = \&\{l_i : T_i\}_{i \in I}$ then $\mathsf{unfold}(U) = \&\{l'_j : T'_j\}_{j \in J}$ and $\mathsf{unfold}(V) = \&\{l_i : V_{(i,j)}\}_{(i,j) \in K}$. Now each type V' which is subtype of both T and U is of the form $\mathsf{unfold}(V') = \&\{l_i : V'_{(i,j)}\}_{(i,j) \in K'}$ where $K' \subseteq K$.

Example 2. The type U' in Section 2 is not the meet of the two types T and S, since does not exist a meet relation which contains the tuple (T, S, U). In particular, the test fails due to the lack in U' of the label **quit** in a choice together with **play**.

Now we introduce another relation which contains all the pairs of types for which the meet is defined.

Definition 6. *A relation $R \subseteq TYPE \times TYPE$ is a meet exists relation if $(T, U) \in R$ implies the following conditions:*

1. *if* $\mathsf{unfold}(T) = [T']$ *then* $\mathsf{unfold}(U) = [U']$ *and* $T' \overset{\mu}{=} U'$
2. *if* $\mathsf{unfold}(T) = !(S'_1, \ldots, S'_n).T'$ *then* $\mathsf{unfold}(U) = !(S''_1, \ldots, S''_n).U'$ *and for all* $i \in \{1, \ldots, n\}$, $(\overline{S'_i}, \overline{S''_i}) \in R$ *and* $(T', U') \in R$
3. *if* $\mathsf{unfold}(T) = ?(S'_1, \ldots, S'_n).T'$ *then* $\mathsf{unfold}(U) = ?(S''_1, \ldots, S''_n).U'$ *and for all* $i \in \{1, \ldots, n\}$, $(S'_i, S''_i) \in R$ *and* $(T', U') \in R$
4. *if* $\mathsf{unfold}(T) = \&\{l_i : T_i\}_{i \in I}$ *then* $\mathsf{unfold}(U) = \&\{l'_j : T'_j\}_{j \in J}$ *and* $K = \{(i, j) \mid l_i = l'_j\}$ *and* $|K| \geq 1$ *and* $\forall i \forall j (i, j) \in K \Rightarrow (T_i, T'_j) \in R$
5. *if* $\mathsf{unfold}(T) = \oplus\{l_i : T_i\}_{i \in I}$ *then* $\mathsf{unfold}(U) = \oplus\{l'_j : T'_j\}_{j \in J}$ *and* $K = \{(i, j) \mid l_i = l'_j\}$, $\forall i \forall j (i, j) \in K \Rightarrow (T_i, T'_j) \in R$
6. *if* $\mathsf{unfold}(T) = \mathtt{end}$ *then* $\mathsf{unfold}(U) = \mathtt{end}$
7. *if* $\mathsf{unfold}(T) = \tau$ *then* $\mathsf{unfold}(U) = \tau$

The co-inductive $\overset{c}{\wedge}$ *relation is defined as* $T \overset{c}{\wedge} U$ *if there exists a* meet exists *relation* R *such that* $(T, U) \in R$.

Simply, previous definition ignores the shape of the meet, checking only the consistency of T and U. Next theorem formally prove the fact that the relation $\overset{c}{\wedge}$ is defined only if the meet of two types exists.

Theorem 2. $T \overset{c}{\wedge} U$ *iff* $\exists V$ *such that* $T \overset{c}{\wedge} U = V$ *.*

Proof. $\Leftarrow$) It suffices to show that $R = \{(T, U) | \exists V \; T \overset{c}{\wedge} U = V\}$ is a meet exists relation. The proof follows directly from the two definitions.

$\Rightarrow$) It suffices to show that $R = \{(T, U, V) | T \overset{c}{\wedge} U \text{ and } V \leq T \text{ and } V \leq U \text{ and } \mathsf{greatest}(T, U, V)\}$ is a meet relation. All cases are simple. If $\mathsf{unfold}(T) = \&\{l_i : T_i\}_{i \in I}$ then $\mathsf{unfold}(U) = \&\{l'_j : T'_j\}_{j \in J}$ and the result follows if we choose $V = \&\{l_i : U_{(i,j)}\}_{(i,j) \in K}$ with $K = \{(i, j) \mid l_i = l'_j\}$ and $|K| \geq 1$ and $\forall i \forall j (i, j) \in K$, $U_{(i,j)} \leq T_i$ and $U_{(i,j)} \leq T'_j$ and $\mathsf{greatest}(T_i, T'_j, U_{(i,j)})$

Example 3. With respect to the four types introduced in section 1 we have $\forall x, y \in \{T, S, U, U'\} \; x \overset{c}{\wedge} y$.

5 Algorithmic Meet

In this section we discuss our algorithm for computing the meet. First we present the algorithm for the membership checking of both relations $\overset{c}{\wedge}$ in Fig. 3

and $\_\overset{c}{\wedge}\_ = \_$ in Fig. 4. The algorithmic sequents for $\overset{c}{\wedge}$ are $\Sigma \vdash T\overset{c}{\wedge}U$ where Σ represents the set of assumptions of the form $T'\overset{c}{\wedge}U'$ (Later we will introduce the set of inverse assumptions Ξ with elements $T\overset{c}{\wedge}U$ too, see Fig. 5). The algorithmic sequents for $\_\overset{c}{\wedge}\_ = \_$ are $\Sigma \vdash T\overset{c}{\wedge}U = V$ where Σ represents the set of assumptions of the form $T'\overset{c}{\wedge}U' = V'$. For the sake of brevity rules for axioms (A2-Assump) and (A3-Assump) are given when T and U and V are any session types with the understanding of an implicit check that at least one type is of the form $\mu\alpha.T'$. This is due to the fact that we consider only well formed set of assumptions; and a set of assumptions is well formed if each contained element has at least one component of the form $\mu\alpha.T'$.

The proof of soundness and completeness of the two algorithms with respect to the co-inductive definition and the proof of termination are standard [1, 8]. (Here the latter algorithm is only trivially extended with triple.)

Moreover rules (A2-Ses) and (A3-Ses) use the $\overset{\mu}{=}$-relation which can be verified with the help of $\leq$ algorithm.

The problem in computing the meet of two infinite types is that the resulting structure can be different from the two types given in input. The relation $\_\overset{c}{\wedge}\_ = \_$ gives a hint on how to build the infinite type but we need a way to represent this type finitely by means of a regular infinite session type. As discussed in the introduction, it is not obvious that such representation exists, namely is the meet of two regular session types regular? In fact at this point we only know that if the meet has a finite representation (since the set of session types are the least fixed point of the generating function derived by the relative grammar) then we have a way of checking its goodness.

The idea of computing such representation comes from the algorithmic membership checking of the greatest fixed point for recursive regular types. Simply, the algorithm acts as follows: consider for example any generic type $\mu\alpha.\mathbf{A}.\mathbf{B}.\mathbf{C}.\alpha$ to be checked against another generic type $\mu\beta.\mathbf{D}.\mathbf{E}.\beta$, where $\mathbf{A}, ..., \mathbf{E}$ are generic actions. Then when unfolding recursion to perform the check, it is important to maintain a set of assumptions, so to prevent checking twice the same relation. After the first unfolding the algorithm matches $\mathbf{A}$ with $\mathbf{D}$, then $\mathbf{B}$ with $\mathbf{E}$, then $\mathbf{C}$ with $\mathbf{D}$, then $\mathbf{A}$ with $\mathbf{E}$, then $\mathbf{B}$ with $\mathbf{D}$ and finally $\mathbf{C}$ with $\mathbf{E}$, for a total of six matches. After that, the algorithm ends since it re-encouters the original pair of types, already present in the assumptions. All the checks made by the algorithm cover the entire range of possibilities for the first type and for the second type, this is also the desired behavior of the meet, namely the meet of the two previous types is composed by six components.

The algorithm $\Sigma \vdash T\overset{c}{\wedge}U \rightarrow V \rhd \Xi$ exploits this characteristic of the membership checking algorithms to compute the full range of possible behaviors of the meet. The algorithm is reported in Fig. 5. The missing rules (ALG-Bas), (ALG-In), (ALG-Ses) and (ALG-Choice) are similar to the algorithm for $\Sigma \vdash T\overset{c}{\wedge}U = V$: in addition, they just propagate the set of *inverse assumptions* Ξ from the premises to the conclusions. (Notice that the values contained in Ξ

$$\text{(A2-End)}\quad \Sigma \vdash \texttt{end}\overset{c}{\wedge}\texttt{end} \qquad \text{(A2-Bas)}\quad \Sigma \vdash \tau\overset{c}{\wedge}\tau \qquad \text{(A2-Assump)}\quad \Sigma, T\overset{c}{\wedge}U \vdash T\overset{c}{\wedge}U$$

$$\text{(A2-Out)}\quad \frac{\forall i, \Sigma \vdash \overline{S_i}\overset{c}{\wedge}\overline{S'_i} \quad \Sigma \vdash T\overset{c}{\wedge}U}{\Sigma \vdash !(S_1,\dots,S_n).T\overset{c}{\wedge}!(S'_1,\dots,S'_n).U} \qquad \text{(A2-In)}\quad \frac{\forall i, \Sigma \vdash S_i\overset{c}{\wedge}S'_i \quad \Sigma \vdash T\overset{c}{\wedge}U}{\Sigma \vdash ?(S_1,\dots,S_n).T\overset{c}{\wedge}?(S'_1,\dots,S'_n).U}$$

$$\text{(A2-Select)}\quad \frac{K = \{(i,j) \mid l_i = l'_j\} \quad |K| \geq 1 \quad \forall (i,j) \in K.\Sigma \vdash T_i\overset{c}{\wedge}T'_j}{\Sigma \vdash \&\{l_i : T_i\}_{i\in I}\overset{c}{\wedge}\&\{l'_j : T'_j\}_{j\in J}} \qquad \text{(A2-Choice)}\quad \frac{K = \{(i,j) \mid l_i = l'_j\} \quad \forall (i,j) \in K.\Sigma \vdash T_i\overset{c}{\wedge}T'_j}{\Sigma \vdash \oplus\{l_i : T_i\}_{i\in I}\overset{c}{\wedge}\oplus\{l'_j : T'_j\}_{j\in J}}$$

$$\text{(A2-Ses)}\quad \frac{T\overset{\mu}{=}U}{\Sigma \vdash [T]\overset{c}{\wedge}[U]} \qquad \text{(A2-Rec1)}\quad \frac{\Sigma, \mu\alpha.T\overset{c}{\wedge}U \vdash T[^{\mu\alpha.T}/_{\alpha}]\overset{c}{\wedge}U}{\Sigma \vdash \mu\alpha.T\overset{c}{\wedge}U} \qquad \text{(A2-Rec2)}\quad \frac{\Sigma, T\overset{c}{\wedge}\mu\alpha.U \vdash T\overset{c}{\wedge}U[^{\mu\alpha.U}/_{\alpha}] = V}{\Sigma \vdash T\overset{c}{\wedge}\mu\alpha.U}$$

Fig. 3. Algorithmic membership checking for $\overset{c}{\wedge}$

are not related to those in Σ: the symbol has just been chosen to remember the fact that this set of assumptions is built from top to bottom as opposed to Σ.) The set of inverse assumptions collects only the pairs of types which allow to apply the assumptions axioms; rules (ALG-Assump1) and (ALG-Assump2). In these cases the algorithm returns (after the $\rightarrow$) a type variable whose name is dependent on the assumption (and on the input types as well). As discussed above all the possible assumptions are μ-guarded since they contain in either the first component or the second component a type starting with a recursion. In this manner, if before the unfolding of a recursion the pair is already present in the set of inverse assumptions the algorithm knows that this is the right time to place a recursion in the result. The recursion added by rules (ALG-Rec1) and (ALG-Rec2) is μ-guarded w.r.t. the same type variable that later allows application of axioms (ALG-Assump1) and (ALG-Assump2). If instead before the unfolding of a recursive type the pair is not present in the set of inverse assumptions, then the rules (ALG-Skip1) and (ALG-Skip2) are applied. The remaining rules simply compute locally the values of the meet, alike those for $\_\overset{c}{\wedge}\_ = \_$.

However, using a free type variable in axioms is a tricky operation and requires additional work to avoid capturing the wrong type variable with recursion. For this reason we need auxiliary definitions in order to guarantee that all type variables returned by (ALG-Rec1) and (ALG-Rec2) are different. We fix a standard naming for the variables to be introduced by (ALG-Assump1) and (ALG-Assump2), which are indeed determined by the exploited assumption present in $\mathfrak{Z}$. Finally, we define the notation for $\mu\mathsf{var}$ to denote the set of variables that can arise from $\mathfrak{Z}$: we let $\mu\mathsf{var}(\emptyset) = \emptyset$ and $\mu\mathsf{var}(T\overset{c}{\wedge}U, \mathfrak{Z}) = \{\alpha_{T,U}\} \cup \mu\mathsf{var}(\mathfrak{Z})$.

The following proposition accounts for the set of free variables of the result:

Proposition 3. *If* $\Sigma \vdash T\overset{c}{\wedge}U \rightarrow V \rhd \mathfrak{Z}$ *then* $\mathsf{fv}(V) = \mu\mathsf{var}(\mathfrak{Z})$.

(A3-END) $\Sigma \vdash \mathtt{end} \overset{c}{\wedge} \mathtt{end} = \mathtt{end}$ (A3-BAS) $\Sigma \vdash \tau \overset{c}{\wedge} \tau = \tau$ (A3-ASSUMP) $\Sigma, T \overset{c}{\wedge} U = V \vdash T \overset{c}{\wedge} U = V$

(A3-OUT)
$$\frac{\forall i, \Sigma \vdash \overline{S_i} \overset{c}{\wedge} \overline{S_i'} = \overline{S_i''} \quad \Sigma \vdash T \overset{c}{\wedge} U = V}{\Sigma \vdash !(S_1, \ldots, S_n).T \overset{c}{\wedge} !(S_1', \ldots, S_n').U = !(S_1'', \ldots, S_n'').V}$$

(A3-IN)
$$\frac{\forall i, \Sigma \vdash S_i \overset{c}{\wedge} S_i' = S_i'' \quad \Sigma \vdash T \overset{c}{\wedge} U = V}{\Sigma \vdash ?(S_1, \ldots, S_n).T \overset{c}{\wedge} ?(S_1', \ldots, S_n').U = ?(S_1'', \ldots, S_n'').V}$$

(A3-SELECT)
$$\frac{K = \{(i,j) \mid l_i = l_j'\} \quad |K| \geq 1 \quad \forall (i,j) \in K. \Sigma \vdash T_i \overset{c}{\wedge} T_j' = U_{(i,j)}}{\Sigma \vdash \&\{l_i : T_i\}_{i \in I} \overset{c}{\wedge} \&\{l_j' : T_j'\}_{j \in J} = \&\{l_i : U_{(i,j)}\}_{(i,j) \in K}}$$

(A3-SES)
$$\frac{T \overset{\mu}{=} U \quad U \overset{\mu}{=} V}{\Sigma \vdash [T] \overset{c}{\wedge} [U] = [V]}$$

(A3-CHOICE)
$$\frac{\begin{array}{c} K = \{(i,j) \mid l_i = l_j'\}, I' = \{(i \in I) \mid \forall j.(i,j) \notin K\}, \\ J' = \{(j \in J) \mid \forall i.(i,j) \notin K\} \end{array} \quad \forall (i,j) \in K. \Sigma \vdash T_i \overset{c}{\wedge} T_j' = U_{(i,j)}}{\Sigma \vdash \oplus\{l_i : T_i\}_{i \in I} \overset{c}{\wedge} \oplus \{l_j' : T_j'\}_{j \in J} = \oplus\{l_i : U_{(i,j)}\}_{(i,j) \in K} \ ; \oplus\{l_i : T_i\}_{i \in I'} \ ; \oplus\{l_j' : T_j'\}_{j \in J'}}$$

(A3-REC1)
$$\frac{\Sigma, \mu\alpha.T \overset{c}{\wedge} U = V \vdash T[^{\mu\alpha.T}/_{\alpha}] \overset{c}{\wedge} U = V}{\Sigma \vdash \mu\alpha.T \overset{c}{\wedge} U = V}$$

(A3-REC2)
$$\frac{\Sigma, T \overset{c}{\wedge} \mu\alpha.U = V \vdash T \overset{c}{\wedge} U[^{\mu\alpha.U}/_{\alpha}] = V}{\Sigma \vdash T \overset{c}{\wedge} \mu\alpha.U = V}$$

(A3-REC3)
$$\frac{\Sigma, T \overset{c}{\wedge} U = \mu\alpha.V \vdash T \overset{c}{\wedge} U = V[^{\mu\alpha.V}/_{\alpha}]}{\Sigma \vdash T \overset{c}{\wedge} U = \mu\alpha.V}$$

Fig. 4. Algorithmic membership checking for $\_ \overset{c}{\wedge} \_ = \_$

(ALG-ASSUMP1) $\Sigma, \mu\alpha.T \overset{c}{\wedge} U \vdash \mu\alpha.T \overset{c}{\wedge} U \rightarrow \alpha_{\mu\alpha.T,U} \rhd \mu\alpha.T \overset{c}{\wedge} U$

(ALG-ASSUMP2) $\Sigma, T \overset{c}{\wedge} \mu\alpha.U \vdash T \overset{c}{\wedge} \mu\alpha.U \rightarrow \alpha_{T,\mu\alpha.U} \rhd T \overset{c}{\wedge} \mu\alpha.U$

(ALG-REC1)
$$\frac{\Sigma, \mu\alpha.T \overset{c}{\wedge} U \vdash T[^{\mu\alpha.T}/_{\alpha}] \overset{c}{\wedge} U \rightarrow V \rhd \daleth, \mu\alpha.T \overset{c}{\wedge} U}{\Sigma \vdash \mu\alpha.T \overset{c}{\wedge} U \rightarrow \mu\alpha_{\mu\alpha.T,U}.V \rhd \daleth}$$

(ALG-SKIP1)
$$\frac{\Sigma, \mu\alpha.T \overset{c}{\wedge} U \vdash T[^{\mu\alpha.T}/_{\alpha}] \overset{c}{\wedge} U \rightarrow V \rhd \daleth \quad \mu\alpha.T \overset{c}{\wedge} U \notin \daleth}{\Sigma \vdash \mu\alpha.T \overset{c}{\wedge} U \rightarrow V \rhd \daleth}$$

(ALG-REC2)
$$\frac{\Sigma, T \overset{c}{\wedge} \mu\alpha.U \vdash T \overset{c}{\wedge} U[^{\mu\alpha.U}/_{\alpha}] \rightarrow V \rhd \daleth, T \overset{c}{\wedge} \mu\alpha.U}{\Sigma \vdash T \overset{c}{\wedge} \mu\alpha.U \rightarrow \mu\alpha_{T,\mu\alpha.U}.V \rhd \daleth}$$

(ALG-SKIP2)
$$\frac{\Sigma, T \overset{c}{\wedge} \mu\alpha.U \vdash T \overset{c}{\wedge} U[^{\mu\alpha.U}/_{\alpha}] \rightarrow V \rhd \daleth \quad T \overset{c}{\wedge} \mu\alpha.U \notin \daleth}{\Sigma \vdash T \overset{c}{\wedge} \mu\alpha.U \rightarrow V \rhd \daleth}$$

(ALG-SELECT)
$$\frac{K = \{(i,j) \mid l_i = l_j'\} \quad |K| \geq 1 \quad \forall i \forall j (i,j) \in K, \quad \Sigma \vdash T_i \overset{c}{\wedge} T_j' \rightarrow U_{(i,j)} \rhd \daleth_{(i,j)}}{\Sigma \vdash \&\{l_i : T_i\}_{i \in I} \overset{c}{\wedge} \&\{l_j' : T_j'\}_{j \in J} \rightarrow \&\{l_i : U_{(i,j)}\}_{(i,j) \in K} \rhd \bigcup_{(i,j) \in K} \daleth_{(i,j)}}$$

(ALG-END) $\Sigma \vdash \mathtt{end} \overset{c}{\wedge} \mathtt{end} \rightarrow \mathtt{end} \rhd \emptyset$

(ALG-OUT)
$$\frac{(\Sigma \vdash \overline{S_i} \overset{c}{\wedge} \overline{S_i'} \rightarrow \overline{S_i''} \rhd \daleth_i)_{1 \leq i \leq n} \quad \Sigma \vdash T \overset{c}{\wedge} V \rightarrow V \rhd \daleth}{\Sigma \vdash !(S_1, \ldots, S_n).T \overset{c}{\wedge} !(S_1', \ldots, S_n').U \rightarrow !(S_1'', \ldots, S_n'').V \rhd \bigcup_{1 \leq i \leq n} \daleth_i \cup \daleth}$$

Fig. 5. The algorithmic meet

Now we prove the soundness and the correctness of the algorithm. All states of the algorithm we consider (with state we mean the instance of a particular premise) are of the form $\Sigma \vdash T\overset{c}{\wedge}U \rightarrow V \rhd \text{Ʒ}$ where Σ and Ʒ are well formed. Moreover we say that a state is completed if the set of inverse assumptions is the empty set. To fix the correspondence between different proof trees, we define the function forget from a set of assumptions $\varSigma$ (set of triples of types), to a set of assumptions Σ (set of pairs of types) as:

$$\mathsf{forget}(\varSigma) = \begin{cases} \mathsf{forget}(\emptyset) = \emptyset \\ \mathsf{forget}(\varSigma', T\overset{c}{\wedge}U = V) = \mathsf{forget}(\varSigma'), T\overset{c}{\wedge}U \end{cases}$$

where $\varSigma, T\overset{c}{\wedge}U = V$ (resp. $\Sigma, T\overset{c}{\wedge}U$, resp. $\text{Ʒ}, T\overset{c}{\wedge}U$) is $\varSigma \cup \{T\overset{c}{\wedge}U = V\}$ (resp. $\Sigma \cup \{T\overset{c}{\wedge}U\}$, resp. $\text{Ʒ} \cup \{T\overset{c}{\wedge}U\}$).

Proposition 4. *Let* $\Sigma \vdash T\overset{c}{\wedge}U \rightarrow V \rhd \text{Ʒ}$ *to be either a completed state or a state in the proof tree of a completed state then there exists a tuple* $\mu\alpha.V'_1, \ldots, \mu\alpha.V'_n$ *s.t.* $\varSigma \vdash T\overset{c}{\wedge}U = V[{}^{\mu\alpha.V'_1}/_{\alpha_1}]\ldots[{}^{\mu\alpha.V'_n}/_{\alpha_n}]$ *where the substitution indexed by i is introduced by the i-th application of either the rule (ALG-Rec1) or the rule (ALG-Rec2) and* $\{\alpha_1, \ldots, \alpha_n\} = \mu\mathsf{var}(\text{Ʒ})$ *and* $\mathsf{forget}(\varSigma) = \Sigma$.

Proof. The proof is by induction on the height of the proof tree of completed states. Notice that we consider only completed states since by assumption $\Sigma \vdash T\overset{c}{\wedge}U \rightarrow V \rhd \text{Ʒ}$ is either a completed state or a state derived from a completed state. From this fact also follows that the order of substitutions is uniquely fixed by the considered proof tree.

The base case is when the only applied rule is (ALG-End), and the statement trivially holds. We sketch the inductive case where (ALG-Rec1) is the last rule applied in the proof tree. We have $\Sigma \vdash T[{}^{\mu\alpha.T}/_{\alpha}]\overset{c}{\wedge}U \rightarrow V \rhd \text{Ʒ}', \mu\alpha.T\overset{c}{\wedge}U$ and by inductive hypothesis $\varSigma \vdash T[{}^{\mu\alpha.T}/_{\alpha}]\overset{c}{\wedge}U = V[{}^{\mu\alpha_1.V'_1}/_{\alpha_1}]\ldots[{}^{\mu\alpha_{n-1}.V'_{n-1}}/_{\alpha_{n-1}}][{}^{\mu\alpha_n.V'_n}/_{\alpha_{\mu\alpha.T,U}}]$ where $\{\alpha_1, \ldots, \alpha_{n-1}\} = \mu\mathsf{var}(\text{Ʒ}')$ and $\mathsf{forget}(\varSigma) = \Sigma$. Now since the value of the variable is not influenced during the running of the algorithm (one can think of $\alpha_{\mu\alpha.T,U}$ as a placeholder) we can assign to $\alpha_{\mu\alpha.T,U}$ the value $\mu\alpha_{\mu\alpha.T,U}.V$ and $V[{}^{\mu\alpha_{\mu\alpha.T,U}.V}/_{\alpha_{\mu\alpha.T,U}}]$ replaces V. Intuitively this operation is done to exhibit the shape of V from the point of view of the $\_\overset{c}{\wedge}\_ = \_$ algorithm since it is one step of unfolding ahead (notice that with this operation both algorithms are similar). Consequently from the other side we must have $\varSigma \vdash T[{}^{\mu\alpha.T}/_{\alpha}]\overset{c}{\wedge}U = V''$ with $V'' = V[\widetilde{{}^{\mu\alpha.V'}/_{\alpha}}][{}^{V[\widetilde{{}^{\mu\alpha.V'}/_{\alpha}}]}/_{\alpha_{\mu\alpha.T,U}}]$ and $[\widetilde{{}^{\mu\alpha.V'}/_{\alpha}}] = [{}^{\mu\alpha_1.V'_1}/_{\alpha_1}]\ldots[{}^{\mu\alpha_{n-1}.V'_{n-1}}/_{\alpha_{n-1}}]$. The thesis follows applying (A3-Rec1) and (A3-Rec3) since we set $\varSigma = \varSigma', \mu\alpha.T\overset{c}{\wedge}U = V'', T\overset{c}{\wedge}U = \mu\alpha.V''$. The following proof trees help understanding this case .

$$\dfrac{\dfrac{\mathsf{forget}(\Sigma''), \mu\alpha.T\overset{c}{\wedge}U \vdash \mu\alpha.T\overset{c}{\wedge}U \to \alpha_{\mu\alpha.T,U} \rhd \mu\alpha.T\overset{c}{\wedge}U}{\vdots}}{\dfrac{\mathsf{forget}(\Sigma), \mu\alpha.T\overset{c}{\wedge}U \vdash T[^{\mu\alpha.T}/_{\alpha}]\overset{c}{\wedge}U \to V \rhd \mu\alpha.T\overset{c}{\wedge}U}{\mathsf{forget}(\Sigma) \vdash \mu\alpha.T\overset{c}{\wedge}U \to \mu\alpha_{\mu\alpha.T,U}.V \rhd \emptyset}}$$

$$\dfrac{\dfrac{\Sigma'', \mu\alpha.T\overset{c}{\wedge}U = \mu\alpha_{\mu\alpha.T,U}.V \vdash \mu\alpha.T\overset{c}{\wedge}U = \mu\alpha_{\mu\alpha.T,U}.V}{\vdots}}{\dfrac{\Sigma', T[^{\mu\alpha.T}/_{\alpha}]\overset{c}{\wedge}U = \mu\alpha_{\mu\alpha.T,U}.V \vdash T[^{\mu\alpha.T}/_{\alpha}]\overset{c}{\wedge}U = V[^{\mu\alpha_{\mu\alpha.T,U}.V}/_{\alpha}]}{\dfrac{\Sigma, \mu\alpha.T\overset{c}{\wedge}U = \mu\alpha_{\mu\alpha.T,U}.V \vdash T[^{\mu\alpha.T}/_{\alpha}]\overset{c}{\wedge}U = \mu\alpha_{\mu\alpha.T,U}.V}{\Sigma \vdash \mu\alpha.T\overset{c}{\wedge}U = \mu\alpha_{\mu\alpha.T,U}.V}}}$$

In the other direction the proposition is simpler since we only make assumptions on the existence of the meet (using the respective relation).

Proposition 5. *If* $\Sigma \vdash T\overset{c}{\wedge}V$ *then* $\exists \text{Ʒ}$ *and* $\Sigma \vdash T\overset{c}{\wedge}V \to U \rhd \text{Ʒ}$.

Proof. The proof is by straightforward induction on the depth of the proof tree. The following is an example tree where the recursion is applied.

$$\dfrac{\dfrac{\Sigma', \mu\alpha.T\overset{c}{\wedge}U \vdash \mu\alpha.T\overset{c}{\wedge}U}{\vdots}}{\dfrac{\Sigma, \mu\alpha.T\overset{c}{\wedge}U \vdash T[^{\mu\alpha.T}/_{\alpha}]\overset{c}{\wedge}U}{\Sigma \vdash \mu\alpha.T\overset{c}{\wedge}U}} \qquad \dfrac{\dfrac{\Sigma', \mu\alpha.T\overset{c}{\wedge}U \vdash \mu\alpha.T\overset{c}{\wedge}U \to \alpha_{\mu\alpha.T,U} \rhd \mu\alpha.T\overset{c}{\wedge}U}{\vdots}}{\dfrac{\Sigma, \mu\alpha.T\overset{c}{\wedge}U \vdash T[^{\mu\alpha.T}/_{\alpha}]\overset{c}{\wedge}U \to V \rhd \mu\alpha.T\overset{c}{\wedge}U, \text{Ʒ}}{\Sigma \vdash \mu\alpha.T\overset{c}{\wedge}U \to \mu\alpha_{\mu\alpha.T,U}.V \rhd \text{Ʒ}}}$$

The final theorem states soundness and correctness of the algorithmic intersection.

Theorem 3. *The algorithm* $\emptyset \vdash T\overset{c}{\wedge}V \to U \rhd \emptyset$ *computes the meet of* T *and* V.

Proof. By Proposition 4 we know that each value computed by the algorithm belongs to a meet relation. Notice that the proposition is applied since the set of inverse assumptions (Ʒ) is empty and thus a completed state. Again, since Ʒ, the two computed values coincide. By the Proposition 5 we know that the algorithm exhibits a result for each T and V for which the meet is defined and by Proposition 3, $\mathsf{fv}(U) = \emptyset$. Then in order to not contradict Theorem 1 each value U in a meet relation, as a result of T and V must be in $\overset{\mu}{=}$-relation with the value computed by the algorithm.

Example 4. We show how the algorithm computes the meet of $T = \mu\alpha.\mu\beta.\&\{s1 : \alpha; s2 : \beta\}$ and $U = \mu\alpha.\mu\beta.\&\{s1 : \alpha; s3 : \beta\}$. The intersection computed in Fig. 6 is μ-equal to $\mu\alpha_{T,U}.\&\{s1 : \alpha_{T,U}\}$ with T_1 and U_1 resulting from the unfolding and ... indicating any set of assumptions.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{T\overset{c}{\wedge}U, \ldots \vdash T\overset{c}{\wedge}U \rightarrow \alpha_{T,U} \rhd T\overset{c}{\wedge}U}{T\overset{c}{\wedge}U, \ldots \vdash \&\{s1 : \alpha; s2 : \beta\}[^{T}/_{\alpha}][^{T_1}/_{\beta}]\overset{c}{\wedge}\&\{s1 : \alpha; s3 : \beta\}[^{U}/_{\alpha}][^{U_1}/_{\beta}] \rightarrow \&\{s1 : \alpha_{T,U}\} \rhd T\overset{c}{\wedge}U}}{\vdots}}{T\overset{c}{\wedge}U, \ldots \vdash \&\{s1 : \alpha; s2 : \beta\}[^{T}/_{\alpha}][^{T_1}/_{\beta}]\overset{c}{\wedge}\mu\alpha.\mu\beta.\&\{s1 : \alpha; s3 : \beta\} \rightarrow \&\{s1 : \alpha_{T,U}\} \rhd T\overset{c}{\wedge}U}}{T\overset{c}{\wedge}U \vdash \mu\beta.\&\{s1 : \alpha; s2 : \beta\}[^{T}/_{\alpha}]\overset{c}{\wedge}\mu\alpha.\mu\beta.\&\{s1 : \alpha; s3 : \beta\} \rightarrow \&\{s1 : \alpha_{T,U}\} \rhd T\overset{c}{\wedge}U}}{\emptyset \vdash \mu\alpha.\mu\beta.\&\{s1 : \alpha; s2 : \beta\}\overset{c}{\wedge}\mu\alpha.\mu\beta.\&\{s1 : \alpha; s3 : \beta\} \rightarrow \mu\alpha_{T,U}.\&\{s1 : \alpha_{T,U}\} \rhd \emptyset}$$

Fig. 6. Algorithmic meet at work

Notice that the given algorithm does not require backtracking (i.e. it is syntax directed) and the set of inference rules reported in Fig. 5 constitutes in fact a real algorithm (see Appendix A for the OCaml code), given that it fails if no premise is satisfied. When the algorithm fails it means that no intersection of two types exists, because Theorem 5 together with Theorem 1 logically imply the soundness and the completeness of our algorithm.

6 Conclusion

As witnessed by the profusion of works about session types [18, 6, 7, 9, 10] (just to cite few), they provide us with an important paradigm to type communication protocols. In this paper we have introduced the meet operation for session types, starting from an algebraic setting as the greatest lower bound of the subtyping relation proposed by Gay and Hole. Subsequently we have given a co-inductive definition relation $\_\overset{c}{\wedge}\_ = \_$ on triples and we have shown that $U\overset{c}{\wedge}T = V$ if and only if the greatest lower bound of U and T is V. The *meet exist* relation $\overset{c}{\wedge}$ instead, contains all the pairs for which the greatest lower bound exists. The co-inductive definition of these two relations gives a hint on how to locally compute the meet but it does not exhibit the representation as a finite regular session type. A further step is that for co-inductive relations the algorithm for checking the membership based on set of assumptions, is standard [1, 8, 14]. The existence of these algorithms says that if the meet is finite then we have a way of checking its appropriateness. These algorithms proceed by exploring all the possible combinations of sub-elements among the input elements until an already explored combination is found and this is exactly the behavior of the meet. Our algorithm $\emptyset \vdash T\overset{c}{\wedge}U \rightarrow V \rhd \emptyset$ is built on this key observation. To prove its correctness we show that the relation $\_\overset{c}{\wedge}\_ = \_$ is able to simulate all the input triples in the algorithm. Vice versa the algorithm is capable to exhibit a result for each pair of type that has the greatest lower bound. The soundness result follows from the uniqueness, up to $\overset{\mu}{=}$, of the greatest lower bound.

Even if in the course of this paper we have focused on the notion of the greatest lower bound we the notion of the least upper bound comes automatically thanks to the syntactic notion of duality. Having a way to compute meet and join for certain types is very useful. In fact, these notions are not limited to the static type checking but can be also used on a scenario where session types represent contracts in the sense of [15, 5] or in the sense of [13, 2].

References

1. R. Amadio and L. Cardelli. Subtyping recursive types. In *Proceedings of POPL'91: 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 104–118. ACM, 1991.
2. L. Bettini, M. Coppo, L. D'Antoni, M. De Luca, M. Dezani-Ciancaglini, and N. Yoshida. Global progress in dynamically interleaved multiparty sessions. In *Proceedings of CONCUR'08, 19th International Conference on Concurrency Theory*, Lect. Notes in Comput. Sci. Springer Verlag, 2008. (to appear).
3. E. Bonelli, A. Compagnoni, and E. Gunter. Correspondence assertions for process synchronization in concurrent communications. *J. Funct. Program.*, 15(2):219–247, 2005.
4. R. Bruni and L. Mezzina. Types and deadlock freedom in a calculus of services, sessions and pipelines. In G. Rosu and J. Meseguer, editors, *Proceedings of AMAST'08, 12th International Conference on Algebraic Methodology and Software Technology*, volume 5140 of *Lect. Notes in Comput. Sci.*, pages 100–115. Springer Verlag, 2008.
5. G. Castagna, N. Gesbert, and L. Padovani. A theory of contracts for web services. In *Proceedings of POPL'08, 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 261–272, New York, NY, USA, 2008. ACM.
6. M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopoulou. Session types for object-oriented languages. In D. Thomas, editor, *Proceedings of ECOOP'06, 20th European Conference on Object-Oriented Programming*, volume 4067 of *Lect. Notes in Comput. Sci.*, pages 328–352. Springer Verlag, 2006.
7. M. Dezani-Ciancaglini, d. U., and U. Yoshida. On progress for structured communications. In G. Barthe and C. Fournet, editors, *Proceedings of TGC'07, 3rd Symposium on Trustworthy Global Computing*, volume 4912 of *Lect. Notes in Comput. Sci.*, pages 257–275. Springer Verlag, 2008.
8. V. Gapeyev, M. Levin, and B. Pierce. Recursive subtyping revealed: (functional pearl). In *Proceedings of ICFP'00, 5th ACM SIGPLAN International Conference on Functional programming*, pages 221–231. ACM, 2000.
9. P. Garralda, A. Compagnoni, and M. Dezani-Ciancaglini. Bass: boxed ambients with safe sessions. In A. Bossi and M. J. Maher, editors, *Proceedings of PPDP'06, 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 61–72. ACM, 2006.
10. S. Gay. Bounded polymorphism in session types. *Mathematical Structures in Computer Science*, 2008. (to appear).
11. S. Gay and M. Hole. Subtyping for session types in the pi calculus. *Acta Inform.*, 42(2):191–225, 2005.
12. K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In C. Hankin, editor, *Proceedings of ESOP'98, 7th European Symposium on Programming*, volume 1381 of *Lect. Notes in Comput. Sci.*, pages 122–138. Springer Verlag, 1998.
13. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *Proceedings of POPL'08, 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 273–284. ACM, 2008.

14. D. Kozen, J. Palsberg, and M. Schwartzbach. Efficient recursive subtyping. In *Proceedings of POPL'93, 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 419–428. ACM, 1993.
15. C. Laneve and L. Padovani. The must preorder revisited. In L. Caires and V. Vasconcelos, editors, *Proceedings of CONCUR'07, 18th International Conference on Concurrency Theory*, volume 4703 of *Lect. Notes in Comput. Sci.*, pages 212–225. Springer Verlag, 2007.
16. R. Milner, J. Parrow, and J. Walker. A calculus of mobile processes, I and II. *Inform. and Comput.*, 100(1):1–40,41–77, 1992.
17. B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. In *Proceedings of LICS'93, 8th IEEE Symposium on Logic in Computer Science*, pages 376–385. IEEE Computer Society, 1993.
18. V. Vasconcelos, S. Gay, and A. Ravara. Type checking a multithreaded functional language with session types. *Theor. Comput. Sci.*, 368(1-2):64–87, 2006.
19. N. Yoshida and V. Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. *Elect. Notes in Th. Comput. Sci.*, 171(4):73–93, 2007.

A Intersection Algorithm Code

We report some OCaml code of the intersection algorithm. The algorithm works on Session Types built with the following data structures:

```
type types=
(*type of the non linear names*)
   Sessionref of types
(*input of a list of type with the continuation*)
 | Inputt of types list * types
(*output ff a list of type with the continuation*)
 | Outputt of types list * types
(*external choice list of label guarded continuation*)
 | Selectt of (string * types) list
(*internal choice list of label guarded continuation*)
 | Choicet of (string * types) list
(*type variable*)
 | Vart of string
(*basic data type*)
 | BasicT
 | Endt
(* recursion *)
 | Rect of string * types
```

and additionally the following functions and declarations are needed:

```
val intersect: ('a * 'b) list ->
     ('a * 'c) list -> ('a * 'b * 'c) list
val notintersect: ('a * 'b) list ->
    ('a * 'c) list -> ('a * 'b) list
val muequal: types -> types -> bool
val dual: types list -> types list
val strtype: types -> string
```

Function `intersect` takes either two internal or two external choices and returns a list of common labels and the respective two continuations for these labels. Function `notintersect` takes either two internal or two external choices and returns a list of labels and continuations of the first type which do not match any label in the second type. Function `muequal` checks the $\stackrel{\mu}{=}$-equality, function `dual` computes the syntactic dual of a session type and Function `strtype` returns a unique string representation for each type. Finally

`TypecoupleSet` is an instance of the `Set`-functor with pairs of types to be used as a set of assumptions Σ and as a set of reverse assumptions $\reflectbox{$\Sigma$}$. The core of the algorithm is the function `typemeet` which takes a couple of types, a set of assumptions and returns a type with a set of inverse assumptions. Finally the complete algorithm:

```
(* return the intersection of t and t1
 together with the set of reverse assumptions *)
 let rec typemeet (t,t1,assump)=
 if TypecoupleSet.mem (t,t1) assump then (  match (t,t1) with
  (*add an assumption in the reverse set*)
  (Rect(_,_),_)|(_,Rect(_,_))->Vart((strtype t)^(strtype t1)),
     TypecoupleSet.add (t,t1) TypecoupleSet.empty )
  else try
   match (t,t1) with
   (BasicT,BasicT)->(BasicT, TypecoupleSet.empty)
   |(Endt,Endt)->(Endt, TypecoupleSet.empty)
   |(Sessionref(t2),Sessionref(t3))->
      if (muequal t2 t3) then
         (t1, TypecoupleSet.empty)
      else raise (Match_failure, "", "")
   |(Inputt(values,s),Inputt(values1,s1))->
      let (values2,reverse)= (List.fold_right2 (fun t t1 (t2tmp,revtmp)->
                let (t2,rev)= typemeet(t,t1,assump) in
                   (t2@t2tmp),(TypecoupleSet.union rev revtmp))
              values values1 TypecoupleSet.empty) in
        let (s2,reverse1)=  typemeet(s,s1,assump) in
          (Inputt(values2,s2),(TypecoupleSet.union reverse reverse1))
   |(Outputt(values,s),Outputt(values1,s1))->
      let (values2,reverse)= (List.fold_right2 (fun t t1 (t2tmp,revtmp)->
                let (t2,rev)= typemeet(t,t1,assump) in
                   (t2@t2tmp),(TypecoupleSet.union rev revtmp))
              (dual values) (dual values1) TypecoupleSet.empty) in
        let (s2,reverse1)=  typemeet(s,s1,assump) in
          (Outputt((dual values2),s2),(TypecoupleSet.union reverse reverse1))
   |(Selectt(values),Selectt(values1))->
      let values2=(intersect values values1) in
        (*recursively compute the meet on the common branches intersection set K *)
        let res= (List.map (function (x,y,z)-> typemeet(y,z,assump)) values2) in
         if (res=[]) then
            raise  Empty_intersection
         else  (*build the correct list both from the intersection
            and from the recursive invocations*)
          ((Selectt(List.map2 (fun (x,y,z) (y1,reverse) ->(x,y1)) values2 res),
            (*compute the union of the reverse assumptions *)
            (List.fold_right (fun (x,reverse) y->
                 (TypecoupleSet.union reverse y))
             res TypecoupleSet.empty)))
   |(Choicet(values),Choicet(values1))->
       let values2=(intersect values values1) in
       (*recursively compute the meet on the common branches intersection set K *)
        let res= (List.map (function (x,y,z)-> (typemeet (y,z,assump))) values2) in
          (*build the correct list both from the intersection, from non itersection
           of both types and from the recursive invocations*)
          (Choicet((List.map2 (fun (x,y,z) (y1,reverse)->(x,y1)) values2 res)@
                  (notintersect values values1) @(notintersect values1 values )),
          (*compute the union of the reverse assumptions *)
          (List.fold_right (fun (x,reverse) y->
                 (TypecoupleSet.union last y))
           res TypecoupleSet.empty))
   |(Rect(id,t),t1)->
       let (res,reverse)=typemeet((unfold t id (Rect(id,t))),t1,
                      (TypecoupleSet.add (Rect(id,t),t1) assump)) in
         if (TypecoupleSet.mem (Rect(id,t),t1) reverse)
         then  (*if the assumption is in the inverse
           set of assumptions then add a recursion*)
           Rect(((strtype t)^(strtype t1)),res),(TypecoupleSet.remove (Rect(id,t),t1) reverse)
         else (*else skip*)  (res,reverse)
   |(t,Rect(id,t1))->
       let (res,reverse)=typemeet(t,(unfold t1 id (Rect(id,t1))),
                 (TypecoupleSet.add (t,Rect(id,t1)) assump) ) in
        if (TypecoupleSet.mem (t,Rect(id,t1)) reverse)
        then   (*if the assumption is in the inverse
           set of assumptions then add a recursion*)
          Rect(((strtype t)^(strtype t1)),res),(TypecoupleSet.remove (t,Rect(id,t1)) reverse)
        else (*else skip*)  (res,last)
   with Match_failure(_)->(Printf.printf "meet error!")
```

B Applications

In this section we show how to use the meet for enhancing two popular session typing systems to deal with the if-then-else construct and at the end we show how to use the meet for a service-oriented calculus to obtain a fully deterministic type system. It is interesting to consider all three case studies, because each has its own peculiarities and requires some specific treatment. It makes little sense to give full details about the syntax, semantics, and type system of each considered language in this appendix, hence we try to focus just on the most relevant aspects with some flashes of code and rules. As we shall see, three main factors may influence the use of meet in a language: the definition of duality relation, the degree of non-determinism in the typing rule for both choices and the use of subtyping to allow subsumption.

A particularity of session typing systems is that they allow to use many (only two in the case of [4]) interleaved opened sessions at the same time but the linear typing environment tracks each session separately without considering dependencies. For this reason we need the predicate $P\langle\langle\zeta, Q\rangle\rangle$ to mean that the first free occurrence of session ζ as a prefix in P is Q; for example given the process $P = \mathtt{accept}\ a(k).k?(y).k'!(5).k'!(6)$ then both $P\langle\langle k, Q\rangle\rangle$ (for any Q) and $P\langle\langle k', k'!(6)\rangle\rangle$ do not hold (the former since k is bound and the latter since $k'!(6)$ is not the minimal subprocess of P using k') but $P\langle\langle k', k'!(5).k'!(6)\rangle\rangle$ holds. The full definition of $\_\langle\langle\_,\_\rangle\rangle$ is omitted since it is straightforward and assumes that linearity of sessions is achieved in some way especially w.r.t. the rule for parallel composition, i.e. $P'\langle\langle\zeta, P|Q\rangle\rangle$ expects only at least one (in the sense of exclusive or) among P and Q satisfies the predicate. Keep in mind that each session-typed language provides primitives for in-session communications able to generate the session types: i.e., input/output, internal/external choices. Consequently Q in $P\langle\langle\zeta, Q\rangle\rangle$ is always one of those actions performed on the session ζ.

B.1 The Gay-Hole Session Typing System

The first system we consider is due to Simon Gay and Malcolm Hole who studied session types for the π-calculus. The meet operation is used to type the addition of the if-then-else construct. We first introduce the syntax of the calculus.

$$
\begin{array}{rll}
P, Q ::= & \mathbf{0} & \text{(terminated process)} \\
\mid & P|Q & \text{(parallel combination)} \\
\mid & !P & \text{(replication)} \\
\mid & x^p?[y_1 : T_1, \ldots, y_n : T_n].P & \text{(input)} \\
\mid & x^p![y_1^{p_1}, \ldots, y_n^{p_n}].P & \text{(output)} \\
\mid & (\nu x : T)P & \text{(channel creation)} \\
\mid & x^p \triangleright \{l_1 : P_1, \ldots, l_n : P_n\} & \text{(branch)} \\
\mid & x^p \triangleleft l.P & \text{(choice)} \\
\mid & \mathtt{if}\ x = y\ \mathtt{then}\ P\ \mathtt{else}\ Q & \text{(if then else)}
\end{array}
$$

The polarity $p \in \{+, -\}$ is optional. If p is missed then the use of x is intended as a traditional (non-linear) π-calculus channel. Otherwise, if x is polarized with

p, the operation is intended as an in-session communication. The typing judgments are of the form $\Gamma \vdash P$ where Γ is a type environment. In this type system Γ contains assumptions on both linear and non-linear names. In case of linear name if both $x^+ : S$ and $x^- : S'$ are in Γ then it must be $S \bot S'$ where $\bot$ represents the co-inductive duality relation. The $\bot$ relation is similar to the syntactic one but is co-inductively defined to hold on the infinite trees generated by the recursive types.

Definition 7. *A relation $R \subseteq TYPE \times TYPE$ is a duality relation if $(T, U) \in R$ implies the following conditions:*

1. *if* $\mathsf{unfold}(T) = !(S_1, \ldots, S_n).T'$ *then* $\mathsf{unfold}(U) = ?(S'_1, \ldots, S'_n).U'$ *and for all* $i \in \{1, \ldots n\}$, $S_i \overset{\mu}{=} S'_i$ *and* $(T', U') \in R$
2. *if* $\mathsf{unfold}(T) = ?(S_1, \ldots, S_n).T'$ *then* $\mathsf{unfold}(U) = !(S'_1, \ldots, S'_n).V'$ *and for all* $i \in \{1, \ldots n\}$, $S_i \overset{\mu}{=} S'_i$ *and* $(T', U') \in R$
3. *if* $\mathsf{unfold}(T) = \&\{l_1 : T_1, \ldots, l_m : T_m\}$ *then* $\mathsf{unfold}(U) = \oplus\{l'_1 : U_1, \ldots, l'_m : U_m\}$ *and for all* $i \in \{1, \ldots, m\}$ *exists one* $j \in \{1, \ldots, m\}$ *such that* $l'_i = l_j$ *and* $(T_i, U_j) \in R$
4. *if* $\mathsf{unfold}(T) = \oplus\{l_1 : T_1, \ldots, l_m : T_m\}$ *then* $\mathsf{unfold}(U) = \&\{l'_1 : V_1, \ldots, l'_m : V_m\}$ *and for all* $i \in \{1, \ldots, m\}$ *exists one* $j \in \{1, \ldots, m\}$ *such that* $l'_i = l_j$ *and* $(T_i, U_j) \in R$
5. *if* $\mathsf{unfold}(T) = \mathtt{end}$ *then* $\mathsf{unfold}(U) = \mathtt{end}$

The co-inductive $\bot$ relation is defined as $T \bot U$ if there exists a duality relation such that $(T, U) \in R$

As a direct consequence of the definition of $\bot$, observe that $\overline{T} \bot T$ (proved in [11]). The fact that two sessions with dual polarity are in the environment together with the subject reduction guarantees no immediate communication errors, that is, for each session each party makes the dual actions w.r.t. the party at the opposite side. As a peculiarity of the type system, let us remark that the typing rules for choices allow maximal freeness, i.e. one can add (in T-Choose) and remove (in T-Offer) branches until duality is achieved:

$$\text{(T-CHOOSE)}\quad \frac{l = l_i \in \{l_i, \ldots, _n\} \quad \Gamma, x^p : T_i \vdash P}{\Gamma, x^p : \oplus\{l_i : T_i\}_{1 \leq i \leq n} \vdash x^p \triangleleft l.P}$$

$$\text{(T-OFFER)}\quad \frac{m \leq n \quad \forall i \in \{1, \ldots, m\}.(\Gamma, x^p : T_i \vdash P_i)}{\Gamma, x^p : \&\{_i : T_i)_{1 \leq i \leq m} \vdash x^p \triangleright \{l_i : P_i\}_{1 \leq i \leq n}}$$

Next theorem shows the correctness of the meet operation for this type system: if a session type in the environment is the result of a meet of two types T and U then it is always possible to maintain the typability substituting this session type with either T or U.

Theorem 4. *If $\Gamma, x^p : S, x^{\overline{p}} : S' \vdash P$ and $S \bot S'$ and for all T and U s.t. $T \overset{c}{\wedge} U = S$ then $\Gamma, x^p : T, x^{\overline{p}} : \overline{T} \vdash P$*

Proof. The proof is by induction on the typing judgment of $\Gamma, x^p : S, x^{\overline{p}} : S' \vdash P$. We sketch the case where the last rule applied to type x^p is the output. It must be the case that $\Gamma, x^p :!(\tilde{W}).S_1, x^{\overline{p}} :?(\tilde{W}').S_2 \vdash P$ where $\mathsf{unfold}(S') = x^{\overline{p}} :?(\tilde{W}').S_2$ and $!(\tilde{W}).S_1 \bot ?(\tilde{W}').S_2$ and $!(\tilde{W}'').T_1 \overset{c}{\wedge} !(\tilde{W}''').U_1 = !(\tilde{W}).S_1$ with $\mathsf{unfold}(T) = !(\tilde{W}'').T_1$ and $\mathsf{unfold}(U) = !(\tilde{W}''').U_1$. By definition of $\_\overset{c}{\wedge}\_ = \_$ we have $\overline{\tilde{W}} \leq \overline{\tilde{W}}''$ equal to $\tilde{W}'' \leq \tilde{W}$. Moreover by definition of $\bot$, $\tilde{W} \overset{\mu}{=} \tilde{W}'$ and by definition of $\leq$, $\tilde{W}'' \leq \tilde{W}'$. Given this typing can be easily proved that
$P \equiv (\nu \tilde{y})(P_1 \langle\!\langle x^{\overline{p}}, x^{\overline{p}}?[\tilde{y} : \tilde{W}'_{\geq}].P' \rangle\!\rangle | P_2 \langle\!\langle x^p, x^p![\tilde{y}^{\tilde{q}}].P'' \rangle\!\rangle | P_3)$ for some $W' \leq W'_{\geq}$, $\tilde{y}$, P', P'', P_1, P_2 and P_2 (we can reason up to $\equiv$ since the subject congruence holds for the original type system). The instance of rule (T-Ins) used to type $\Gamma', x^{\overline{p}} : ?(\tilde{W}').S_2 \vdash x^{\overline{p}}?[\tilde{y} : \tilde{W}'_{\geq}].P'$ has the form

$$\frac{\Gamma', x^{\overline{p}} : S_2, \tilde{y} : \tilde{W}'_{\geq} \vdash P' \quad \tilde{W}' \leq \tilde{W}'_{\geq}}{\Gamma', x^{\overline{p}} :?[\tilde{W}'].S_2 \vdash x^{\overline{p}}?[\tilde{y} : \tilde{W}'_{\geq}].P'}\ \text{(T-INS)}$$

and it allows to use $\tilde{W}''$ in place of $\tilde{W}'$ since $\tilde{W}'' \leq \tilde{W}' \leq \tilde{W}'_{\geq}$.

The result follows by the application of the inductive hypothesis on $\Gamma, x^p : S_1, x^{\overline{p}} : S_2 \vdash (\nu \tilde{y})(P'|P''|P_3)$. The statement for output follows since the rule (T-OutS) is covariant and the statements involving choices follows since the rule (T-Choose) allows to add arbitrary branches in the external choice and the rule (T-Offer) allows to select a subset of the choices.

The correctness of the typing rule for the if construct follows from the previous theorem.

$$\frac{\Gamma_1 \vdash P \quad \Gamma_2 \vdash Q \quad \Gamma = \Gamma_1 \overset{c}{\wedge} \Gamma 2}{\Gamma \vdash \mathtt{if}\ x = y\ \mathtt{then}\ P\ \mathtt{else}\ Q}\ \text{(T-IF)}$$

where

$$\begin{aligned}\Gamma_1 \overset{c}{\wedge} \Gamma 2 = \{&(x^p : U) | x^p : S \in \Gamma_1 \text{ and } x^p : T \in \Gamma_2 \text{ and}\\ &\emptyset \vdash S \overset{c}{\wedge} T \to U \rhd \emptyset\}\\ \cup\{&(x^p : S) | x^p \in \Gamma_i \setminus \Gamma_{i+1\, mod\, 2}\}\end{aligned}$$

The other typing rules with respect to [11] stay unchanged and thanks to Theorem 4 (applied as many time is necessary) the subject reduction given for the original type system also holds for the new one.

B.2 The Honda-Vasconcelos-Kubo Session Typing System

Next, we show how to use the meet for the system given in [12] to replace the if-then-else rule. The syntax of the calculus is introduced by means of the example in the introduction. Typing judgments have the form $\Theta; \Gamma \vdash P \rhd \Delta$ where Θ is an environment containing assumptions about process variables (used to type recursive processes), Γ is the standard environment for non linear names (here called ports, like the `db` in the example) and Δ is the environment for session usages (like `k` in the example). Similarly to the previous type system, rule for external choices allows to add arbitrary branches. Differently from previous type system, here subtyping is not used on the sent/received values, the employed duality relation is the syntactic one and the rule for external choice does not allow to select an arbitrarily subset of the branches. From these diversities the equivalent of theorem 4 does not hold directly in this type system. For clarity we report the rule for choices and the original rule for the if-then-else construct

$$\text{(BR)}\ \frac{\Theta; \Gamma \vdash P_1 \triangleright \Delta, k : S_1 \ldots \Theta; \Gamma \vdash P_n \triangleright \Delta, k : S_n}{\Theta; \Gamma \vdash k \triangleright \{l_1 : P_1 \parallel \ldots \parallel l_n : P_n\} \triangleright \Delta, k : \{l_1 : S_1, \ldots, l_n : S_n\}}$$

$$\text{(SEL)}\ \frac{\Theta; \Gamma \vdash P \triangleright \Delta, k : S_j \quad 1 \leq j \leq n}{\Theta; \Gamma \vdash k \triangleleft l_j; P \triangleright \Delta, k : \oplus\{l_i : S_1, \ldots, l_n : S_n\}}$$

$$\text{(IF)}\ \frac{\Gamma \vdash e \triangleright bool \quad \Theta; \Gamma \vdash P \triangleright \Delta \quad \Theta; \Gamma \vdash Q \triangleright \Delta}{\Theta; \Gamma \vdash \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q \triangleright \Delta}$$

To deal with the lack of subtyping we introduce a new partial order $\leq'$ which allows only the syntactic equality on the sent and received values within a session and on which we built a new meet operation $\_\overset{\tilde{c}}{\wedge}\_ = \_$ suitable for this system.

Definition 8. *A relation $\mathcal{R} \subseteq TYPE \times TYPE$ is a width type simulation relation if $(T, V) \in \mathcal{R}$ implies the following conditions:*

- *if* $\mathsf{unfold}(T) = !(S_1, \ldots, S_n).T'$ *then* $\mathsf{unfold}(V) = !(S_1, \ldots, S_n).V'$ *and* $(T', V') \in \mathcal{R}$
- *if* $\mathsf{unfold}(T) = ?(S_1, \ldots, S_n).T'$ *then* $\mathsf{unfold}(V) = ?(S_1, \ldots, S_n).V'$ *and* $(T', V') \in \mathcal{R}$
- *the remaining rules are the same as in Definition 2.*

The co-inductive $\leq'$ relation is defined as $T \leq' V$ if exists a width type simulation relation $\mathcal{R}$ s.t. $(T, V) \in \mathcal{R}$

Consequently the new meet operation $\_\overset{\tilde{c}}{\wedge}\_ = \_$ is defined as:

Definition 9. *A relation $\mathcal{R} \subseteq TYPE \times TYPE \times TYPE$ is a width meet relation if $(T, V, U) \in \mathcal{R}$ implies the following conditions:*

- *if* $\mathsf{unfold}(T) = !(S_1, \ldots, S_n).T'$ *then* $\mathsf{unfold}(V) = !(S_1, \ldots, S_n).V'$ *and* $\mathsf{unfold}(U) = !(S_1, \ldots, S_n).U'$ *and* $(T', V', U') \in \mathcal{R}$
- *if* $\mathsf{unfold}(T) = ?(S_1, \ldots, S_n).T'$ *then* $\mathsf{unfold}(V) = ?(S_1, \ldots, S_n).V'$ *and* $\mathsf{unfold}(U) = ?(S_1, \ldots, S_n).U'$ *and* $(T', V', U') \in \mathcal{R}$
- *the remaining rules are the same as in Definition 5.*

The co-inductive $\_\overset{\tilde{c}}{\wedge}\_ = \_$ relation is defined as $T \overset{\tilde{c}}{\wedge} V = U$ if exists a width meet relation $\mathcal{R}$ s.t. $(T, V, U) \in \mathcal{R}$.

Notice that in both relations we cannot even demand for $\stackrel{\mu}{=}$-equality of the sent and received values since we are using a syntactic duality relation.

This time the technique employed to maintain session linearity does not use polarities to distinguish between dual session sides but use a special type $\perp$ (different from the co-inductive duality relation) in the typing rule for parallel composition to means that $\perp$ is the result of the parallel composition of two different sessions with dual types. Intuitively, this means (together with the subject reduction) that P and Q can be composed in parallel guaranteeing communication safety. Our correctness theorem for this system reflects this choice. Two different well typed processes P and Q which have the same common session k with a dual type in the respective linear environment, can be still correctly typed if we substitute P with a process P_T which uses k with a greater type (in the sense of $\leq$). The needed to introduce both an another process P_T and a different result type V is due to the fact that the rule (Br) inserts in the type all the offered choices.

Theorem 5. *If $\Theta; \Gamma \vdash P \triangleright \Delta, k : S$ and $\Theta; \Gamma \vdash Q \triangleright \Delta', k : \overline{S}$ then for all T and U s.t. $T \stackrel{\tilde{c}}{\wedge} U = S$ and P_T s.t. $\Theta; \Gamma \vdash P_T \triangleright \Delta, k : T$, $\exists V$ s.t. $\Theta; \Gamma \vdash P_T \triangleright \Delta, k : V$ and $\Theta; \Gamma \vdash Q \triangleright \Delta', k : \overline{V}$*

Proof. The proof is by induction on the typing of $\Theta; \Gamma \vdash P \triangleright \Delta, k : S$ and $\Theta; \Gamma \vdash Q \triangleright \Delta, k : \overline{S}$. We sketch the case where the last applied rule to type k is an internal choice and then $S = \oplus\{l_1 : S_1, \ldots, l_n : S_n\}$ and $\overline{S} = \{l_1 : \overline{S_1}, \ldots, l_n : \overline{S_n}\}$. This also means that $P\langle\langle k, k \triangleleft l; P_1\rangle\rangle$ with $l \in \{l_1, \ldots, l_n\}$ and $Q\langle\langle k, k \triangleright \{l_1 : Q_1 \parallel \ldots \parallel l_n : Q_n\}\rangle\rangle$. Moreover, since $S \leq T$, $\mathsf{unfold}(T) = \oplus\{l'_1 : T_1, \ldots, l'_m : T_m\}$ with $m \leq n$ and $\forall i \in \{1, \ldots, m\}$ exists $j \in \{1, \ldots, n\}$ s.t. $l'_j = l_i$ and $T_i \leq S_j$ which implies $P_T\langle\langle k, k \triangleleft l_h; P'_T\rangle\rangle$ with $l_h \in \{l_1, \ldots, l_n\}$. Applying the induction hypothesis on $\Theta; \Gamma \vdash P_1 \triangleright \Delta, k : S_h$ and $\Theta; \Gamma \vdash Q_h \triangleright \Delta, k : \overline{S_h}$ gives a type V_h and the result follows since: $\Theta, \Gamma \vdash P_T \triangleright \Delta, k : \oplus\{l_1 : S_1, \ldots, l_h : V_h \ldots, l_n : S_n\}$ and $\Theta, \Gamma \vdash Q \triangleright \Delta', k : \&\{l_1 : \overline{S_1}, \ldots, l_h : \overline{V_h} \ldots, l_n : \overline{S_n}\}$.

Previous theorem states that replacing rule (If) with the following one keep the subject reduction:

$$\text{(If)}\quad \frac{\Gamma \vdash e \triangleright bool \quad \Theta; \Gamma \vdash P \triangleright \Delta_1 \quad \Theta; \Gamma \vdash Q \triangleright \Delta_2 \quad \Delta = \Delta_1 \stackrel{\tilde{c}}{\wedge} \Delta_2}{\Theta; \Gamma \vdash \texttt{if } e \texttt{ then } P \texttt{ else } Q \triangleright \Delta}$$

where $\Delta_1 \stackrel{\tilde{c}}{\wedge} \Delta_2$ is defined pointwise as before.

B.3 The Bruni-Mezzina Session Typing System

In this section we present a fully deterministic typing system for session usage that is, we remove the nondeterminism in the rule for external choices thanks to the meet relation. We present the idea for the system in [4] however it can be easily adapted in all the session type systems.

One can think of this language as a particular instance of the language described in the previous subsection. The non linear names are called services and the operations of service invocation $a.P$ and service definition $\overline{a}.P$ are similar to the `accept`/`request` primitives. The main difference is that the session channel k used for communications with the other party is implicitly given. Consequently one can use communication primitives $\langle l\rangle.P$ (internal choice of l), $\Sigma_{i=1}^{n}(l_i).P_i$ (external choice), $(\tilde{x}).P$ (variable input) and $\langle\tilde{e}\rangle.P$ (expression output) to synchronize with the partner without the session channel k in the syntax. It is worth noticing that there are not any primitives for sessions delegation, this is a important fact since the lack of subtyping is not influent, that is we can use $\_\overset{c}{\wedge}\_ = \_$ as meet. Moreover, one can output to the previously opened session with the primitive $\mathtt{return}\ \tilde{e}.P$. Previously opened sessions are due to previous service invocations/definitions. As an example consider the `db` service in the introduction, it becomes:

$$
\begin{array}{l}
\mathtt{db.}\big(\mathtt{(}\mathbf{query}\mathtt{).(}sqlvar\mathtt{).}P_1+\\
\quad\mathtt{(}\mathbf{update}\mathtt{).}P_2+\\
\quad\mathtt{(}\mathbf{add}\mathtt{).}P_3\big)\\
\overline{\mathtt{db}}\mathtt{.}\\
\quad\mathtt{if}\ test\ \mathtt{then}\ \langle\mathbf{query}\rangle.\langle sqlstr\rangle\mathtt{.}P\\
\quad\mathtt{else}\langle\mathbf{update}\rangle.Q
\end{array}
$$

Consider now that P wishes to communicate the result of the query to someone else different from its partner, then it can use the `return` primitive to output the result upon the previously open session. For example, if after the invocation to `db` one have contacted another service `log` to log the result ($\overline{\mathtt{log}}.\overline{\mathtt{db}}.\mathtt{if}\ test\ldots$) then each return inside `db` is viewed as an output in the session created by $\overline{\mathtt{log}}$ invocation. Given this structure, typing judgments of the language have the form $\Theta;\Gamma \vdash P \triangleright T;U$ where T and U are the types of the current session and the type of the parent (the previously opened) session. Notice that we do not have any subject (i.e. k) since the session name is not relevant. Now we give the typing rules, for the choices

$$
\text{(TSELECT)}\quad \frac{\Theta;\Gamma \vdash P \triangleright T;U}{\Theta;\Gamma \vdash \langle l\rangle.P \triangleright \oplus\{l:T\};U}
$$

$$
\text{(TBRANCH)}\quad \frac{\Gamma;\Theta \vdash P_1 : T_1;U \ldots \Gamma;\Theta \vdash P_n : T_n;U}{\Gamma;\Theta \vdash \Sigma_{i=0}^{n}(l_i).P_i : \&\{l_i : T_i\}_{1\le i\le n}}
$$

which are completely deterministic and the rule for the if-then-else

$$
\text{(TIF)}\quad \frac{\Gamma \vdash e \triangleright bool \quad \Theta;\Gamma \vdash P : T_1;U \quad \Theta;\Gamma \vdash Q \triangleright T_2;U \quad T = T_1\overset{c}{\wedge}T_2}{\Theta;\Gamma \vdash \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q \triangleright T;U}
$$

that uses the intersection. The last ingredient is the duality relation which allows the branches in a internal choice to be a subset of the branches in the dual external choice.

Definition 10. *A relation $R \subseteq TYPE \times TYPE$ is a subtyping duality relation if $(T, V) \in R$ implies the following conditions:*

- *if* $\mathsf{unfold}(T) = \&\{l_1 : T_1, \ldots, l_m : T_m\}$ *then* $\mathsf{unfold}(V) = \oplus\{l'_1 : V_1, \ldots, l'_n : V_n\}$ *and $n \leq m$ and for all $i \in \{1, \ldots, n\}$ there exists one $j \in \{1, \ldots, m\}$ such that $l'_i = l_j$ and $(T_j, V_i) \in R$*
- *if* $\mathsf{unfold}(T) = \oplus\{l_1 : T_1, \ldots, l_m : T_m\}$ *then* $\mathsf{unfold}(V) = \&\{l'_1 : V_1, \ldots, l'_n : V_n\}$ *and $m \leq n$ and for all $i \in \{1, \ldots, m\}$ there exists one $j \in \{1, \ldots, n\}$ such that $l_i = l'_j$ and $(T_i, V_j) \in R$*
- *the remaining rules are the same to the Definition 7.*

The co-inductive $\perp_{sub}$ relation is defined as $T \perp_{sub} V$ if exists subtyping duality relation R s.t. $(T, V) \in R$.

Next theorem states that the meet can be used to fully infer the session behaviors. Thanks to the new duality relation, this time the type of S' after the substitution with either T or U remains the same.

Theorem 6. *If $S \perp_{sub} S'$ then for all T and U s.t. $T \overset{c}{\wedge} U = S$ we have $T \perp_{sub} S'$ and $S \perp_{sub} T$*

Proof. The proof consists in proving that $R_S = \{(V, T) | \exists U \ U \leq V$ and $U \perp_{sub} T\}$ and $R_{S'} = \{(T, V) | \exists U \ U \leq V$ and $T \perp_{sub} U\}$ are subtyping duality relations. We sketch the case for R_S where V is an external choice. If $\mathsf{unfold}(V) = \oplus\{l_1 : V_1, \ldots, l_n : V_n\}$ then by definition of $\leq$, $\mathsf{unfold}(U) = \oplus\{l'_1 : U_1, \ldots, l'_m : U_m\}$ with $n \leq m$ and for all $i \in \{1, \ldots, n\}$ exists $j \in \{1, \ldots, m\}$ and $l_i = l'_j$ and $U_j \leq V_i$. By definition of $\perp_{sub}$, $\mathsf{unfold}(T) = \&\{l''_1 : T_1, \ldots, l''_h : T_h\}$ with $m \leq h$, and for all $j \in \{1, \ldots, m\}$ exists one $z \in \{1, \ldots, h\}$ such that $l'_j = l''_z$ and $U_j \perp_{sub} T_z$. The result follows since $n \leq m \leq h$ and for all $j \in \{1, \ldots, n\} \subseteq \{1, \ldots, m\}$ exists one $z \in \{1, \ldots, h\}$ such that $l'_j = l''_z$ and $U_j \perp_{sub} T_z$.

Previous theorem together with rule (Tif) make it possible to have different protocols in each branch of the if-then-else.