Objects and Session Types

Dedicated to Giuseppe Longo at the occasion of his 60th birthday

Mariangiola Dezani-Ciancaglini [a] Sophia Drossopoulou [b]
Dimitris Mostrous [b] Nobuko Yoshida [b]

[a] *Dipartimento di Informatica, Università di Torino*

[b] *Department of Computing, Imperial College London*

Abstract

A session takes place between two parties; after establishing a connection, each party interleaves local computations and communications (sending or receiving) with the other. Session types characterise such sessions in terms of the types of values communicated and the shape of protocols, and have been developed for the π-calculus, CORBA interfaces, and functional languages. We study the incorporation of session types into object-oriented languages through MOOSE, a multi-threaded language with session types, thread spawning, iterative and higher-order sessions. Our design aims to consistently integrate the object-oriented programming style and sessions, and to be able to treat various case studies from the literature. We describe the design of MOOSE, its syntax, operational semantics and type system, and develop a type inference system. After proving subject reduction, we establish the progress property: once a communication has been established, well-typed programs will never starve at communication points.

1 Introduction

Object-based communication oriented software is commonly implemented using either sockets or remote method invocation, such as Java RMI and C# remoting. Sockets provide generally untyped stream abstractions, while remote method invocation offers the benefits of standard method invocation in a distributed setting. However, both have shortcomings: socket-based code requires a significant amount of dynamic checks and type-casts on the values exchanged, in order to ensure type safety; remote method invocation does ensure that methods are used as mandated by their type signatures, but does not allow programmers to express design patterns frequently arising in distributed applications, where *sequences* of messages of different types are exchanged through a single connection following fixed protocols. A natural question is the seamless integration of tractable descriptions of type-safe communication patterns with object-oriented programming idioms.

A *session* is such a sequence of interactions between two parties. It starts after a connection has been established. During the session, each party may execute its own local computation, interleaved with several communications with the other party. Communications take the form of sending and receiving values over a channel. Additionally, throughout interaction between the two parties, there should be a perfect matching of sending actions in one with receiving actions in the other, and vice versa. This form of structured interaction is found in many application scenarios.

Session types have been proposed in [32], aiming to characterise such sessions, in terms of the types of messages received or sent by a party. For example, the session type `begin.!int.!int.?bool.end` expresses that two `int`-values will be sent, then a `bool`-value will be expected to be received, and then the protocol will be complete. Thus, session types specify the communication behaviour of a piece of software, and can be used to verify the safety of communication protocols between two parties. Session types have been studied for several different settings, *i.e.*, π-calculus-based systems [6, 7, 11–13, 18, 27, 28, 32, 34, 35, 48], mobile processes [43], boxed ambients [26], CORBA [49], functional languages [29], and recently, for CDL, a W3C standard description language for web services [11, 13, 47, 52].

In this paper we study the incorporation of session types into object-oriented languages. To our knowledge, except for some of our earlier work [15, 19, 21, 23], such an integration has not been attempted so far. We propose the language Moose, a multi-threaded object-oriented core language augmented with session types, which supports thread spawning, iterative sessions, and higher-order sessions.

The design of Moose was guided by the wish for the following properties:

object oriented style We wanted Moose programming to be as natural as possible to people used to mainstream object oriented languages. In order to achieve an object oriented style, Moose allows sessions to be handled modularly using methods.

expressivity We wanted to be able to express common case studies from the literature on session types and concurrent programming idioms [42], as well as those from the WC3 standard documents [13, 52]. In order to achieve expressivity, we introduced conditional, and iterative sessions, the ability to spawn new threads, and to send and receive sessions (*i.e.*, higher-order sessions).

type preservation The guarantee that execution preserves types, *i.e.*, the subject reduction property, proved to be an intricate task. In fact, several session type systems in the literature fail to preserve typability after reduction of certain subtle configurations, which we identified through a detailed analysis of how types of communication channels evolve during reduction. Type preservation requires linear usage of live channels; in order to guarantee this we had to prevent aliasing of channels, manifested by the fact that running session types (*i.e.*, the types of live channels) cannot be assigned to fields. We claim this restriction is quite natural since channels are not objects. Note that aliasing is less problematic in a functional setting like that one considered in [51] than in an imperative setting like the one we are dealing with here.

progress We wanted to be able to guarantee that once a session has started, *i.e.*, a connection has been established, threads neither starve nor deadlock at the points of communication during the session. Progress is a highly desirable property in communication-based programs. Establishing this property was an intricate task as well, and, to the best of our knowledge, no other session type system in the literature, but those in [4, 10, 15, 18, 21, 23], can ensure it. The combination of higher-order sessions, spawn and the requirement to prevent deadlock during sessions posed the major challenge for our type system.

This work is an extended version of [20], with complete definitions, more explanations, detailed proofs and more comparisons with related work. Furthermore, we introduced minor differences in order to deal with small discrepancies which we discovered while developing the more detailed proofs.

The paper is organised as follows: §2 illustrates the basic ideas through an example. §3 defines the syntax of the language. §4 presents the operational semantics. §5 describes design decisions, such as the restriction on channel aliasing, that ensured type preservation and progress. §6 illustrates the typing system. §7 gives basic theorems on type safety and communication safety. §8 describes type inference. §9 discusses the related work, and §10 concludes. More examples of MOOSE can be found in [42]. The proofs are given in the appendices.

2 Business Protocol Example

We describe a typical collaboration pattern that appears in many web service business protocols [11–13, 35, 52] using MOOSE. This simple protocol contains essential features by which we can demonstrate the expressivity of MOOSE: it requires a combination of session establishing, higher-order session passing, spawn, conditional sessions, and deadlock-freedom during the session.

In Fig. 1 we show the sequence diagram for the protocol, which models the purchasing of items. We show the participants, the sessions between them, and the program variables whose value is communicated on each channel. First, the `Seller` and `Buyer` participants initiate interaction over channel `c1`; then, the `Buyer` sends a product id to the `Seller`, and receives a price quote in return; finally, the `Buyer` may either accept or reject this price. Thus, here we show the first case of a conditional session. If the price received is acceptable, then the `Seller` connects with the `Shipper` over channel `c2`. First the `Seller` sends to the `Shipper` the details of the purchased item. Then the `Seller` delegates its part of the remaining activity with the `Buyer` to the `Shipper`, that is realised by sending `c1` over `c2`. Now the `Shipper`

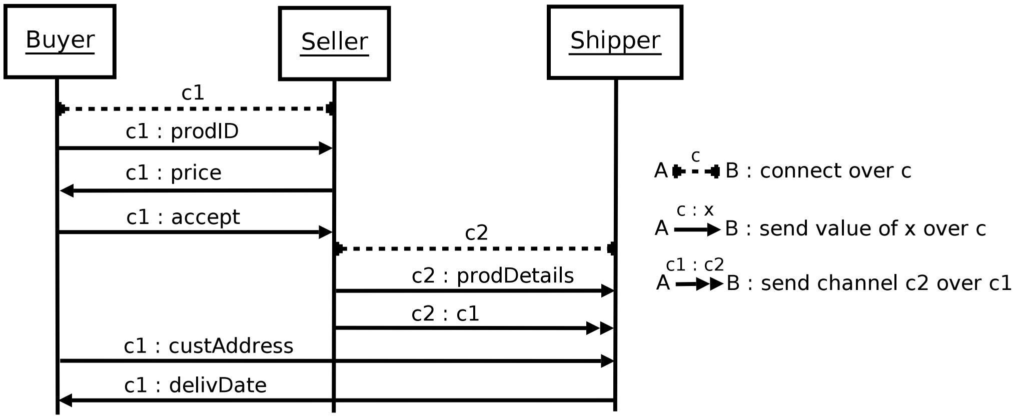

Fig. 1. Sequence diagram for item purchasing protocol.

```
session BuyProduct =
  begin.!String.?double.!<!Address.?DeliveryDetails.end, ε.end>
session RequestDelivery =
  begin.!ProductDetails.!(?Address.!DeliveryDetails.end).end
```

Fig. 2. Session types for the buyer-seller-shipper example

will await the `Buyer`'s address, before responding with the delivery date. If the price is not acceptable, then the interaction terminates.

In Fig. 2 we declare the necessary session types, and in Fig. 3 we encode the given scenario in MOOSE, using one class per protocol participant. The session types `BuyProduct` and `RequestDelivery` describe the communication patterns between `Buyer` and `Seller`, and `Seller` and `Shipper`, respectively. The session type `BuyProduct` models the sending of a `String`, then the reception of a `double`, and finally a conditional behaviour, in which a `bool` is (implicitly) sent before a branch is followed: the first branch requires that an `Address` is sent, then a `DeliveryDetails` received, and finally that the session is closed; the second branch models an empty communication sequence and the closing of the session. We write $\overline{\mathtt{BuyProduct}}$ for the *dual* type, which is constructed by taking `BuyProduct` and changing occurrences of ! to ? and vice versa; these types represent the two complementary behaviours associated with a session, in which the sending of a value in one end corresponds to its reception at the other. In other words, $\overline{\mathtt{BuyProduct}}$ is the same as `begin.?String.!double.?<?Address.!DeliveryDetails.end,end>`. Note that in the case of the conditional, the thread with ! in its type decides which branch is to be followed and communicates the boolean value, while the other thread passively awaits the first thread's decision. The session type `RequestDelivery`describes sending a `ProductDetails` instance, followed by sending a 'live' session channel of remaining type `?Address.!DeliveryDetails.end`.

Sessions can start when two compatible `connect` statements are active. In Fig. 3, the first component of `connect` is the shared channel that is used to start commu-

```
class Buyer {

   Address addr;

   void buy( String prodID, double maxPrice ) {
      connect c1 BuyProduct {
         c1.send( prodID );
         c1.sendIf( c1.receive <= maxPrice ) {
             c1.send( addr );
             DeliveryDetails delivDetails := c1.receive;
         }{ null; /* buyer rejects price, end protocol */ }
      } /* End connect */
   } /* End method buy */
}

class Seller {
   void sell() {
      connect c1 BuyProduct {
         String prodID := c1.receive;
         double price := getPrice( prodID ); // implem. omitted
         c1.send( price );
         c1.receiveIf { // buyer accepts price
            ProductDetails prodDetails := new ProductDetails();
            // ... init prodDetails with prodID, size, etc
            spawn { connect c2 RequestDelivery {
                    c2.send( prodDetails ); c2.sendS( c1 );} }
            }{ null; /* receiveIf : buyer rejects */ }
      } /* End connect */
   } /* End method sell */
}

class Shipper {
    void delivery() {
       connect c2 RequestDelivery {
            ProductDetails prodDetails := c2.receive;
            c2.receiveS( x ) {
            Address custAddress := x.receive;
            DeliveryDetails delivDetails := new DeliveryDetails();
            //... set state of delivDetails
            x.send( delivDetails ); }
       } /* End connect */
    } /* End method delivery */
}
```

Fig. 3. Code for the buyer, seller and shipper.

nication, the second is the session type, and the third is the *session body*, which implements the session type. The method `buy` of class `Buyer` contains a `connect` statement that implements the session type `BuyProduct`, while the method `sell` of class `Seller` contains a `connect` statement over the same channel and the dual session type. When a `Buyer` and a `Seller` are executing concurrently their respective methods, they can engage in a session, which will result in a fresh channel being replaced for occurrences of the shared channel `c1` within both session bodies; freshness guarantees that the new channel only occurs in these two threads, therefore the objects can proceed to perform their interactions without the possibility of external interference.

Once the session has started in the body of method `buy`, the product identifier, `prodID`, is sent using `c1.send(prodID)` and the price quote is received using `c1.receive`. If the price is acceptable, *i.e.*, if `c1.receive <= maxPrice`, then `true` is sent and the first branch of the conditional is taken, starting on line 9. In this case, the customer's address, `addr`, is sent and an instance of `DeliveryDetails` is received. If the price is not acceptable, then `false` is sent and the second branch of the conditional starting on line 11 is taken, and the connection closes.

The body of method `sell` implements behaviour dual to the above. Note that in `c1.receiveIf{...}{...}` the branch to be selected depends on the boolean value received from the other end, which will execute the complementary expression `c1.sendIf(..){...}{...}`. The first branch of the `Seller`'s conditional contains a nested `connect` in line 25, via which the product details are sent to the `Shipper`, followed by the actual runtime channel that was substituted for `c1` when the outer `connect` took place; the latter is sent through the construct `c2.sendS(c1)`, which realises *higher-order session communication*. Notice that the code in lines 25-26 is within a `spawn`, which reduces to a new thread with the enclosed expression as its body.

The method `delivery` of class `Shipper` should now be clear, with the exception of `c2.receiveS(x){..}` which is dual to `c2.sendS(c1)`. In the first expression, the received channel is bound to variable `x`.

The above example shows how MOOSE achieves deadlock-freedom: whenever we have `c.send(v)`, eventually an expression of the shape `c.receive` will appear in some other thread, unless the thread diverges, or a null pointer exception is thrown, or there is a nested connect instruction waiting for the dual connect instruction. Likewise for the other communication expressions. By these *progress* conditions, no session will remain incomplete; see Examples 5.4, 5.5, and 5.6. For the precise definition of progress see Theorem 7.10.

(type)	t	$::=$	$C \mid \mathsf{bool} \mid \mathsf{s} \mid (\mathsf{s},\bar{\mathsf{s}})$
(class)	*class*	$::=$	$\mathsf{class}\ C\ \mathsf{extends}\ C\ \{\ \widetilde{\mathsf{f\,t}}\ \ \tilde{M}\ \}$
(method)	M	$::=$	$\mathsf{t\,m}\ (\widetilde{\mathsf{t\,x}},\widetilde{\rho\,\mathsf{y}})\ \{\mathsf{e}\}$
(expression)	e	$::=$	$\mathsf{this} \mid \mathsf{x} \mid \mathsf{v} \mid \mathsf{e};\,\mathsf{e} \mid \mathsf{e.f}:=\mathsf{e} \mid \mathsf{e.f} \mid \mathsf{e.m}(\,\tilde{\mathsf{e}}\,) \mid \mathsf{new}\ C$
		$\mid$	$\mathsf{spawn}\{\ \mathsf{e}\ \} \mid \mathsf{new}\ (\mathsf{s},\bar{\mathsf{s}}) \mid \mathsf{connect\ u\ s}\{\mathsf{e}\}$
		$\mid$	$\mathsf{u.receive} \mid \mathsf{u.send}(\mathsf{e})$
		$\mid$	$\mathsf{u.receiveS}(\mathsf{x})\{\mathsf{e}\} \mid \mathsf{u.sendS}(\mathsf{u})$
		$\mid$	$\mathsf{u.receiveIf}\{\mathsf{e}\}\{\mathsf{e}\} \mid \mathsf{u.sendIf}(\mathsf{e})\{\mathsf{e}\}\{\mathsf{e}\}$
		$\mid$	$\mathsf{u.receiveWhile}\{\mathsf{e}\} \mid \mathsf{u.sendWhile}(\mathsf{e})\{\mathsf{e}\} \mid \mathsf{NullExc}$
(channel)	u	$::=$	$\mathsf{c} \mid \mathsf{x}$
(value)	v	$::=$	$\mathsf{c} \mid \mathsf{null} \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{o}$
(thread)	P	$::=$	$\mathsf{e} \mid P \vert P$

Fig. 4. Syntax, where syntax occurring only at runtime appears shaded.

3 A Concurrent Object Oriented Language with Sessions

In Fig. 4 we describe the syntax of MOOSE. We distinguish *user syntax*, *i.e.*, source level code, and *runtime syntax*, which includes null pointer exceptions, threads and heaps. The syntax is based on FJ [37] with the addition of imperative and communication primitives similar to those from [3, 6, 21, 32, 34, 51]. We designed MOOSE as a multi-threaded concurrent language for simplicity of presentation; note however that MOOSE can easily be extended to model distribution; see § 9.

Channels We distinguish *shared channels* and *live channels*. Shared channels have not yet been connected; they are used to decide if two threads can communicate, in which case they are replaced by fresh live channels. After a connection has been created the channel is live; data may be transmitted through such active channels only. The types of MOOSE enforce the condition that there are exactly two threads which contain occurrences of the same live channel: we call it *bilinearity condition*. This is proved in Lemma 7.8.

User syntax The metavariable t ranges over types for expressions, ρ ranges over running session types, C ranges over class names and s ranges over shared session types. Each shared session type s has one corresponding *dual*, denoted $\bar{\mathsf{s}}$, which is obtained by replacing each ! (output) by ? (input) and vice versa. We introduce the full syntax of types in § 6, Fig. 8. Class and method declarations are as expected.

The first nine productions for expressions, e, e', are standard for concurrent object oriented programming, and represent the receiver (this), a method parameter (x), a value (v), a sequence of expressions ($\mathsf{e};\mathsf{e}'$), field assignment ($\mathsf{e.f} = \mathsf{e}'$), field access (e.f), method call ($\mathsf{e.m}(\tilde{\mathsf{e}})$), object creation (new *C*), and spawning of a new thread (spawn $\{$ e $\}$). The remaining productions are related to session creation and communication: first, the channel constructor new $(\mathsf{s}, \overline{\mathsf{s}})$, which builds a fresh shared channel used to establish a private session; next, the *communication expressions*, *i.e.*, connect u s$\{$e$\}$ and all the remaining session expressions. The reason for declaring both s and $\overline{\mathsf{s}}$ in the channel constructor is that we want to stress that the fresh created channel can replace two variables of types s and $\overline{\mathsf{s}}$, respectively, in order to establish a private communication, see Example 4.1.

The values are channels, null, and the literals true and false. Thread creation is declared using spawn $\{$ e $\}$, in which the expression e is called the *thread body*.

The expression connect u s$\{$e$\}$ starts a session: the channel u appears within the term $\{$e$\}$ in session communications that agree with session type s. The remaining eight expressions, which realise the exchanges of data, are called *session expressions*, and start with "u ._"; we call u the *subject* of such expressions. In Fig. 4 and in the explanations below, session expressions are pairwise coupled: we say that expressions in the same pair and with the same subject are *dual* to each other; *e.g.*, c3.send(true) and c3.receive are dual expressions.

The first pair is for exchange of values (which can be shared channels): u .receive receives a value via u, while u .send (e) evaluates e and sends the result over u. The second pair expresses live channel exchange : in u .receiveS (x)$\{$e$\}$ the received channel will be bound to x within the expression e, in which x is used for communications. The expression u .sendS (u') sends the channel u' over u. The third pair is for *conditional* communication: u .receiveIf $\{$e$\}\{\mathsf{e}'\}$ receives a boolean value via channel u, and if it is true continues with e, otherwise with e'; the expression u .sendIf (e)$\{\mathsf{e}'\}\{\mathsf{e}''\}$ first evaluates the boolean expression e, then sends the result via channel u and if the result was true continues with e', otherwise with e''. The fourth is for *iterative* communication: the expression u .receiveWhile $\{$e$\}$ receives a boolean value via channel u, and if it is true continues with e and iterates, otherwise ends; the expression u .sendWhile (e)$\{\mathsf{e}'\}$ first evaluates the boolean expression e, then sends its result via channel u and if the result was true continues with e' and iterates, otherwise ends.

We do not define the standard iteration and conditional statements, as these can be straightforwardly encoded in our calculus. For example while(e)$\{\mathsf{e}'\}$ can be simulated by c .sendWhile (e)$\{\mathsf{e}'\}$, assuming a session over c and the expression c .receiveWhile $\{$null $\}$ in another thread. Similarly for the conditional using c .sendIf (e)$\{\mathsf{e}'\}\{\mathsf{e}''\}$.

Also the general branch/select constructors are easily encoded in MOOSE and so we left them out to avoid syntactic sugar.

Finally, we do not include primitives for recursive sessions. This allows a simpler presentation, and more importantly, it enables us to formulate the *progress* property of our calculus based on a non-interleaving restriction (see § 5); with recursion, simply nested sessions, which we allow, would clearly result in interleaved traces after unfolding in the inner scope.

Runtime syntax The runtime syntax (shown shaded in Fig. 4) extends the user syntax: it extends values to allow for object identifiers o, which denote references to instances of classes; adds $\mathsf{NullExc}$ to expressions, denoting the null pointer error; finally, introduces threads running in parallel. Single and multiple *threads* are ranged over by P, P'. The expression $P \mid P'$ says that P and P' are running in parallel.

4 Operational Semantics

This section presents the operational semantics of MOOSE, which is inspired by the standard small step call-by-value reduction of Featherweight Java [46], extended with imperative features, as *e.g.*, in [45], and following the style of [3] and mainly that of [21]. We only discuss the more interesting rules. First we list the evaluation contexts.

$$\begin{aligned} E ::=\ & [\,] \mid E.\mathsf{f} \mid E;\mathsf{e} \mid E.\mathsf{f} := \mathsf{e} \mid \mathsf{o}.\mathsf{f} := E \mid E.\mathsf{m}(\tilde{\mathsf{e}}) \mid \mathsf{o}.\mathsf{m}(\tilde{\mathsf{v}}, E, \tilde{\mathsf{e}}) \\ \mid\ & \mathsf{c}.\mathsf{send}(E) \mid \mathsf{c}.\mathsf{sendIf}(E)\{\mathsf{e}\}\{\mathsf{e}'\} \end{aligned}$$

Notice that $\mathsf{connect}\ \mathsf{c}\ \mathsf{s}\,\{E\}$, $\mathsf{c}.\mathsf{receiveS}(\mathsf{x})\{E\}$, $\mathsf{c}.\mathsf{sendIf}(\mathsf{e})\{E\}\{\mathsf{e}\}$, $\mathsf{c}.\mathsf{sendIf}(\mathsf{e})\{\mathsf{e}\}\{E\}$, $\mathsf{c}.\mathsf{receiveIf}\{E\}\{\mathsf{e}\}$, $\mathsf{c}.\mathsf{receiveIf}\{\mathsf{e}\}\{E\}$, $\mathsf{c}.\mathsf{receiveWhile}\{E\}$, and $\mathsf{c}.\mathsf{sendWhile}(\mathsf{e})\{E\}$ are not evaluation contexts: the first would allow session bodies to run before the start of the session; the second would allow execution of an expression waiting for a live channel before actually receiving it; the remaining would allow parts of a conditional or iterative session to run before determining which branch should be selected, or whether the iteration should continue.

Fig. 5 defines auxiliary functions used in the operational semantics and typing rules. As in [37] we assume a fixed, global class table. The class *Object* does not have fields/methods and his declaration does not occur in the class table. The decoration $\circledcirc \in \{\ominus, \oplus\}$ in the function mtype will be motivated in Example 5.5.

Objects and channels are stored in *heaps*, whose syntax is given by:

Field lookup

$$\mathsf{fields}(\mathit{Object}) = \bullet \qquad \frac{\mathsf{fields}(D) = \widetilde{\mathsf{f}'\mathsf{t}'} \qquad \text{class } C \text{ extends } D\ \{\widetilde{\mathsf{f}\,\mathsf{t}}\ \tilde{M}\} \in \mathtt{CT}}{\mathsf{fields}(C) = \widetilde{\mathsf{f}'\mathsf{t}'}, \widetilde{\mathsf{f}\,\mathsf{t}}}$$

Method lookup

$$\mathsf{methods}(\mathit{Object}) = \bullet \qquad \frac{\mathsf{methods}(D) = \tilde{M}' \qquad \text{class } C \text{ extends } D\ \{\widetilde{\mathsf{f}\,\mathsf{t}}\ \tilde{M}\} \in \mathtt{CT}}{\mathsf{methods}(C) = \tilde{M}', \tilde{M}}$$

Method type lookup

$$\frac{\text{class } C \text{ extends } D\ \{\widetilde{\mathsf{f}\,\mathsf{t}}\ \tilde{M}\} \in \mathtt{CT} \quad \mathsf{t}\,\mathsf{m}\ (\widetilde{\tau\mathsf{x}})\ \{\mathsf{e}\} \in \tilde{M}}{\mathsf{mtype}(\mathsf{m}, C) = \tilde{\tau} \overset{\circledcirc}{\rightarrow} \mathsf{t}}$$

$$\frac{\text{class } C \text{ extends } D\ \{\widetilde{\mathsf{f}\,\mathsf{t}}\ \tilde{M}\} \in \mathtt{CT} \quad \mathsf{m} \notin \tilde{M}}{\mathsf{mtype}(\mathsf{m}, C) = \mathsf{mtype}(\mathsf{m}, D)}$$

Method body lookup

$$\frac{\text{class } C \text{ extends } D\ \{\widetilde{\mathsf{f}\,\mathsf{t}}\ \tilde{M}\} \in \mathtt{CT} \qquad \mathsf{t}\,\mathsf{m}\ (\widetilde{\tau\mathsf{x}})\ \{\mathsf{e}\} \in \tilde{M}}{\mathsf{mbody}(\mathsf{m}, C) = (\tilde{\mathsf{x}}, \mathsf{e})}$$

$$\frac{\text{class } C \text{ extends } D\ \{\widetilde{\mathsf{f}\,\mathsf{t}}\ \tilde{M}\} \in \mathtt{CT} \qquad \mathsf{m} \notin \tilde{M}}{\mathsf{mbody}(\mathsf{m}, C) = \mathsf{mbody}(\mathtt{m}, D)}$$

τ is either t or ρ.

Fig. 5. Lookup Functions

$$h \quad ::= \quad [\,] \mid h :: [\mathsf{o} \mapsto (C, \widetilde{\mathsf{f}:\mathsf{v}})] \mid h :: \mathsf{c}.$$

Heaps, ranged over h, are built inductively using the heap composition operator ‘::’, and contain mappings of object identifiers to instances of classes, and channels. In particular, a heap will contain the set of *fresh* objects and channels, both shared and live, that have been created since the beginning of execution, and the shared channels appearing free in the initial user program. The heap produced by composing $h :: [\mathsf{o} \mapsto (C, \widetilde{\mathsf{f}:\mathsf{v}})]$ will map o to the object $(C, \widetilde{\mathsf{f}:\mathsf{v}})$, where C is the class name and $\widetilde{\mathsf{f}:\mathsf{v}}$ is a representation for the vector of distinct mappings from field names to their values for this instance. The heap produced by composing $h :: \mathsf{c}$ will contain the fresh channel c. Heap membership for object identifiers and channels is checked using standard set notation, we therefore write it as $\mathsf{o} \in h$ and $\mathsf{c} \in h$, respectively. Heap update for objects is written $h[\mathsf{o} \mapsto (C, \widetilde{\mathsf{f}:\mathsf{v}})]$, and field update is written $(C, \widetilde{\mathsf{f}:\mathsf{v}})[\mathsf{f} \mapsto \mathsf{v}]$. Heap composition is undefined if the added object’s identifier (or the channel) is already in the heap; heap update is undefined if the updated

Fld

$$\frac{h(\mathsf{o}) = (C, \widetilde{\mathsf{f} : \mathsf{v}})}{\mathsf{o}.\mathsf{f}_\mathsf{i}, h \longrightarrow \mathsf{v}_\mathsf{i}, h}$$

Seq

$$\mathsf{v}; \mathsf{e}, h \longrightarrow \mathsf{e}, h$$

FldAss

$$\frac{h' = h[\mathsf{o} \mapsto h(\mathsf{o})[\mathsf{f} \mapsto \mathsf{v}]]}{\mathsf{o}.\mathsf{f} := \mathsf{v}, h \longrightarrow \mathsf{v}, h'}$$

NewC

$$\frac{\mathsf{fields}(C) = \widetilde{\mathsf{f}\ \mathsf{t}} \quad \mathsf{o} \notin h}{\mathsf{new}\ C, h \longrightarrow \mathsf{o}, h :: [\mathsf{o} \mapsto (C, \widetilde{\mathsf{f} : \mathsf{init}(\mathsf{t})})]}$$

NewS

$$\frac{\mathsf{c} \notin h}{\mathsf{new}\ (\mathsf{s}, \bar{\mathsf{s}}), h \longrightarrow \mathsf{c}, h :: \mathsf{c}}$$

Cong

$$\frac{\mathsf{e}, h \longrightarrow \mathsf{e}', h'}{E[\mathsf{e}], h \longrightarrow E[\mathsf{e}'], h'}$$

Meth

$$\frac{h(\mathsf{o}) = (C, \ldots) \quad \mathsf{mbody}(\mathsf{m}, C) = (\tilde{\mathsf{x}}, \mathsf{e})}{\mathsf{o}.\mathsf{m}(\tilde{\mathsf{v}}), h \longrightarrow \mathsf{e}[{}^{\mathsf{o}}/\mathtt{this}][{}^{\tilde{\mathsf{v}}}/\tilde{\mathsf{x}}], h}$$

NullProp

$$\frac{}{E[\mathsf{NullExc}], h \longrightarrow \mathsf{NullExc}, h}$$

NullFldAss

$$\mathsf{null}.\mathsf{f} := \mathsf{v}, h \longrightarrow \mathsf{NullExc}, h$$

NullFld

$$\mathsf{null}.\mathsf{f}, h \longrightarrow \mathsf{NullExc}, h$$

NullMeth

$$\mathsf{null}.\mathsf{m}(\tilde{\mathsf{v}}), h \longrightarrow \mathsf{NullExc}, h$$

In **NewC**, $\mathsf{init}(\mathsf{bool}) = \mathsf{false}$ otherwise $\mathsf{init}(\mathsf{t}) = \mathsf{null}$.

Fig. 6. Expression Reduction

object's identifier is not in the heap.

An object identifier o (channel c) is said to be *fresh* in heap h when $\mathsf{o} \notin h$ ($\mathsf{c} \notin h$). This condition, formalised in Lemma 7.5, guarantees that newly created objects and channels are not already used anywhere in a well-typed configuration.

Expressions Fig. 6 shows the rules for execution of expressions which correspond to the sequential part of the language. These are standard [5, 22, 37], except for the addition of a fresh shared channel to the heap (rule **NewS**). In rule **NewC** the auxiliary function $\mathsf{fields}(C)$ examines the class table and returns the field declarations for C. The method invocation rule is **Meth**; the auxiliary function $\mathsf{mbody}(\mathsf{m}, C)$ looks up m in the class C, and returns a pair consisting of the formal parameter names and the method's code. The result is the method body where the keyword `this` is replaced by the receiver's object identifier o, and the formal parameters $\tilde{\mathsf{x}}$ are replaced by the actual parameters $\tilde{\mathsf{v}}$. Note that the replacement of `this` by o cannot lead to unwanted behaviours since the receiver cannot change during execution of the method body.

Threads The reduction rules for threads are shown in Fig. 7. Rule **Struct** gives standard structural equivalence rules of the π-calculus [41], written $\equiv$. This equivalence is used in rule **Str**. We define *multi-step* reduction as: $\longrightarrow\!\!\!\!\rightarrow \overset{\mathsf{def}}{=} (\longrightarrow \cup \equiv)^*$.

Struct

$$P \,|\, \mathsf{null} \equiv P \quad P \,|\, P_1 \equiv P_1 \,|\, P \quad P \,|\, (P_1 \,|\, P_2) \equiv (P \,|\, P_1) \,|\, P_2 \quad P \equiv P' \Rightarrow P \,|\, P_1 \equiv P' \,|\, P_1$$

Str

$$\frac{P_1' \equiv P_1 \;\; P_1, h \longrightarrow P_2, h' \;\; P_2 \equiv P_2'}{P_1', h \longrightarrow P_2', h'}$$

Spawn

$$E[\mathsf{spawn}\{\ \mathsf{e}\ \}], h \longrightarrow E[\mathsf{null}] \,|\, \mathsf{e}, h$$

Par

$$\frac{P, h \longrightarrow P', h'}{P \,|\, P_0, h \longrightarrow P' \,|\, P_0, h'}$$

Connect

$$E_1[\mathsf{connect\ c\ s}\,\{\mathsf{e}_1\}] \,|\, E_2[\mathsf{connect\ c}\ \overline{\mathsf{s}}\{\mathsf{e}_2\}], h \;\longrightarrow\; E_1[\mathsf{e}_1[{}^{\mathsf{c}'}\!/\!_{\mathsf{c}}]] \,|\, E_2[\mathsf{e}_2[{}^{\mathsf{c}'}\!/\!_{\mathsf{c}}]], h :: \mathsf{c}'$$
$$\mathsf{c}' \notin h$$

ComS

$$E_1[\mathsf{c.send}\,(\mathsf{v})] \,|\, E_2[\mathsf{c.receive}], h \;\longrightarrow\; E_1[\mathsf{null}] \,|\, E_2[\mathsf{v}], h$$

ComSS

$$E_1[\mathsf{c.sendS}\,(\mathsf{c}')] \;|\; E_2[\mathsf{c.receiveS}\,(\mathsf{x})\{\mathsf{e}\}], h \;\longrightarrow\; E_1[\mathsf{null}] \;|\; \mathsf{e}\,[{}^{\mathsf{c}'}\!/\!_{\mathsf{x}}] \;|\; E_2[\mathsf{null}], h$$

ComSIf-true

$$E_1[\mathsf{c.sendIf}\,(\mathsf{true})\{\mathsf{e}_1\}\{\mathsf{e}_2\}] \,|\, E_2[\mathsf{c.receiveIf}\,\{\mathsf{e}_3\}\{\mathsf{e}_4\}], h \;\longrightarrow\; E_1[\mathsf{e}_1] \,|\, E_2[\mathsf{e}_3], h$$

ComSIf-false

$$E_1[\mathsf{c.sendIf}\,(\mathsf{false})\{\mathsf{e}_1\}\{\mathsf{e}_2\}] \,|\, E_2[\mathsf{c.receiveIf}\,\{\mathsf{e}_3\}\{\mathsf{e}_4\}], h \;\longrightarrow\; E_1[\mathsf{e}_2] \,|\, E_2[\mathsf{e}_4], h$$

ComSWhile

$$\begin{aligned} & E_1[\mathsf{c.sendWhile}\,(\mathsf{e})\{\mathsf{e}_1\}] \,|\, E_2[\mathsf{c.receiveWhile}\,\{\mathsf{e}_2\}], h \longrightarrow \\ & \qquad E_1[\mathsf{c.sendIf}\,(\mathsf{e})\{\mathsf{e}_1; \mathsf{c.sendWhile}\,(\mathsf{e})\{\mathsf{e}_1\}\}\{\mathsf{null}\}] \\ & \qquad |\; E_2[\mathsf{c.receiveIf}\,\{\mathsf{e}_2; \mathsf{c.receiveWhile}\,\{\mathsf{e}_2\}\}\{\mathsf{null}\}], \;\; h \end{aligned}$$

Fig. 7. Thread Reduction

In rule **Spawn**, when spawn { e } is the active redex within an arbitrary evaluation context, the *thread body* e becomes a new thread, and the original spawn expression is replaced by null in the context.

Rule **Connect** describes the opening of sessions: if two threads require a session on the same channel name c with dual session types, then a new fresh channel c′ is created and added to the heap. The freshness of c′ guarantees privacy and bilinearity of the session communication between the two threads. Finally, the two connect expressions are replaced by their respective session bodies, where the shared channel c has been substituted by the live channel c′. Note that all channels which occur in a well-typed thread occur also in any well-formed heap which agrees with the thread, see Lemma 7.5.

Rule **ComS** gives simple session communication: value v is sent by one thread and received by another. Rule **ComSS** formalises the act of delegating a session. One thread awaits to receive a live channel, which will be bound to the variable x within the expression e, and another thread is ready to send such a channel. Notice that when the channel is exchanged, the receiver spawns a new thread to handle the consumption of the delegated session. This strategy is necessary in order to avoid deadlocks in the presence of circular paths of session delegation; see Example 4.4.

In rules **ComSIf-true** and **ComSIf-false**, depending on the value of the boolean, execution proceeds with either the first or the second branch. Rule **CommSWhile** simply expresses the iteration by means of the conditional. This operation allows to repeat a sequence of actions within a single session, which is convenient when describing practical communication protocols (see [11, 13, 21]).

The following examples justify some aspects of our operational semantics.

Example 4.1 motivates the inclusion of new channel creation in the language.

We extend the example of Fig. 3 with this extra functionality: the Buyer should receive notification when – after the session finishes – the goods are dispatched from the warehouse.

This requires a call-back session, a reversal of roles in which the service decides when to establish a connection with the waiting client. Because the call-back continues a previous session, it should be established over a shared channel agreed by, and unique to, the original participants. This, in turn, requires the ability to generate fresh shared channels, which can be distributed before the end of the initial session.

The Shipper can be modified, with the following code inserted at line 40, extending the original protocol:

```
40    x.send( delivDetails ); // }
41    // Create call-back channel with
42    // s = begin.!DeliveryDate.end
43    (s,s̄) y := new (s,s̄);
44    // send y to "Warehouse" over c3, uses s
45    connect c3 begin.!DeliveryDetails.!s.end {
46       c3.send( delivDetails ); c3.send( y ); }
47    x.send( y ); } // send y to "Buyer", uses s̄
48    ...
```

In the above, a new fresh channel is created at line 43, with a session type allowing the exchange of a `DeliveryDate` object. This channel is then distributed to the `Warehouse` (code not shown), at lines 45 and 46, and `Buyer`, at line 47. Now, the `Buyer` can wait for the `Warehouse` to connect, at some point, and provide the exact delivery date, over the channel shared uniquely by the two.

Example 4.2 demonstrates how server objects can be modelled using sessions and thread creation via `spawn`.

Again, we extend the example of Fig. 3, enabling a `Seller` object to serve multiple `Buyer` requests concurrently. This is shown below, where `e` represents the original session body of Fig. 3, lines 19 to 27:

```
16 class Seller {
17    void sell() {
18       while( true ) {
19          connect c1 B̅u̅y̅P̅r̅o̅d̅u̅c̅t̅ {
20             spawn{ e }; // Thread with original
21                         // session body
22          } /* End connect */
23       } /* End while */
24    } /* End method sell */
25 }
```

In the above, we first placed the body of method `sell` inside a non-terminating loop, allowing clients to be served in sequence. However, after a connection with a `Buyer` has been established, we do not want other buyers to have to wait until the previous session is complete. Instead, we allocate a new thread for each connection, by placing the original session body `e` within a `spawn` at line 20. Using this code, after a connection is made, a new thread is dispatched to execute the session body, and the `Seller`’s code can iterate and connect with another client immediately.

Example 4.3 demonstrates the use of iterative sessions.

As before, we extend the example of Fig. 3, this time to allow a Buyer to order more than one product per invocation of method buy. The code replacing the original from line 5 onwards is as follows:

```
5  void buy( String[] prodID, double[] maxPrice ) {
6     connect c1 BuyProduct{
7        int i := 0;
8        c1.sendWhile( i++ < prodID.length ) {
9           c1.send( prodID[i] );
10          // was price accepted?
11          c1.send( c1.receive <= maxPrice[i] ); }
12       // Now send address, and get delivery
13       // details, as done originally ...
14    } /* End connect */
15 } /* End method buy */
```

First, the signature of buy is changed to expect array arguments – we use arrays and other language features which are not defined in our language, but which are orthogonal to sessions. Second, at lines 8 to 11 we implement an iterative session part: at line 8, we require that the session part iterates as long as there are products in the array given as argument; at line 9, we send the current product identifier, based on the index i; then, at line 11, we receive a price quote, compare it to the maximum acceptable price for the item, and send the boolean result back to the Seller, so that the item can be added to, or ignored from, the order. When the iterations are finished, after line 11, the protocol would continue along the same lines as the original – but we omit this code.

Example 4.4 demonstrates the reason for the definition of rule **ComSS** which creates a new thread out of the expression in which the sent channel replaces the channel variable. A more natural and simpler formulation of this rule would avoid spawning a new thread:

$$E_1[\mathsf{c}.\mathsf{receiveS}\,(\mathsf{x})\{\mathsf{e}\}] \mid E_2[\mathsf{c}.\mathsf{sendS}\,(\mathsf{c}')], h \longrightarrow E_1[\mathsf{e}\,[{}^{\mathsf{c}'}\!/\mathsf{x}]] \mid E_2[\mathsf{null}\,], h$$

However, using the above version of the rule, and assuming session types s1 and s2, defined as s1=begin.?int.end, and s2=begin.?(!int.end).end, parallel execution of the threads P_1 and P_2 shown below reduces to

$$\mathsf{c}'_1.\mathsf{send}\,(5)\,;\,\mathsf{c}'_1.\mathsf{receive} \mid \mathsf{null}, \quad h::\mathsf{c}'_1$$

where c'_1 is the fresh live channel that replaced c_1 when the connection was established. Notice that both ends of the session are in one thread, so the last configuration is stuck.

```
connect c1 s1 {
 connect c2 s2 {
  c2.receiveS(x) { x.send(5)} };
  c1.receive
}
```

P_1

```
connect c1 s̅1̅ {
 connect s̅2̅ {
  c2.sendS(c1)
 }
}
```

P_2

5 Motivating the Design of the Type System

This section discusses the key ideas behind the type system introduced in § 6 with some examples, focusing on type preservation and progress.

Type preservation In order to achieve subject reduction, we need to ensure that at any time during execution, no more than two threads have access to the same live channel, and also, that no thread has aliases (*i.e.*, more than one reference) to a live channel.

Example 5.1 demonstrates that bilinearity is required for type preservation, and that in order to guarantee bilinearity we need to restrict aliases on live channels. Assume in the following, that we allowed live channels to be stored in fields, and that in the threads P_1, P_2 and P_3 the field accesses $\mathsf{o}_1.\mathsf{f}_1$, $\mathsf{o}_2.\mathsf{f}_2$, and $\mathsf{o}_3.\mathsf{f}_3$ all point to the same live channel c in heap h.

$$\overbrace{\mathsf{o}_1.\mathsf{f}_1.\mathsf{send}\,(3);\mathsf{o}_1.\mathsf{f}_1.\mathsf{send}\,(\mathsf{true})}^{P_1} \mid \overbrace{\mathsf{o}_2.\mathsf{f}_2.\mathsf{send}\,(4);\mathsf{o}_2.\mathsf{f}_2.\mathsf{send}\,(\mathsf{false})}^{P_2}$$
$$\mid \underbrace{\mathsf{o}_3.\mathsf{f}_3.\mathsf{receive};\mathsf{o}_3.\mathsf{f}_3.\mathsf{receive}}_{P_3}, \quad h$$

It is clear that P_3 expects to receive first an integer and then a boolean via channel c; but P_3 could communicate first with P_1 and then with P_2 (or vice versa) receiving two integers, destroying the intended sequence of communication between the two original partners of the session. To avoid the creation of aliases on live channels, we do not allow live channel types to be used as the types of fields, nor do we allow more than one live channel parameter in methods.

Example 5.2 demonstrates that guaranteeing bilinearity requires restrictions on sending/receiving live channels. In the following, assuming that the three threads, P_1, P_2 and P_3 could be typed, for some s_1 and s_2,

```
connect c1 s1 {
   connect c2 s2 {
      c2.sendS(c1) };
   c1.receive }
```

P_1

```
connect c1 s̅1 {
   c1.receive;
   c1.send(3)
}
```

P_2

```
connect c2 s̅2 {
   c2.receiveS(x){ x.send(4) }
}
```

P_3

then, starting with a heap h, the above three threads in parallel reduce to:

$$\mathsf{c}'_1.\mathsf{receive} \mid \mathsf{c}'_1.\mathsf{receive}\,;\, \mathsf{c}'_1.\mathsf{send}\,(3) \mid \mathsf{c}'_1.\mathsf{send}\,(4), \quad h::\mathsf{c}'_1::\mathsf{c}'_2$$

where c'_1 and c'_2 are the fresh live channels that replaced respectively $\mathtt{c}_1$ and $\mathtt{c}_2$ when the sessions began. Clearly, this configuration violates the bilinearity condition.

We therefore need a notion of whether a live channel has been *consumed*, *i.e.*, whether it cannot be further used for the communication of values. There is no explicit user syntax for consuming channels. Instead, channels are implicitly consumed 1) at the end of a connection, 2) when they are sent over a channel, and 3) when they are used within spawn. However, types do distinguish consumed channels using the end suffix; this condition originates from [34, 51]. This allows us to know if a live channel passed as parameter in a method call will be consumed or not by the execution of the method body. In § 6.1 we show that P_1 is type incorrect for any $\mathtt{s}_1$ and $\mathtt{s}_2$.

Progress in Moose means that indefinite waiting may only happen at the point where a connection is required, and in particular when the dual of a connect is missing. In other words, there will never be a deadlock at the communication points on live channels. This can only be guaranteed if the communications are always processed in a given order, *i.e.*, if there is no interleaving of sessions.

Example 5.3 shows how a well-behaved program can be rejected by our type system, to ensure general progress.

```
connect c1 begin.!int.end {
   connect c2 begin.?int.end
       {
      c1.send(3); c2.receive}
}
```

P_1

```
connect c1 begin.?int.end {
   connect c2 begin.!int.end
       {
      c1.receive; c2.send(5)}
}
```

P_2

In the above the interleaved communications on channels c_1 and c_2 would reach completion; however, as the next example shows, a small modification in the order of communications – which is undetected at the type level – can result in a deadlocked state.

Example 5.4 demonstrates how session interleaving may cause deadlocks.

```
connect c1 begin.!int.end {
   connect c2 begin.?int.end
       {
      c1.send(3); c2.receive}
}
```

P_1

```
connect c1 begin.?int.end {
   connect c2 begin.!int.end
       {
      c2.send(5); c1.receive}
}
```

P_2

In the above example we have indefinite waiting after establishing the connection, because P_1 cannot progress unless P_2 reaches the statement $\mathsf{c}_1.\mathsf{receive}$, and P_2 cannot progress unless P_1 reaches the statement $\mathsf{c}_2.\mathsf{receive}$, and so we have a deadlock at a communication point. A similar deadlock between live channels has been investigated in the context of linear and behavioural types of mobile processes, *e.g.*, [38, 53]. Note that *nesting* of sessions does not affect progress. Let us consider the following processes:

$$P_1' = \mathsf{connect}\ \mathsf{c}_1\ \mathsf{begin.?int.end}\{\mathsf{c}_1.\mathsf{receive};\mathsf{connect}\ \mathsf{c}_2\ \mathsf{begin.!int.end}\{\mathsf{c}_2.\mathsf{send}\,(5)\}\}$$

$$P_2' = \mathsf{connect}\ \mathsf{c}_1\ \mathsf{begin.!int.end}\{\mathsf{c}_1.\mathsf{send}\,(3);\mathsf{connect}\ \mathsf{c}_2\ \mathsf{begin.?int.end}\{\mathsf{c}_2.\mathsf{receive}\}\}$$

$$P_3' = \mathsf{connect}\ \mathsf{c}_1\ \mathsf{begin.!int.end}\{\mathsf{connect}\ \mathsf{c}_2\ \mathsf{begin.?int.end}\{\mathsf{c}_2.\mathsf{receive}\};\mathsf{c}_1.\mathsf{send}\,(3)\}$$

Parallel execution of P_1' and P_2' does not cause deadlock, while parallel execution of P_1' with P_3' does, but it does so at the connection point for c_2. However, such deadlock is acceptable, since it would disappear if we placed a suitable $\mathsf{connect}$ in parallel.

In order to avoid interleaving at live channels, we require that within each "scope" no more than one live channel can be used for communication; we call this the "hot set." The formal definition can be found in § 6. The hot set offers a simpler typing system than those based on behavioural types [38, 53] which need to keep track of dependencies between channels.

In § 6.1, we will show that P_1 and P_2 are type incorrect.

The following, similar, example justifies the requirement that also spawned processes use the current hot channel for communication.

```
connect c1 begin.!int.end {
   connect c2 begin.?int.end{
     spawn { c1.send(3);
           spawn { c2.receive }
     }
   }
}
```

P_3

```
connect c1 begin.?int.end {
   connect c2 begin.!int.end{
      spawn { c2.send(5);
         spawn { c1.receive }
    }
  }
}
```

P_4

Namely, execution of $P_3 \mid P_4$ starting with a heap h leads to

$$\mathsf{null} \mid \mathsf{c}'_1.\mathsf{send}\,(3);\mathsf{spawn}\,\{\,\mathsf{c}'_2.\mathsf{receive}\,\} \mid \mathsf{null} \mid \mathsf{c}'_2.\mathsf{send}\,(5);\mathsf{spawn}\,\{\,\mathsf{c}'_1.\mathsf{receive}\,\}, \quad h{::}\mathsf{c}'_1{::}\mathsf{c}'_2$$

which is deadlocked. As we will see, the type system makes P_3 type incorrect.

Example 5.5 demonstrates that in order to avoid deadlocks, we also need to take into account the live channels used to send and receive inside the method bodies. Consider a method `m` of class C with a parameter `x` of type `?int.end` and body `x.receive`. In this case, the two threads P_1 and P_2 below in parallel, starting with a heap h, reduce to

$$\mathsf{c}'_2.\mathsf{send}\,(3);\mathsf{c}'_1.\mathsf{send}\,(5) \mid \mathsf{c}'_1.\mathsf{receive};\mathsf{c}'_2.\mathsf{receive}\,, \quad h{::}\mathsf{c}'_1{::}\mathsf{c}'_2$$

```
connect c1 begin.!int.end {
    connect c2 begin.!int.end {
                   c2.send(3)
                   };
       c1.send(5)
 }
```

P_1

```
connect c1 begin.?int.end {
    connect c2 begin.?int.end {
       new C.m(c1);
              c2.receive
      }
}
```

P_2

In order to avoid problems like the above, we require that the only channel used for sending and receiving in the method body to be the first channel parameter, if any, and we decorate the method type with the superscript $\oplus$ to indicate that the method body may send or receive on the first channel parameter; and $\ominus$ to indicate that the method body does not send or receive on any of the channel parameters.

Example 5.6 demonstrates that allowing live channels in the body of a channel receive expression may destroy progress. We assume session types s_1=`begin.!int.end` and s_2=`begin.!(!int.end).end`.

```
connect c1 s1 {
    connect c2 s2 {
        c2.sendS(c1) };
}
```

P_1

```
connect c1 s̄1 {
    connect c2 s̄2 {
      c2.receiveS(x){
        x.send(3);
        connect c3 s1 {
            c3.sendS(c1)
      };
    };
}
```

P_2

Starting with a heap h, the two threads above reduce to a deadlock at a communication point

$$\mathsf{c}'_1.\mathsf{send}\,(3); \mathsf{connect}\ \mathsf{c}_3\,\mathsf{s}\,\{\mathsf{c}_3.\mathsf{sendS}\,(\mathsf{c}'_1)\}, \quad h :: \mathsf{c}'_1 :: \mathsf{c}'_2$$

Discussion In this section we showed how the aim to guarantee progress drove the design of the type system, and how this aim imposed some conditions on the use of live channels.

We believe that these conditions are not that restrictive. First, we can represent most of the communication protocols in the session types literature, as well as traditional synchronisation [42, § 3], while at the same time ensuring progress. Secondly, since these conditions are only essential for progress, if we remove hot sets from typing judgements, and we allow multiple live channel parameters in methods, we will obtain a more relaxed type system which allows deadlock on live channels, but still preserves type safety.

6 Type System

Types The full syntax of types is given in Fig. 8.

Partial session types, ranged over by π, represent sequences of communications, where ε is the empty communication, and $\pi_1.\pi_2$ consists of the communications in π_1 followed by those in π_2. We use $\dagger$ as a convenient abbreviation that ranges over $\{!, ?\}$. The partial session types $!\mathtt{t}$ and $?\mathtt{t}$ express respectively the sending and reception of a value of type $\mathtt{t}$.

The *conditional* partial session type has the shape $\dagger\langle\pi_1, \pi_2\rangle$. When $\dagger$ is $!$, $\dagger\langle\pi_1, \pi_2\rangle$ describes sessions which send a boolean value and proceed with π_1 if the value is true, or π_2 if the value is false; when $\dagger$ is $?$, the behaviour is the same, except

$\dagger$	$::=$	$! \mid ?$	direction
π	$::=$	$\varepsilon \mid \pi.\pi \mid \dagger \mathsf{t} \mid \dagger\langle\pi,\pi\rangle \mid \dagger\langle\pi\rangle^{*} \mid \dagger(\eta)$	partial session type
η	$::=$	$\pi.\mathsf{end} \mid \dagger\langle\eta,\eta\rangle \mid \pi.\eta$	ended session type
ρ	$::=$	$\pi \mid \eta$	running session type
s	$::=$	$\mathsf{begin}.\eta \mid \mathsf{sch}$	shared session type
θ	$::=$	$\rho \mid \mathsf{begin}.\rho \mid \updownarrow$	session type
t	$::=$	$C \mid \mathsf{bool} \mid \mathsf{s} \mid (\mathsf{s},\bar{\mathsf{s}})$	standard type

Fig. 8. Syntax of types

that the boolean that determines the branch is to be received instead. The *iterative* partial session type $\dagger\langle\pi\rangle^{*}$ describes sessions that respectively send or receive a boolean value, and if that value is true continue with π, *iterating*, while if the value is false, continue to the following partial session types, if any.

The partial session types $!(\eta)$ and $?(\eta)$ represent the exchange of a live channel, and therefore of an active session, with remaining communications determined by the ended session type η. Note that typing the live channel by η instead of π ensures that this channel is no longer used in the sending thread. In fact each successive use of the channel should concatenate η with a not empty running session type, but this concatenation is not allowed, see Definition 6.1. Example 5.2 shows why this is necessary.

An *ended session type*, η, is a partial session type concatenated either with end or with a conditional whose branches in turn are both ended session types. It expresses a sequence of communications with its termination, *i.e.*, no further communications on that channel are allowed at the end. A conditional ended session type allows to type spawns or connects in the branches. For example, the channel `c1` in the body of method `sell` in Fig. 3 cannot be typed by

```
begin.?String.!double.?<!Address.?DeliveryDetails,ε>.end
```

because the branching in line 23 contains a spawn.

We use ρ to range over both partial session types and ended session types: we call it a *running session type*.

A *shared session type*, s, starts with the keyword begin and has one or more endpoints, denoted by end. Between the start and each ending point, a sequence of session parts describe the communication protocol. The shared session type sch is used for those shared channels that are free in a thread, and which can be used according to any type respecting the (dynamic) duality check of rule **Connect** (Fig. 7).

The typing rules ensure that this type cannot be used directly to describe a session, but it is necessary for defining *freshness* of channels.

A *session type* θ is a running session type, possibly prefixed by begin, so possibly a shared session type, or $\updownarrow$. We use $\updownarrow$ when typing threads, to indicate the type of a channel which is being used by two threads in complementary ways.

Standard types, t, are either class identifiers (C), or booleans (bool), or shared session types (s), or pairs of shared session types with their duals (*i.e.*, $(\mathsf{s},\overline{\mathsf{s}})$).

Each session type θ except for $\updownarrow$ has a corresponding *dual*, denoted $\overline{\theta}$, which is obtained as follows:

- $\overline{?} = !\quad \overline{!} = ?$
- $\overline{\mathsf{begin}.\rho} = \mathsf{begin}.\overline{\rho}$
- $\overline{\pi.\mathsf{end}} = \overline{\pi}.\mathsf{end}\quad \overline{\pi.\dagger\langle\eta_1,\eta_2\rangle} = \overline{\pi}.\overline{\dagger}\langle\overline{\eta_1},\overline{\eta_2}\rangle$
- $\overline{\varepsilon} = \varepsilon\quad \overline{\dagger\mathsf{t}} = \overline{\dagger}\mathsf{t}\quad \overline{\dagger(\eta)} = \overline{\dagger}(\eta)$
  $\overline{\dagger\langle\pi_1,\pi_2\rangle} = \overline{\dagger}\langle\overline{\pi_1},\overline{\pi_2}\rangle\quad \overline{\dagger\langle\pi\rangle^*} = \overline{\dagger}\langle\overline{\pi}\rangle^*\quad \overline{\pi_1.\pi_2} = \overline{\pi_1}.\overline{\pi_2}$

Note that, in the fourth line, the type of the value to be sent (received) in output (input) is not dualised, as it should be the same for both sides of a session. The same applies to the communication of live channels. Also, observe that duality is an involution, *i.e.*, $\theta = \overline{\overline{\theta}}$.

Type System We type expressions and threads with respect to a fixed, global class table $\mathtt{CT}$, as reflected in the rules of Fig. 9 which define well-formed standard types. By $\mathscr{D}(\mathtt{CT})$ we denote the domain of the class table $\mathtt{CT}$, *i.e.*, the set of classes declared in $\mathtt{CT}$. We assume $\mathtt{CT}$ satisfies some usual sanity conditions as in FJ [37]. [1] In the same figure we also define subtyping, $<:$, on class names: we assume that the subtyping between classes is acyclic as in [37]. In addition, we have $(\mathsf{s},\overline{\mathsf{s}}) <: \mathsf{s}$ and $(\mathsf{s},\overline{\mathsf{s}}) <: \overline{\mathsf{s}}$, as in standard π-calculus channel subtyping rules [33]: a channel on which both communication directions are allowed may also transmit data following only one of the two directions.

The typing judgement for threads has two environments, *i.e.*, has the shape:

$$\Gamma;\Sigma \vdash P : \mathsf{thread}$$

where the *standard environment* Γ associates standard types to this, parameters, objects, and shared channels, while the *session environment* Σ contains only judgements for live channel names and channel variables. Fig. 9 defines well formedness

[1] Note, that we could easily have extended the syntax to allow dynamic class creation, but this is orthogonal to session typing.

Well-formed Standard Types

Class
$$\frac{C \in \mathscr{D}(\mathtt{CT})}{\vdash C : \mathtt{tp}}$$

Wf-Session
$$\frac{}{\vdash \mathsf{s} : \mathtt{tp}}$$

Pair
$$\frac{}{\vdash (\mathsf{s},\bar{\mathsf{s}}) : \mathtt{tp}}$$

Bool
$$\frac{}{\vdash \mathsf{bool} : \mathtt{tp}}$$

Subtyping

$$\frac{}{(\mathsf{s},\bar{\mathsf{s}}) <: \mathsf{s}} \qquad \frac{}{(\mathsf{s},\bar{\mathsf{s}}) <: \bar{\mathsf{s}}} \qquad \frac{C \in \mathscr{D}(\mathtt{CT})}{C <: C} \qquad \frac{C <: D \qquad D <: E}{C <: E}$$

$$\frac{\mathsf{class}\ C\ \mathsf{extends}\ D\ \{\widetilde{\mathtt{f}\,\mathsf{t}}\ \tilde{M}\} \in \mathtt{CT}}{C <: D}$$

Standard Environments, and Well-formed Standard Environments

$$\Gamma \ ::= \ \emptyset \mid \Gamma, \mathtt{this} : C \mid \Gamma, \mathsf{x} : \mathsf{t} \mid \Gamma, \mathsf{o} : C \mid \Gamma, \mathsf{c} : \mathsf{sch}$$

Emp
$$\frac{}{\emptyset \vdash \mathsf{ok}}$$

Ethis
$$\frac{\Gamma \vdash \mathsf{ok} \quad C \in \mathscr{D}(\mathtt{CT}) \quad \mathsf{this} \notin \mathscr{D}(\Gamma)}{\Gamma, \mathsf{this} : C \vdash \mathsf{ok}}$$

EVar
$$\frac{\Gamma \vdash \mathsf{ok} \quad \vdash \mathsf{t} : \mathtt{tp} \quad \mathsf{x} \notin \mathscr{D}(\Gamma)}{\Gamma, \mathsf{x} : \mathsf{t} \vdash \mathsf{ok}}$$

EOid
$$\frac{\Gamma \vdash \mathsf{ok} \quad C \in \mathscr{D}(\mathtt{CT}) \quad \mathsf{o} \notin \mathscr{D}(\Gamma)}{\Gamma, \mathsf{o} : C \vdash \mathsf{ok}}$$

ECha
$$\frac{\Gamma \vdash \mathsf{ok} \quad \mathsf{c} \notin \mathscr{D}(\Gamma)}{\Gamma, \mathsf{c} : \mathsf{sch} \vdash \mathsf{ok}}$$

Session Environments, and Well-formed Session Environments

$$\Sigma \ ::= \ \emptyset \mid \Sigma, \mathsf{u} : \theta$$

SEmp
$$\frac{}{\emptyset \vdash \mathsf{ok}}$$

SERC
$$\frac{\Sigma \vdash \mathsf{ok} \quad \mathsf{u} \notin \mathscr{D}(\Sigma)}{\Sigma, \mathsf{u} : \theta \vdash \mathsf{ok}}$$

Fig. 9. Standard Types, Subtyping, and Environments

of standard and session environments, where the domain of an environment is defined as usual and denoted by $\mathscr{D}()$.

As we already discussed in Example 5.4, in order to avoid session interleaving, we need to distinguish the unique (if any) channel identifier currently used to communicate data. Therefore we record a third set, the *hot set* $\mathscr{S}$, which is either empty, or contains a single channel identifier belonging to the session environment. Thus the typing judgement for expressions has the shape:

$$\Gamma;\Sigma;\mathscr{S} \vdash \mathsf{e} : \mathsf{t}$$

where $\mathscr{S}$ is either $\emptyset$ or $\{\mathsf{u}\}$ with $\mathsf{u} \in \mathscr{D}(\Sigma)$.

We adopt the convention that typing rules are applicable only when the session environments in the conclusions are defined.

Expressions The typing rules for expressions are given in Fig. 10 and Fig. 11. Looking at these rules two observations on hot sets are immediate:

- in all rules except **Conn**, **ReceiveS**, **Weak** and **WeakB** the hot sets of all the premises and of the conclusion coincide;
- in all rules whose conclusion is a session expression the hot set of the conclusion is the subject of the session expression.

These two conditions ensure that if rule **WeakB** is not applied in deriving the type of an expression or thread, then all communications use the same live channel, and therefore sessions are not interleaved. This is proved in Lemma 7.9.

In rule **Conn** the ended session type becomes shared, and therefore in the conclusion the hot set is empty.

The condition $\eta \neq \varepsilon.\mathsf{end}$ in rules **SendS** and **ReceiveS** ensures that the exchanged channels have not yet been consumed. This requirement simplifies the progress proof, since it guarantees that all live channels have a type different from $\varepsilon.\mathsf{end}$ (see Lemma B.2(1)). Since $\mathsf{u}\,.\mathsf{receiveS}\,(\mathsf{x})\{\mathsf{e}\}$ in rule **ReceiveS** receives along the live channel u a channel that will replace x, the hot set of the premise is $\{\mathsf{x}\}$, while that of the conclusion is $\{\mathsf{u}\}$. Example 5.6 justifies the requirement that x is the only live channel of e.

Lastly, rule **Weak** replaces an empty hot set by a set containing an arbitrary element.

Notice that, in the derivation of a judgment of the shape $\Gamma;\emptyset;\mathscr{S} \vdash \mathsf{e} : \mathsf{t}$ (*i.e.*, where the session environment is empty) the type rule **WeakB** has never been used. This is so, because after using rule **WeakB** the session environment will contain a premise whose predicate is a session type starting with begin, and rules **Conn**, **ReceiveS** cannot discharge such a kind of premises. See the discussion on the typing of Example 5.4 in Subsection 6.1.

The session environments of the conclusions are obtained from those of the premises and possibly other session environments using the *concatenation* operator, $\circ$, defined below. The typing rules concatenate the session environments to take into account the order of execution of expressions.

Typing Rules for Values

Chan
$$\frac{\Gamma, \mathsf{c} : \mathsf{sch} \vdash \mathsf{ok}}{\Gamma, \mathsf{c} : \mathsf{sch}; \emptyset; \emptyset \vdash \mathsf{c} : (\mathsf{s}, \overline{\mathsf{s}})}$$

Null
$$\frac{\Gamma \vdash \mathsf{ok} \quad \vdash \mathsf{t} : \mathtt{tp}}{\Gamma; \emptyset; \emptyset \vdash \mathsf{null} : \mathsf{t}}$$

Oid
$$\frac{\Gamma, \mathsf{o} : C \vdash \mathsf{ok}}{\Gamma, \mathsf{o} : C; \emptyset; \emptyset \vdash \mathsf{o} : C}$$

True
$$\frac{\Gamma \vdash \mathsf{ok}}{\Gamma; \emptyset; \emptyset \vdash \mathsf{true} \ : \mathsf{bool}}$$

False
$$\frac{\Gamma \vdash \mathsf{ok}}{\Gamma; \emptyset; \emptyset \vdash \mathsf{false} \ : \mathsf{bool}}$$

Typing Rules for Standard Expressions

Var
$$\frac{\Gamma, \mathsf{x} : \mathsf{t} \vdash \mathsf{ok}}{\Gamma, \mathsf{x} : \mathsf{t}; \emptyset; \emptyset \vdash \mathsf{x} : \mathsf{t}}$$

This
$$\frac{\Gamma, \mathsf{this} : C \vdash \mathsf{ok}}{\Gamma, \mathsf{this} : C; \emptyset \vdash \mathsf{this} : C}$$

Fld
$$\frac{\Gamma; \Sigma; \mathscr{S} \vdash \mathsf{e} : C \quad \mathsf{f}\,\mathsf{t} \in \mathsf{fields}(C)}{\Gamma; \Sigma; \mathscr{S} \vdash \mathsf{e.f} : \mathsf{t}}$$

Seq
$$\frac{\Gamma; \Sigma; \mathscr{S} \vdash \mathsf{e} : \mathsf{t} \qquad \Gamma; \Sigma'; \mathscr{S} \vdash \mathsf{e}' : \mathsf{t}'}{\Gamma; \Sigma \circ \Sigma'; \mathscr{S} \vdash \mathsf{e}; \mathsf{e}' : \mathsf{t}'}$$

FldAss
$$\frac{\Gamma; \Sigma; \mathscr{S} \vdash \mathsf{e} : C \qquad \Gamma; \Sigma'; \mathscr{S} \vdash \mathsf{e}' : \mathsf{t} \quad \mathsf{f}\,\mathsf{t} \in \mathsf{fields}(C)}{\Gamma; \Sigma \circ \Sigma'; \mathscr{S} \vdash \mathsf{e.f} := \mathsf{e}' : \mathsf{t}}$$

NewC
$$\frac{\Gamma \vdash \mathsf{ok} \quad C \in \mathscr{D}(\mathtt{CT})}{\Gamma; \emptyset; \emptyset \vdash \ \mathsf{new}\ C : C}$$

NewS
$$\frac{\Gamma \vdash \mathsf{ok}}{\Gamma; \emptyset; \emptyset \vdash \ \mathsf{new}\ (\mathsf{s}, \overline{\mathsf{s}}) : (\mathsf{s}, \overline{\mathsf{s}})}$$

Spawn
$$\frac{\Gamma; \Sigma; \mathscr{S} \vdash \mathsf{e} : \mathsf{t} \quad \mathit{ended}(\Sigma)}{\Gamma; \Sigma; \mathscr{S} \vdash \mathsf{spawn}\,\{\ \mathsf{e}\ \} : \mathit{Object}}$$

NullPE
$$\frac{\Gamma \vdash \mathsf{ok} \quad \vdash \mathsf{t} : \mathtt{tp}}{\Gamma; \emptyset; \emptyset \vdash \mathsf{NullExc} : \mathsf{t}}$$

MethMinus
$$\frac{\begin{array}{c}\Gamma; \Sigma_0; \mathscr{S} \vdash \mathsf{e} : C \quad \Gamma; \Sigma_i; \mathscr{S} \vdash \mathsf{e}_i : \mathsf{t}_i \quad i \in \{1 \ldots n\} \\ \mathsf{mtype}(\mathsf{m}, C) = \mathsf{t}_1, \ldots, \mathsf{t}_n, \rho_1, \ldots, \rho_m \overset{\ominus}{\rightarrow} \mathsf{t}\end{array}}{\Gamma; \Sigma_0 \circ \Sigma_1 \ldots \circ \Sigma_n \circ \{\mathsf{u}_1 : \rho_1, \ldots, \mathsf{u}_m : \rho_m\}; \mathscr{S} \vdash \mathsf{e.m}(\mathsf{e}_1, \ldots, \mathsf{e}_n, \mathsf{u}_1, \ldots, \mathsf{u}_m) : \mathsf{t}}$$

MethPlus
$$\frac{\begin{array}{c}\Gamma; \Sigma_0; \{\mathsf{u}_1\} \vdash \mathsf{e} : C \quad \Gamma; \Sigma_i; \{\mathsf{u}_1\} \vdash \mathsf{e}_i : \mathsf{t}_i \quad i \in \{1 \ldots n\} \\ \mathsf{mtype}(\mathsf{m}, C) = \mathsf{t}_1, \ldots, \mathsf{t}_n, \rho_1, \ldots, \rho_m \overset{\oplus}{\rightarrow} \mathsf{t}\end{array}}{\Gamma; \Sigma_0 \circ \Sigma_1 \ldots \circ \Sigma_n \circ \{\mathsf{u}_1 : \rho_1, \ldots, \mathsf{u}_m : \rho_m\}; \{\mathsf{u}_1\} \vdash \mathsf{e.m}(\mathsf{e}_1, \ldots, \mathsf{e}_n, \mathsf{u}_1, \ldots, \mathsf{u}_m) : \mathsf{t}}$$

Fig. 10. Typing Rules for Expressions I

Typing Rules for Communication Expressions

Conn

$$\frac{\Gamma;\emptyset;\emptyset \vdash \mathsf{u} : \mathsf{begin}.\eta \quad \Gamma\setminus\mathsf{u}\,;\Sigma,\mathsf{u}:\eta;\{\mathsf{u}\} \vdash \mathsf{e}:\mathsf{t}}{\Gamma;\Sigma;\emptyset \vdash \mathsf{connect\ u\ begin}.\eta\ \{\mathsf{e}\}:\mathsf{t}}$$

Send

$$\frac{\Gamma;\Sigma;\{\mathsf{u}\} \vdash \mathsf{e}:\mathsf{t}}{\Gamma;\Sigma\circ\{\mathsf{u} :!\mathsf{t}\};\{\mathsf{u}\} \vdash \mathsf{u}.\mathsf{send}(\ \mathsf{e}\ ) : \mathit{Object}}$$

Receive

$$\frac{\Gamma \vdash \mathsf{ok} \quad \vdash \mathsf{t} : \mathsf{tp}}{\Gamma;\{\mathsf{u} : ?\mathsf{t}\};\{\mathsf{u}\} \vdash \mathsf{u}.\mathsf{receive}:\mathsf{t}}$$

SendS

$$\frac{\Gamma \vdash \mathsf{ok} \quad \eta \neq \varepsilon.\mathsf{end}}{\Gamma;\{\mathsf{u}':\eta,\mathsf{u}:!(\eta)\};\{\mathsf{u}\} \vdash \mathsf{u}.\mathsf{sendS}(\mathsf{u}')\ :\ \mathit{Object}}$$

ReceiveS

$$\frac{\Gamma\setminus\mathsf{x}\,;\ \{\mathsf{x}:\eta\};\{\mathsf{x}\} \vdash \mathsf{e}:\mathsf{t} \quad \eta \neq \varepsilon.\mathsf{end}}{\Gamma;\{\mathsf{u} : ?(\eta)\};\{\mathsf{u}\} \vdash \mathsf{u}.\mathsf{receiveS}(\mathsf{x})\{\mathsf{e}\} : \mathit{Object}}$$

SendIf

$$\frac{\Gamma;\Sigma_0;\{\mathsf{u}\} \vdash \mathsf{e}:\mathsf{bool} \qquad \Gamma;\Sigma,\mathsf{u}:\rho_i;\{\mathsf{u}\} \vdash \mathsf{e}_i:\mathsf{t} \quad i \in \{1,2\}}{\Gamma;\Sigma_0\circ\Sigma,\mathsf{u} :!\langle\rho_1,\rho_2\rangle;\{\mathsf{u}\} \vdash \mathsf{u}.\mathsf{sendIf}(\mathsf{e})\{\mathsf{e}_1\}\{\mathsf{e}_2\}:\mathsf{t}}$$

ReceiveIf

$$\frac{\Gamma;\Sigma,\mathsf{u}:\rho_i;\{\mathsf{u}\} \vdash \mathsf{e}_i:\mathsf{t} \quad i \in \{1,2\}}{\Gamma;\Sigma,\mathsf{u} :?\langle\rho_1,\rho_2\rangle;\{\mathsf{u}\} \vdash \mathsf{u}.\mathsf{receiveIf}\{\mathsf{e}_1\}\{\mathsf{e}_2\}:\mathsf{t}}$$

SendWhile

$$\frac{\Gamma;\emptyset;\emptyset \vdash \mathsf{e}:\mathsf{bool} \qquad \Gamma;\{\mathsf{u}:\pi\};\{\mathsf{u}\} \vdash \mathsf{e}':\mathsf{t}}{\Gamma;\{\mathsf{u} :!\langle\pi\rangle^*\};\{\mathsf{u}\} \vdash \mathsf{u}.\mathsf{sendWhile}(\mathsf{e})\{\mathsf{e}'\}:\mathsf{t}}$$

ReceiveWhile

$$\frac{\Gamma;\{\mathsf{u}:\pi\};\{\mathsf{u}\} \vdash \mathsf{e}:\mathsf{t}}{\Gamma;\{\mathsf{u} :?\langle\pi\rangle^*\};\{\mathsf{u}\} \vdash \mathsf{u}.\mathsf{receiveWhile}\{\mathsf{e}\}:\mathsf{t}}$$

Non-structural Typing Rules for Expressions

WeakES

$$\frac{\Gamma;\ \Sigma;\mathscr{S} \vdash \mathsf{e}:\mathsf{t} \quad \mathsf{u} \notin \mathscr{D}(\Sigma)}{\Gamma;\ \Sigma,\mathsf{u}:\varepsilon;\mathscr{S} \vdash \mathsf{e}:\mathsf{t}}$$

WeakE

$$\frac{\Gamma;\Sigma,\mathsf{u}:\pi;\mathscr{S} \vdash \mathsf{e}:\mathsf{t}}{\Gamma;\Sigma,\mathsf{u}:\pi.\mathsf{end};\mathscr{S} \vdash \mathsf{e}:\mathsf{t}}$$

Consume

$$\frac{\Gamma;\Sigma,\mathsf{u}:\varepsilon.\mathsf{end};\mathscr{S} \vdash \mathsf{e}:\mathsf{t}}{\Gamma;\Sigma,\mathsf{u} :\updownarrow;\mathscr{S} \vdash \mathsf{e}:\mathsf{t}}$$

Sub

$$\frac{\Gamma;\Sigma;\mathscr{S} \vdash \mathsf{e}:\mathsf{t}}{\Gamma;\Sigma;\mathscr{S} \vdash \mathsf{e}:\mathsf{t}'}\ \mathsf{t} <: \mathsf{t}'$$

Weak

$$\frac{\Gamma;\Sigma;\emptyset \vdash \mathsf{e}:\mathsf{t} \quad \mathsf{u} \in \mathscr{D}(\Sigma)}{\Gamma;\Sigma;\{\mathsf{u}\} \vdash \mathsf{e}:\mathsf{t}}$$

WeakB

$$\frac{\Gamma;\ \Sigma,\mathsf{c}:\rho;\{\mathsf{c}\} \vdash \mathsf{e}:\mathsf{t}}{\Gamma;\ \Sigma,\mathsf{c}:\mathsf{begin}.\rho\,;\emptyset \vdash \mathsf{e}:\mathsf{t}}$$

Fig. 11. Typing Rules for Expressions II

The concatenation of two channel types θ and θ' is the unique channel type (if it exists) which prescribes all the communications of θ followed by all those of θ'. The concatenation only exists if θ is a partial session type possibly prefixed by begin, and θ' is a running session type. The concatenation cancels meaningless ε, so for example $\varepsilon\circ\updownarrow=\updownarrow$. The extension to session environments is straightforward. As usual, $\perp$ stands for undefined.

Definition 6.1 (Concatenation)

- $$\theta\circ\theta' = \begin{cases} \theta & \textit{if } \theta' = \varepsilon \\ \theta' & \textit{if } \theta = \varepsilon \\ \theta.\mathsf{end} & \textit{if } \theta' = \varepsilon.\mathsf{end} \textit{ and} \\ & \theta \textit{ is a partial session type possibly prefixed by } \mathsf{begin} \\ \mathsf{begin}.\theta' & \textit{if } \theta = \mathsf{begin}.\varepsilon \textit{ and } \theta' \textit{ is a running session type} \\ \theta.\theta' & \textit{if } \theta \textit{ is a partial session type possibly prefixed by } \mathsf{begin} \\ & \textit{and } \theta' \textit{ is a running session type} \\ \perp \quad \textit{otherwise}. \end{cases}$$

- $$\Sigma \setminus \Sigma' = \{\mathsf{u} : \Sigma(\mathsf{u}) \mid \mathsf{u} \in \mathscr{D}(\Sigma) \setminus \mathscr{D}(\Sigma')\}$$

- $$\Sigma\circ\Sigma' = \begin{cases} \Sigma\setminus\Sigma' \cup \Sigma'\setminus\Sigma \cup \{\mathsf{u} : \Sigma(\mathsf{u}) \circ \Sigma'(\mathsf{u}) \mid \mathsf{u} \in \mathscr{D}(\Sigma) \cap \mathscr{D}(\Sigma')\} \\ \qquad \textit{if} \quad \forall \mathsf{u} \in \mathscr{D}(\Sigma) \cap \mathscr{D}(\Sigma') : \ \Sigma(\mathsf{u}) \circ \Sigma'(\mathsf{u}) \neq \perp; \\ \perp \quad \textit{otherwise}. \end{cases}$$

In the above definition we avoided the occurrence of meaningless ε, *e.g.*, we never create the session type $!\mathsf{bool}.\varepsilon$. This is why the definition considers several different cases. An alternative would allow the occurrence of meaningless ε, and would consider session types which differ for ε occurrences as equivalent.

In the following we discuss the most interesting typing rules for expressions.

Rule **Spawn** requires that all sessions used by the spawned thread are finally consumed, *i.e.*, they are all ended session types. This is necessary in order to preserve the bilinearity condition, *e.g.*, avoid configuartions such as $\mathsf{spawn}\,\{\,\mathsf{c}.\mathsf{send}\,(1)\,\};\mathsf{c}.\mathsf{send}\,(\mathsf{true})$. To guarantee the consumption we define:

$$\textit{ended}(\Sigma) = \forall \mathsf{u} : \theta \in \Sigma.\ \ \theta \text{ is an ended session type.}$$

For example, $\textit{ended}(\{\mathsf{c} : ?\mathsf{bool}.\mathsf{end}, \mathsf{c}' : !\langle ?\mathsf{bool}.\mathsf{end}, !\mathsf{bool}.\mathsf{end}\rangle\})$ holds, while, on the other hand $\textit{ended}(\{\mathsf{c} : ?\mathsf{bool}\})$ does *not* hold.

MMinus-ok

$$\frac{\mathsf{mtype}(\mathtt{m},C) = \mathtt{t}_1,\ldots,\mathtt{t}_n,\rho_1,\ldots,\rho_m \overset{\ominus}{\to} \mathtt{t} \qquad \{\mathtt{this}:C,\widetilde{\mathtt{x}:\mathtt{t}}\};\ \{\widetilde{\mathtt{y}:\rho}\};\ \emptyset \vdash \mathtt{e}:\mathtt{t}}{\mathtt{t}\ \mathtt{m}\ (\widetilde{\mathtt{t}\,\mathtt{x}},\widetilde{\rho\,\mathtt{y}})\ \{\mathtt{e}\}:\mathtt{ok}\ \mathtt{in}\ C}$$

MPlus-ok

$$\frac{\mathsf{mtype}(\mathtt{m},C) = \mathtt{t}_1,\ldots,\mathtt{t}_n,\rho_1,\ldots,\rho_m \overset{\oplus}{\to} \mathtt{t} \qquad \{\mathtt{this}:C,\widetilde{\mathtt{x}:\mathtt{t}}\};\ \{\widetilde{\mathtt{y}:\rho}\};\ \{\mathtt{y}_1\} \vdash \mathtt{e}:\mathtt{t}}{\mathtt{t}\ \mathtt{m}\ (\widetilde{\mathtt{t}\,\mathtt{x}},\widetilde{\rho\,\mathtt{y}})\ \{\mathtt{e}\}:\mathtt{ok}\ \mathtt{in}\ C}$$

C-ok

$$\frac{\mathsf{mtype}(\mathtt{m},D)\ \text{defined} \implies \mathsf{mtype}(\mathtt{m},C) = \mathsf{mtype}(\mathtt{m},D) \qquad \tilde{M}:\mathtt{ok}\ \mathtt{in}\ C}{\mathsf{class}\ C\ \mathsf{extends}\ D\ \{\widetilde{\mathtt{f}\,\mathtt{t}}\ \tilde{M}\}:\mathtt{ok}}$$

CT-ok

$$\frac{\mathsf{class}\ C\ \mathsf{extends}\ D\ \{\widetilde{\mathtt{f}\,\mathtt{t}}\ \tilde{M}\}:\mathtt{ok} \qquad D = \mathsf{Object}\ \text{or}\ D\ \text{defined in}\ \mathtt{CT} \qquad \mathtt{CT}:\mathtt{ok}}{\mathtt{CT},\mathsf{class}\ C\ \mathsf{extends}\ D\ \{\widetilde{\mathtt{f}\,\mathtt{t}}\ \tilde{M}\}:\mathtt{ok}}$$

Fig. 12. Well-formed Class Tables

Rules **MethMinus** and **MethPlus** retrieve the type of the method $\mathtt{m}$ from the class table using the auxiliary function $\mathsf{mtype}(\mathtt{m},C)$. The session environments of the premises are concatenated with $\{\mathtt{u}_1:\rho_1,\ldots,\mathtt{u}_m:\rho_m\}$, which represents the communication protocols of the live channels $\mathtt{u}_1,\ldots,\mathtt{u}_m$ during the execution of the method body. Rule **MethMinus** requires the hot sets of all the premises and of the conclusion to be the same. Rule **MethPlus** expects the actual parameter $\mathtt{u}_1$ to be a channel identifier that will be used within the method body directly as if it was part of an open session. Therefore the hot sets of all the premises and of the conclusion must be $\{\mathtt{u}_1\}$. We call $\mathtt{u}_1$ the *subject of the method call.* So a call of methods whose type is decorated by $\oplus$ has a subject, while a call of methods by $\ominus$ has no subject.

Rule **Conn** ensures that a session body properly uses its unique channel according to the required session type. The first premise says that the channel identifier used for the session ($\mathtt{u}$) can be typed with the appropriate shared session type ($\mathsf{begin}.\eta$). The second premise ensures that the session body can be typed in the restricted environment $\Gamma \setminus \mathtt{u}$ with a session environment containing $\mathtt{u}:\eta$ and with hot set $\{\mathtt{u}\}$.

Lastly, in rules **ReceiveIF** and **SendIF** *both* ρ_1 and ρ_2 are either partial session types or ended session types – this is guaranteed by the syntax of conditional session types (see Fig. 8).

We discuss the non-structural rules in Subsection 6.1.

Class Tables Fig. 12 defines well-formed class tables. Note that we expect the selection of the $\circledcirc$ in the method type lookup function $\mathsf{mtype}(..,..)$ to correctly pick between $\ominus$ and $\oplus$ so as to satisfy rules **MMinus** − **ok** and **MPlus** − **ok**, which type-check the method bodies with respect to a class C taking as environments the association between formal parameters and their types and the association between this and C. These rules differ in the hot sets used to type the method bodies; thus **MPlus** − **ok** allows a receive or send on the first channel parameter, while **MMinus** − **ok** does not allow any send or receive on the channel parameters.

In keeping with [37], we leave implicit the requirement that methods are not overloaded, *i.e.*, that no method is defined more than once in a class body, and that no field is declared more than once in a class hierarchy. Also in keeping with [37], we explicitly require that method overriding preserves the type of the overridden method.

Start

$$\frac{\Gamma;\Sigma\vdash \mathsf{e}:\mathsf{t}}{\Gamma;\Sigma\vdash \mathsf{e}:\mathsf{thread}}$$

Par

$$\frac{\Gamma;\Sigma_i\vdash P_i:\mathsf{thread}\quad i\in\{1,2\}}{\Gamma;\Sigma_1\|\Sigma_2\vdash P_1\,|\,P_2:\mathsf{thread}}$$

Fig. 13. Typing Rules for Threads

Thread In the typing rules for threads, we need to take into account that the same channel can occur with dual types in the session environments of two premises. For this reason we compose the session environments of the premises using the *parallel composition*, $\|$.

Definition 6.2 *We define parallel composition, $\|$, on session types and on session environments as follows:*

$$\theta\|\theta' = \begin{cases} \updownarrow & \textit{if} \quad \theta=\overline{\theta'} \\ \bot & \textit{otherwise.} \end{cases}$$

$$\Sigma\|\Sigma' = \begin{cases} \Sigma\setminus\Sigma' \cup \Sigma'\setminus\Sigma \cup \{\mathsf{u}:\Sigma(\mathsf{u})\,\|\,\Sigma'(\mathsf{u})\mid \mathsf{u}\in\mathscr{D}(\Sigma)\cap\mathscr{D}(\Sigma')\} \\ \qquad \textit{if} \quad \forall \mathsf{u}\in\mathscr{D}(\Sigma)\cap\mathscr{D}(\Sigma'):\ \Sigma(\mathsf{u})\,\|\,\Sigma'(\mathsf{u})\neq\bot \\ \bot \quad \textit{otherwise.} \end{cases}$$

Note that $\updownarrow\|\theta=\theta\|\updownarrow=\bot$.

Using the operator $\|$ the typing rules for processes are straightforward (see Fig. 13). Rule **Start** promotes an expression to the thread level; and rule **Par** types a composition of threads if the composition of their session environments is defined.

In writing session environments we assume the following operator precedence: , $\circ$ $\|$. For example $\Sigma_0, \mathsf{c} : \pi \circ \Sigma_1 \| \Sigma_2$ is short for $((\Sigma_0, \mathsf{c} : \pi) \circ \Sigma_1) \| \Sigma_2$.

6.1 Justifying Examples

In this subsection we discuss the typing of the threads shown in § 5 and we also give examples justifying the non-structural rules, except for rule **Sub** which is obvious.

Example 5.1: The thread $P_1 \,|\, P_2$ is not typable since the parallel composition of the corresponding session environments is undefined.

Example 5.2: The thread P_1 cannot be typed since:

- the expression in line 3 can only be typed by rule **SendS** which requires for the sent channel c_1 a live channel type terminating by end in the session environment;
- the expression in line 4 can only be typed by rule **Receive** which requires also a live channel type different from ε for the channel c_1 in the session environment;
- to type the composition of these two expressions, **Seq** requires the concatenation of the corresponding session environments to be defined, but this is false since a type terminating by end cannot be concatenated to a live channel type different from ε.

Examples 5.3 and 5.4: Neither thread can be typed. For example, to type the expressions in line 3 in P_1 using rules **Send**, and **Receive**, $\{\mathsf{c}_1\}$ and $\{\mathsf{c}_2\}$ should be the hot sets, respectively. Notice that rule **Seq** requires the premises to share the same hot set. We could use rules **WeakES** and **WeakB** to force the hot set of the first premise to be $\{\mathsf{c}_2\}$, but then rule **Conn** would not be applicable to type the whole expression P_1.

Example 5.5: It is clear from rules **MethMinus** and **MethPlus** that the hot sets of the receivers, of the actual parameters and of the method bodies cannot be two different live channels.

Use of non-structural rules.

Rule **WeakES**, where **ES** stands for empty session, is necessary to add a channel to a session environment and rule **Weak** is used to specify a hot set. Look for example at the typing of $\mathsf{c}.\mathsf{send}(\mathsf{true})$, as shown in Fig. 14.

With rule **WeakES** we can derive $\emptyset; \{\mathsf{c} : \varepsilon\}; \emptyset \vdash \mathsf{null} : \mathsf{thread}$ and then with rules **Start** and **Par** we can derive $\emptyset; \{\mathsf{c} : \updownarrow\} \vdash \mathsf{null} | \mathsf{null} : \mathsf{thread}$. Since $\mathsf{null} | \mathsf{null} \equiv \mathsf{null}$, in order to have type preservation under structural equivalence we need to be able to also derive that $\emptyset; \{\mathsf{c} : \updownarrow\} \vdash \mathsf{null} : \mathsf{thread}$. This gives the motivation for rule **Consume**.

$$\dfrac{\dfrac{\dfrac{\emptyset;\emptyset;\emptyset\vdash \mathsf{true}:\mathsf{bool}}{\emptyset;\{\mathsf{c}:\varepsilon\};\emptyset\vdash \mathsf{true}:\mathsf{bool}}}{\emptyset;\{\mathsf{c}:\varepsilon\};\{\mathsf{c}\}\vdash \mathsf{true}:\mathsf{bool}}}{\emptyset;\{\mathsf{c}:!\mathsf{bool}\};\{\mathsf{c}\}\vdash \mathsf{c.send\,(true)}:\mathit{Object}}$$

Fig. 14. A Type Derivation using Rules **WeakES** and **Weak**

The derivation works as follows: use rules **Null**, **WeakES** and **WeakE** to obtain $\emptyset;\{\mathsf{c}:\varepsilon.\mathsf{end}\};\emptyset\vdash\mathsf{null}:\mathit{Object}$. Then, apply **Consume** and obtain $\emptyset;\{\mathsf{c}:\updownarrow\};\emptyset\vdash\mathsf{null}:\mathit{Object}$. Then, apply **Start** and obtain $\emptyset;\{\mathsf{c}:\updownarrow\}\vdash\mathsf{null}:\mathsf{thread}$.

The design of rule **Consume** is delicate, and we considered several alternatives. We chose to start from the predicate ε.end for the same subject u, since this simplifies the proof of Lemma 7.2 (see Appendix A). Moreover, we chose to design rule **Consume** for expressions (and not for processes) since this gives us the property that $\Gamma;\Sigma\vdash\mathsf{e}:\mathsf{thread}$ implies $\Gamma;\Sigma;\mathscr{S}\vdash\mathsf{e}:\mathsf{t}$ for some $\mathscr{S},\mathsf{t}$. This property significantly simplifies the proof of subject reduction.

Rule **WeakE**, where **E** stands for end, allows us to obtain ended session types as predicates of session environments, as required in order to be able to apply rules **Conn**, **Spawn**, **ReceiveS**. For example, through application of **True**, **Weak**, **Send**, we obtain $\emptyset;\{\mathsf{u}:!\mathsf{bool}\};\{\mathsf{u}\}\vdash\mathsf{u.send\,(true)}:\mathsf{bool}$. Then, through application of **WeakE** we obtain $\emptyset;\{\mathsf{u}:!\mathsf{bool.end}\};\{\mathsf{u}\}\vdash\mathsf{u.send\,(true)}:\mathsf{bool}$. Then, **Spawn** is applicable, and gives $\emptyset;\{\mathsf{u}:!\mathsf{bool.end}\};\{\mathsf{u}\}\vdash\mathsf{spawn}\{\,\mathsf{u.send\,(true)}\,\}:\mathit{Object}$.

Rule **WeakB**, where **B** stands for begin, is necessary for type preservation under execution. For example, consider the threads P_1 and P_2 defined as follows:

```
connect c1 begin.!bool.end {
    connect c2 begin.!bool.end {
      c2.send(true) };
c1.send(false) }
```

P_1

```
connect c1 begin.?bool.end {
    connect c2 begin.?bool.end {
        c2.receive };
c1.receive }
```

P_2

Clearly, we can derive $\emptyset;\emptyset\vdash P_1|P_2:\mathsf{thread}$.

Starting with a heap h, the above two threads in parallel reduce to:

$$\mathsf{c}_1'.\mathsf{send\,(true)}\,;\,\mathsf{c}_2'.\mathsf{send\,(false)}\,|\,\mathsf{c}_1'.\mathsf{receive}\,;\,\mathsf{c}_2'.\mathsf{receive}\,,\qquad h::\mathsf{c}_1'::\mathsf{c}_2'$$

where c_1' and c_2' are the fresh live channels, that replaced respectively c_1 and c_2 when the sessions began. Fig. 15 shows a typing for $\mathsf{c}_1'.\mathsf{receive}\,;\,\mathsf{c}_2'.\mathsf{receive}$; the first rule on the right is **WeakB**.

$$\dfrac{\emptyset;\{\mathsf{c}'_1 :?\mathsf{bool}\};\{\mathsf{c}'_1\} \vdash \mathsf{c}'_1.\mathsf{receive} : \mathsf{bool} \qquad \dfrac{\dfrac{\emptyset;\{\mathsf{c}'_2 :?\mathsf{bool}\};\{\mathsf{c}'_2\} \vdash \mathsf{c}'_2.\mathsf{receive} : \mathsf{bool}}{\emptyset;\{\mathsf{c}'_2 : \mathsf{begin}.?\mathsf{bool}\};\emptyset \vdash \mathsf{c}'_2.\mathsf{receive} : \mathsf{bool}}}{\emptyset;\{\mathsf{c}'_2 : \mathsf{begin}.?\mathsf{bool}\};\{\mathsf{c}'_1\} \vdash \mathsf{c}'_2.\mathsf{receive} : \mathsf{bool}}}{\emptyset;\{\mathsf{c}'_1 :?\mathsf{bool}, \mathsf{c}'_2 : \mathsf{begin}.?\mathsf{bool}\};\{\mathsf{c}'_1\} \vdash \mathsf{c}'_1.\mathsf{receive}\ ;\ \mathsf{c}'_2.\mathsf{receive} : \mathsf{bool}}$$

Fig. 15. A Type Derivation using Rule **WeakB**

HCha
$$\dfrac{\mathsf{c} \in \mathscr{D}(h) \quad (\mathsf{s}, \bar{\mathsf{s}}) <: \mathsf{s}'}{h \vdash \mathsf{c} : \mathsf{s}'}$$

HNull
$$\dfrac{C \in \mathscr{D}(\mathtt{CT})}{h \vdash \mathsf{null} : C}$$

HTrue
$$\dfrac{}{h \vdash \mathsf{true} : \mathsf{bool}}$$

HFalse
$$\dfrac{}{h \vdash \mathsf{false} : \mathsf{bool}}$$

HObj
$$\dfrac{h(\mathsf{o}) = (C', \ldots) \qquad C' <: C}{h \vdash \mathsf{o} : C}$$

WfObj
$$\dfrac{h(\mathsf{o}) = (C, \widetilde{\mathsf{f} : \mathsf{v}}) \qquad \mathsf{fields}(C) = \widetilde{\mathsf{f}\ \mathsf{t}} \quad h \vdash \mathsf{v}_i : \mathsf{t}_i}{h \vdash \mathsf{o}}$$

WfHeap
$$\dfrac{\begin{array}{l}\forall \mathsf{o} \in \mathscr{D}(h) :\ h \vdash \mathsf{o} \qquad \forall \mathsf{o} \in \mathscr{D}(\Gamma) :\ h \vdash \mathsf{o} : \Gamma(\mathsf{o}) \\ \forall \mathsf{c} \in \mathscr{D}(\Gamma) \cup \mathscr{D}(\Sigma) :\ \mathsf{c} \in h \qquad \mathscr{D}(\Gamma) \cap \mathscr{D}(\Sigma) = \emptyset\end{array}}{\Gamma;\Sigma \vdash h}$$

Fig. 16. Types of Runtime Entities, and Well-formed Heaps

7 Type Safety and Communication Safety

We will consider only reductions of well-typed expressions and threads. We define agreement between environments and heaps in the standard way and we denote it by $\Gamma;\Sigma \vdash h$. The judgment is defined in Fig. 16. The judgment $h \vdash \mathsf{v} : \mathsf{t}$ guarantees that the runtime value v has type t. In rule **HCha** we use $<:$ in order to write only one rule, which allows to derive types of both shapes ($(\mathsf{s}, \bar{\mathsf{s}})$ and s). For objects we take subclasses into consideration in rule **HObj**. The judgment $h \vdash \mathsf{o}$ guarantees that the object o is well formed, *i.e.*, that its fields contain values according to the declared field types in C, the class of that object. The judgment $\Gamma;\Sigma \vdash h$ guarantees that the heap is well formed for Γ and Σ, *i.e.*, that all objects are well formed, all objects in the domain of Γ have a class which is a superclass of their declared class in h, all channels in the domain of Γ and of Σ are channels in h.

We define $\Gamma;\Sigma;\mathscr{S} \vdash \mathsf{e};h$, as a shorthand for $\Gamma;\Sigma;\mathscr{S} \vdash \mathsf{e} : \mathsf{t}$ for some t and $\Gamma;\Sigma \vdash h$. Similarly $\Gamma;\Sigma \vdash P;h$ means $\Gamma;\Sigma \vdash P : \mathsf{thread}$ and $\Gamma;\Sigma \vdash h$.

In this section, we outline the proof of subject reduction, while we give full details and proofs in Appendix A.

As usual, we use Generation Lemmas. The Generation Lemmas in this work are somewhat unusual, because, due to the non-structural rules, when an expression is typed, we only can deduce *some* information about the session environment and hot set used in the typing. For example, $\Gamma;\Sigma;\mathscr{S} \vdash \mathsf{x}:\mathsf{t}$ does *not* imply that $\Sigma = \emptyset$; instead, it implies that $\mathscr{R}(\Sigma) \subseteq \{\varepsilon, \varepsilon.\mathsf{end}, \mathsf{begin}.\varepsilon.\mathsf{end}, \mathsf{begin}.\varepsilon\}$, where $\mathscr{R}(\Sigma)$ is the range of Σ.

In order to express the Generation Lemmas, we define the partial order $\preceq$ among pairs of session environments, and hot sets, which basically reflects the differences introduced through the application of nonstructural rules.

Definition 7.1 (Weakening Order $\preceq$) *(1)* $\Sigma;\mathscr{S} \preceq \Sigma';\mathscr{S}'$ *is the smallest partial order such that:*

- $\Sigma;\mathscr{S} \preceq \Sigma, \mathsf{u} : \varepsilon;\mathscr{S}$ *if* $\mathsf{u} \notin \mathscr{D}(\Sigma)$,
- $\Sigma, \mathsf{u} : \pi;\mathscr{S} \preceq \Sigma, \mathsf{u} : \pi.\mathsf{end};\mathscr{S}$,
- $\Sigma, \mathsf{u} : \varepsilon.\mathsf{end};\mathscr{S} \preceq \Sigma, \mathsf{u} :\updownarrow;\mathscr{S}$,
- $\Sigma;\emptyset \preceq \Sigma;\{\mathsf{u}\}$,
- $\Sigma, \mathsf{c} : \rho;\{\mathsf{c}\} \preceq \Sigma, \mathsf{c} : \mathsf{begin}.\rho;\emptyset$.

(2) $\Sigma \preceq \Sigma'$ *if* $\Sigma;\mathscr{S} \preceq \Sigma';\mathscr{S}'$ *for some* $\mathscr{S}$, $\mathscr{S}'$.

For example $\{\mathsf{c} :?\mathsf{bool}\};\{\mathsf{c}\} \preceq \{\mathsf{c} : \mathsf{begin}.?\mathsf{bool}, \mathsf{c}' :\updownarrow\};\emptyset$.

Lemma 7.2 states that the ordering relation $\preceq$ preserves the types of expressions, and is proven in Appendix A.

Lemma 7.2 *If* $\Sigma;\mathscr{S} \preceq \Sigma';\mathscr{S}'$ *and* $\Gamma;\Sigma;\mathscr{S} \vdash \mathsf{e}:\mathsf{t}$, *then* $\Gamma;\Sigma';\mathscr{S}' \vdash \mathsf{e}:\mathsf{t}$.

Generation Lemmas for standard expressions, communication expressions, and processes are given in Appendix A (see Lemmas A.1, A.2, and A.3) and make use of the relation $\preceq$. For example, $\Gamma;\Sigma;\mathscr{S} \vdash \mathsf{u}.\mathsf{send}(\mathsf{e}):\mathsf{t}$ implies $\mathsf{t} =$ *Object* and $\Gamma;\Sigma';\{\mathsf{u}\} \vdash \mathsf{e}:\mathsf{t}$ and $\Sigma' \circ \{\mathsf{u} : !\mathsf{t}\};\{\mathsf{u}\} \preceq \Sigma;\mathscr{S}$.

Lemma 7.3 states that the typing of $E[\mathsf{e}]$ can be broken down into the typing of e, and the typing of $E[\mathsf{x}]$. Furthermore, Σ, the environment used to type $E[\mathsf{e}]$, can be broken down into two environments, $\Sigma = \Sigma_1 \circ \Sigma_2$, where Σ_1 is used to type e, and Σ_2 is used to type $E[\mathsf{x}]$. The proof is given in Appendix A.

Lemma 7.3 (Subderivations)

If $\Gamma;\Sigma;\mathscr{S} \vdash E[\mathsf{e}]:\mathsf{t}$, *then there exist* $\Sigma_1, \Sigma_2, \mathsf{t}'$, *such that for all* x *fresh in* E,Γ, $\Sigma = \Sigma_1 \circ \Sigma_2$, *and* $\Gamma;\Sigma_1;\mathscr{S} \vdash \mathsf{e}:\mathsf{t}'$, *and* $\Gamma, \mathsf{x} : \mathsf{t}';\Sigma_2;\mathscr{S} \vdash E[\mathsf{x}]:\mathsf{t}$.

On the other hand, Lemma 7.4 allows the combination of the typings of $E[\mathtt{x}]$ and the typing of $\mathtt{e}$, provided that the contexts Σ_1 and Σ_2 used for the two typings can be composed through $\circ$, and that the type of $\mathtt{e}$ is the same as that of $\mathtt{x}$ in the first typing. The proof is given in Appendix A.

Lemma 7.4 (Context Substitution) *If* $\Gamma;\Sigma_1;\mathscr{S} \vdash \mathtt{e} : \mathtt{t}'$*, and* $\Gamma, \mathtt{x} : \mathtt{t}';\Sigma_2;\mathscr{S} \vdash E[\mathtt{x}] : \mathtt{t}$*, and* $\Sigma_1 \circ \Sigma_2$ *is defined, then* $\Gamma;\Sigma_1 \circ \Sigma_2;\mathscr{S} \vdash E[\mathtt{e}] : \mathtt{t}$.

Lemma 7.5 establishes a desirable property of freshness: for a well-typed expression (thread) and an associated well-formed heap, if a channel or object identifier occurs in the expression (thread), then it occurs in the heap too.

Lemma 7.5 (Fresh Name) *(1) If* $\Gamma;\Sigma;\mathscr{S} \vdash \mathtt{e}; h$*, then*
(a) $\mathtt{o} \in \mathtt{e} \Rightarrow \mathtt{o} \in h$*;*
(b) $\mathtt{c} \in \mathtt{e} \Rightarrow \mathtt{o} \in h$*.*
(2) If $\Gamma;\Sigma \vdash P; h$*, then*
(a) $\mathtt{o} \in P \Rightarrow \mathtt{o} \in h$*;*
(b) $\mathtt{c} \in P \Rightarrow \mathtt{o} \in h$*.*

We now state the Subject Reduction theorem:

Theorem 7.6 (Subject Reduction) *(1)* $\Gamma;\Sigma;\mathscr{S} \vdash \mathtt{e} : \mathtt{t}$*, and* $\Gamma;\Sigma \vdash h$*, and* $\mathtt{e}, h \longrightarrow \mathtt{e}', h'$ *imply* $\Gamma';\Sigma;\mathscr{S} \vdash \mathtt{e}' : \mathtt{t}$*, and* $\Gamma';\Sigma \vdash h'$*, with* $\Gamma \subseteq \Gamma'$*.*
(2) $\Gamma;\Sigma \vdash P; h$ *and* $P, h \longrightarrow P', h'$ *imply* $\Gamma';\Sigma' \vdash P; h'$ *with* $\Gamma \subseteq \Gamma'$ *and* $\Sigma \subseteq \Sigma'$*.*

The proof, given in Appendix A, is by structural induction on the derivation $\mathtt{e}, h \longrightarrow \mathtt{e}', h'$ or $P, h \longrightarrow P', h'$. It uses the Generation Lemmas, the Subderivations Lemma, and the Context Substitution Lemma, as well as further lemmas, stated and proven in the appendix, and which deal with properties of the relation $\preceq$, of the operations $\circ$, and $\|$, and substitutions.

7.2 Communication Safety

Even more interesting than subject reduction, are the following properties:

P1 (communication error freedom) no communication error can occur, *i.e.*, there cannot be two sends or two receives on the same channel in parallel in two different threads;

P2 (progress) typable threads can always progress unless one of the following situations occurs:
- a null pointer exception is thrown;
- there is a connect instruction waiting for the dual connect instruction.

P3 (communication-order preserving) after a session has begun the required communications are always executed in the expected order.

In order to state **P1**, we add a new constant $\mathsf{CommErr}$ (*communication error*) to the syntax and the following rule to the operational semantics:

$$E_1[\mathsf{e}]\,|\,E_2[\mathsf{e}'] \longrightarrow \mathsf{CommErr}$$

if e and e' are session expressions with the same subject and are not dual of each other, – dual expressions were defined on page 8. We can now prove that we never reach a state containing such incompatible expressions.

Corollary 7.7 ($\mathsf{CommErr}$ **Freedom)** *Assume* $\Gamma;\Sigma \vdash P;h$ *and* $P,\emptyset \longrightarrow\!\!\!\!\rightarrow P',h'$. *Then* P' *does not contain* $\mathsf{CommErr}$, *i.e., there does not exist* Q *such that* $P' \equiv Q\,|\,\mathsf{CommErr}$.

The proof of the above corollary follows from the fact that a communication error only happens if two threads contain, in evaluation positions, session expressions with the same subject, which are not dual of each other, and that the parallel concatenation of such threads is not well typed. The rest is straightforward from the subject reduction theorem.

7.3 Progress

This subsection states the main result of this paper – the progress property **P2** holds in our typing system. A summary of the proof is given here; the full proof is relegated to Appendix B.

Properties **P2** and **P3** hold for a thread obtained by reducing a well-typed (from an empty session environment) closed thread in which all expressions are user expressions.

We write $\prod_{0\leq \mathsf{i}<\mathsf{n}} \mathsf{e}_\mathsf{i}$ for $\mathsf{e}_0 \mid \mathsf{e}_1 \mid \ldots \mid \mathsf{e}_{\mathsf{n}-1}$. We say a thread P is Γ-*initial* if $\Gamma;\emptyset \vdash P : \mathsf{thread}$ is derivable and the domain of Γ contains only shared channels and $P \equiv \prod_{0\leq \mathsf{i}<\mathsf{n}} \mathsf{e}_\mathsf{i}$ where e_i is a user expression. We denote by h_Γ the heap which contains exactly the shared channels in the domain of Γ. Notice that P Γ-initial implies $\Gamma;\emptyset \vdash P;h_\Gamma$, *i.e.,* the heap h_Γ agrees with Γ-initial threads.

We start by formalising two crucial properties assured by our type system. The first property is the bilinearity of live channels.

Lemma 7.8 *Assume* P_0 *is* Γ*-initial and* $P_0,h_\Gamma \longrightarrow\!\!\!\!\rightarrow P,h$. *Then each live channel occurs exactly in two threads in* P.

The second property assures that sessions are not interleaved when rule **WeakB** is never applied. For stating this property contexts are handy. As usual we add the hole $[\,]$ to the syntax of expressions and we say that a context is an expression which contains one hole (notation $C[\,]$). Clearly evaluation contexts (as defined at page 9) are particular contexts.

Lemma 7.9 *If* $\mathsf{connect\ u\ s}\{\mathsf{e}\}$ *is an expression which is well typed without using rule* **WeakB** *and* $\mathsf{e} = C[\mathsf{e}']$*, where* e' *is a session expression or a method call with subject* u'*, then one of the following conditions holds:*

(1) $\mathsf{u} = \mathsf{u}'$*;*
(2) $C[\,] = C_1[\mathsf{connect\ u}'\mathsf{s}'\{C_2[\,]\}]$*;*
(3) $C[\,] = C_1[\mathsf{u}''.\mathsf{receiveS}\,(\mathsf{x})\{C_2[\,]\}]$ *and* $\mathsf{u}' = \mathsf{x}$.

Theorem 7.10 (Progress) *Assume* P_0 *is* Γ*-initial and* $P_0, h_\Gamma \longrightarrow\!\!\!\!\rightarrow P, h$*. Then one of the following holds.*

- *In P, all expressions are values, i.e.,* $P \equiv \prod_{0 \le \mathsf{i} < \mathsf{n}} \mathsf{v_i}$*;*
- $P, h \longrightarrow P', h'$*;*
- *P contains a null pointer exception, i.e.,* $P \equiv \mathsf{NullExc} \mid Q$*; or*
- *P stops with a connect waiting for its dual instruction, i.e.,* $P \equiv E[\mathsf{connect\ c\ s}\{\mathsf{e}\}] \mid Q$.

Proof Outline We show that execution of initial processes preserves the following properties

- each live channel occurs in exactly two threads;
- if e precedes e' in some thread (*i.e.,* e will be executed before e'), then all live channels in e are more recent than the live channels in e' (we assume that channels created at runtime have a “time stamp” and can be distinguished according to how recent they are).

We then argue that execution of initial processes leads to a configuration which is either a sequence a values, or contains a null pointer exception, or is waiting for a connect, or has at least one live channel. In the latter case, we chose the most recent one, and find the two threads in which the channel is live. Because execution preserves well typedness, we know that the channel has dual types in the two threads. Because of this, if the threads are session expressions, we can show that they can communicate. Otherwise, they can execute independently.

Appendix B contains the detailed proof.

Note that the Progress Theorem shows that *threads can always communicate at live channels*. From the above theorem, immediately we get:

Corollary 7.11 (Completion of Sessions) *Assume* P_0 *is* Γ*-initial and* $P_0, h_\Gamma \longrightarrow\!\!\!\!\rightarrow P, h$*. Suppose* $P \equiv \prod_{0 \le \mathsf{i} < \mathsf{n}} \mathsf{e_i}$ *and irreducible. Then either all* $\mathsf{e_i}$ *are values* $(0 \le \mathsf{i} < \mathsf{n})$ *or there is some* j $(0 \le \mathsf{j} < \mathsf{n})$ *such that* $\mathsf{e_j} \in \{\mathsf{NullExc}, E[\mathsf{connect\ c\ s}\{\mathsf{e}\}]\}$.

Finally we state the main property (**P3**) of our typing system. For this purpose, we define the partial order $\sqsubseteq$ on session types as follows.

Definition 7.12 (Evaluation Order) *$\theta \sqsubseteq \theta'$ is the smallest partial order such that:*

- *$\varepsilon \sqsubseteq \rho$;*
- *$\rho \sqsubseteq \pi.\rho$;*
- *$\pi_i \sqsubseteq \dagger\langle \pi_1, \pi_2 \rangle$ ($i \in \{1,2\}$);*
- *$\pi_i.\rho \sqsubseteq \dagger\langle \pi_1, \pi_2 \rangle.\rho$ ($i \in \{1,2\}$);*
- *$\eta_i \sqsubseteq \dagger\langle \eta_1, \eta_2 \rangle$ ($i \in \{1,2\}$);*
- *$\dagger\langle \pi.\langle\pi\rangle^*, \varepsilon \rangle \sqsubseteq \langle\pi\rangle^*$;*
- *$\dagger\langle \pi.\langle\pi\rangle^*, \varepsilon \rangle.\rho \sqsubseteq \langle\pi\rangle^*.\rho$;*
- *$\rho \sqsubseteq \rho'$ implies $\mathsf{begin}.\rho \sqsubseteq \mathsf{begin}.\rho'$.*

The partial order $\sqsubseteq$ takes into account reduction as formalised in the following theorem: any configuration $E[\mathsf{e}_0] \mid Q, h$ reachable from the initial configuration and containing the irreducible session expression e_0, if it proceeds, then either

(1) it does so in the sub-thread Q, or
(2) Q contains an expression e_0' (dual of e_0), which
  (a) interacts with e_0, and
  (b) has a dual type at c, and
  (c) then the type of channel c in the resulting process "correctly shrinks" as $\theta' \sqsubseteq \theta$.

Theorem 7.13 (Communication-Order Preservation) *Let P_0 be Γ-initial. Assume that $P_0, h_\Gamma \twoheadrightarrow E[\mathsf{e}_0] \mid Q, h \longrightarrow P', h'$ where e_0 is an irreducible session expression with subject c. Then:*

(1) $P' \equiv E[\mathsf{e}_0] \mid Q'$, or
(2) $Q \equiv E'[\mathsf{e}_0'] \mid R$ with e_0' dual of e_0 and
  (a) *$E[\mathsf{e}_0] \mid E'[\mathsf{e}_0'] \mid R, h \longrightarrow \mathsf{e} \mid \mathsf{e}' \mid R', h'$;*
  (b) *$\Gamma; \Sigma, \mathsf{c} : \theta \vdash E[\mathsf{e}_0] : \mathsf{thread}$ and $\Gamma; \Sigma', \mathsf{c} : \overline{\theta} \vdash E'[\mathsf{e}_0'] : \mathsf{thread}$; and*
  (c) *$\Gamma; \hat{\Sigma}, \mathsf{c} : \theta' \vdash \mathsf{e} : \mathsf{thread}$ and $\Gamma; \hat{\Sigma}', \mathsf{c} : \overline{\theta}' \vdash \mathsf{e}' : \mathsf{thread}$ with $\theta' \sqsubseteq \theta$.*

8 Inference of Session Environments, Hot Sets, and Session Types for connect

Although the type system is flexible enough to express interesting protocols, typing as described so far is somewhat inconvenient, in that it requires the hot sets and the session environments to be assumed (or "guessed"). In this section, we develop *inference rules* for expressions and threads which have the shape

$$\Gamma \vdash \mathsf{e} : \mathsf{t}\ [\!]\ \Sigma; \mathscr{S} \qquad \text{and} \qquad \Gamma \vdash P : \mathsf{thread}\ [\!]\ \Sigma$$

and which express that session environments and hot sets are derived rather than assumed.

Extension of session environment schemes

$$\Sigma(\!(\mathsf{u})\!) = \begin{cases} \Sigma(\mathsf{u}) & \text{if } \mathsf{u} \in \mathscr{D}(\Sigma), \\ \varepsilon & \text{otherwise.} \end{cases}$$

Ending of running session type schemes and of session environment schemes

$$\rho\downarrow = \begin{cases} \dagger\langle \pi_1\downarrow, \pi_2\downarrow\rangle, & \text{if } \rho = \dagger\langle \pi_1, \pi_2\rangle \text{ for some } \pi_1, \pi_2, \\ \pi.\dagger\langle \pi_1\downarrow, \pi_2\downarrow\rangle, & \text{if } \rho = \pi.\dagger\langle \pi_1, \pi_2\rangle \text{ for some } \pi, \pi_1, \pi_2, \\ \rho, & \text{if } \rho \text{ is an ended session type scheme,} \\ \rho.\mathsf{end} & \text{otherwise.} \end{cases}$$

$$\Sigma\downarrow = \{\mathsf{u} : \Sigma(\mathsf{u})\downarrow \ | \mathsf{u} \in \mathscr{D}(\Sigma)\}$$

Union of hot sets

$$\mathscr{S}_1 \uplus \mathscr{S}_2 = \begin{cases} \mathscr{S}_1 \cup \mathscr{S}_2 & \text{if either } \mathscr{S}_1 = \mathscr{S}_2 \text{ or } \mathscr{S}_1 = \emptyset \text{ or } \mathscr{S}_2 = \emptyset, \\ \bot & \text{otherwise.} \end{cases}$$

Fig. 17. Auxiliary Operators for Inference

For simplicity we only consider typing of initial threads, and therefore we do not allow to use rule **WeakB**. This is enough, since by the Subject Reduction Theorem we know that all threads obtained by reducing well-typed threads are well typed too.

We extend the syntax of types with the *standard type variables*, ranged over by ϕ, which stand for standard types, and the *partial session type variables*, ranged over by ψ, which stand for partial session types. In this way, for each one of the syntactic categories in Fig. 8, we obtain a corresponding category of *schemes,* and similarly for the environments. We use for them the same notational conventions. Notice that, since we do not allow to use rule **WeakB**, all predicates in session environment schemes for typing expressions are running session type schemes.

Fig. 17 gives some auxiliary operators on session environment schemes and hot sets. The *ending operator*, $\downarrow$, appends if meaningful end to running session type schemes, propagates inside the final branches of conditional partial session types, and does nothing otherwise. The ending operator generalises to session environment schemes in the expected way (see Fig. 17).

The more interesting inference rules for the expressions and threads occupy Fig. 18. Other rules are left to Appendix C. The rules are applicable only if all sets in the conclusion are defined.

MethMinusI

$$\frac{\Gamma\vdash \mathsf{e}:C\;[\,]\;\Sigma_0;\mathscr{S}_0\quad \Gamma\vdash \mathsf{e}_i:\mathsf{t}_i\;[\,]\;\Sigma_i;\mathscr{S}_i\quad i\in\{1\dots n\}\qquad \mathsf{mtype}(\mathsf{m},C)=\mathsf{t}_1,\dots,\mathsf{t}_n,\rho_1,\dots,\rho_m\overset{\ominus}{\rightarrow}\mathsf{t}}{\Gamma\vdash \mathsf{e.m}(\mathsf{e}_1,\dots,\mathsf{e}_n,\mathsf{u}_1,\dots,\mathsf{u}_m):\mathsf{t}\;[\,]\;\Sigma_0\circ\Sigma_1\dots\circ\Sigma_n\circ\{\mathsf{u}_1:\rho_1,\dots,\mathsf{u}_m:\rho_m\};\mathscr{S}_0\uplus\mathscr{S}_1\uplus\dots\uplus\mathscr{S}_n}$$

MethPlusI

$$\frac{\Gamma\vdash \mathsf{e}:C\;[\,]\;\Sigma_0;\mathscr{S}_0\quad \Sigma_0\subseteq\{\mathsf{u}_1\}\quad \Gamma\vdash \mathsf{e}_i:\mathsf{t}_i\;[\,]\;\Sigma_i;\mathscr{S}_i\quad \mathscr{S}_i\subseteq\{\mathsf{u}_1\}\quad i\in\{1\dots n\}\qquad \mathsf{mtype}(\mathsf{m},C)=\mathsf{t}_1,\dots,\mathsf{t}_n,\rho_1,\dots,\rho_m\overset{\oplus}{\rightarrow}\mathsf{t}}{\Gamma\vdash \mathsf{e.m}(\mathsf{e}_1,\dots,\mathsf{e}_n,\mathsf{u}_1,\dots,\mathsf{u}_m):\mathsf{t}\;[\,]\;\Sigma_0\circ\Sigma_1\dots\circ\Sigma_n\circ\{\mathsf{u}_1:\rho_1,\dots,\mathsf{u}_m:\rho_m\};\{\mathsf{u}_1\}}$$

ConnI

$$\frac{\Gamma\backslash\mathsf{u}\vdash \mathsf{e}:\mathsf{t}\;[\,]\;\Sigma;\mathscr{S}\quad \Sigma(\!(\mathsf{u})\!)=\rho\quad \mathsf{s}=\mathsf{begin}.\sigma(\rho\downarrow)\quad \mathscr{S}\subseteq\{\mathsf{u}\}\quad \begin{array}{l}\text{if } \mathsf{u} \text{ is a variable } \Gamma(\mathsf{u})<:\mathsf{s}\\ \text{if } \mathsf{u} \text{ is a name } \Gamma(\mathsf{u})=\mathsf{sch}\end{array}}{\Gamma\vdash \mathsf{connect\ u\ s}\ \{\mathsf{e}\}:\sigma(\mathsf{t})\;[\,]\;\sigma(\Sigma)\setminus\mathsf{u};\emptyset}$$

SendI

$$\frac{\Gamma\vdash \mathsf{e}:\mathsf{t}\;[\,]\;\Sigma;\mathscr{S}\quad \mathscr{S}\subseteq\{\mathsf{u}\}}{\Gamma\vdash \mathsf{u.send}(\ \mathsf{e}\ ):\mathit{Object}\;[\,]\;\Sigma\circ\{\mathsf{u}:!\mathsf{t}\};\{\mathsf{u}\}}$$

ReceiveI

$$\frac{\Gamma\vdash \mathsf{ok}}{\Gamma\vdash \mathsf{u.receive}:\phi\;[\,]\;\{\mathsf{u}:?\phi\};\{\mathsf{u}\}}$$

SendSI

$$\frac{\Gamma\vdash \mathsf{ok}}{\Gamma\vdash \mathsf{u.sendS}(\mathsf{u}'):\ \mathit{Object}\;[\,]\;\{\mathsf{u}':\psi.\mathsf{end},\mathsf{u}:!(\psi.\mathsf{end})\};\{\mathsf{u}\}}$$

ReceiveSI

$$\frac{\Gamma\backslash\mathsf{x}\vdash \mathsf{e}:\mathsf{t}\;[\,]\;\{\mathsf{x}:\rho\};\mathscr{S}\quad \mathscr{S}\subseteq\{\mathsf{x}\}\quad \rho\neq\varepsilon}{\Gamma\vdash \mathsf{u.receiveS}(\mathsf{x})\{\mathsf{e}\}:\mathit{Object}\;[\,]\;\{\mathsf{u}:?(\rho\downarrow)\};\{\mathsf{u}\}}$$

SendIfI

$$\frac{\begin{array}{c}\Gamma\vdash \mathsf{e}:\mathsf{t}_0\;[\,]\;\Sigma_0;\mathscr{S}_0\quad \Gamma\vdash \mathsf{e}_i:\mathsf{t}_i\;[\,]\;\Sigma_i;\mathscr{S}_i\quad \Sigma_i(\!(\mathsf{u})\!)=\rho_i\quad \mathscr{S}_j\subseteq\{\mathsf{u}\}\quad i\in\{1,2\}\quad j\in\{0,1,2\}\\ \sigma=\mathsf{E}(\{\langle\mathsf{t}_1;\mathsf{t}_2\rangle,\langle\mathsf{t}_0;\mathsf{bool}\rangle\}\cup\{\langle\Sigma_1(\mathsf{u}');\Sigma_2(\mathsf{u}')\rangle\mid\forall\mathsf{u}'\neq\mathsf{u}.\mathsf{u}'\in\mathscr{D}(\Sigma_1)\cap\mathscr{D}(\Sigma_2)\}\cup\\ \{\langle\Sigma_i(\mathsf{u}');\varepsilon.\mathsf{end}\rangle\mid\forall\mathsf{u}'\neq\mathsf{u}.\mathsf{u}'\in\mathscr{D}(\Sigma_i)\ \&\ \mathsf{u}'\notin\mathscr{D}(\Sigma_j)\ i,j\in\{1,2\}\})\\ \rho_i'=\begin{cases}\rho_i\downarrow & \text{if } \rho_j \text{ is an ended session type scheme,}\\ \rho_i & \text{otherwise}\end{cases}\quad i\neq j,\ i,j\in\{1,2\}\\ \Sigma=\{\mathsf{u}':\varepsilon.\mathsf{end}\mid\forall\mathsf{u}'\neq\mathsf{u}.\mathsf{u}'\in\mathscr{D}(\Sigma_2)\ \&\ \mathsf{u}'\notin\mathscr{D}(\Sigma_1)\}\end{array}}{\Gamma\vdash \mathsf{u.sendIf}(\mathsf{e})\{\mathsf{e}_1\}\{\mathsf{e}_2\}:\sigma(\mathsf{t})\;[\,]\;\sigma(\Sigma_0)\circ(\sigma(\Sigma_1\setminus\mathsf{u},\mathsf{u}:!\langle\rho_1',\rho_2'\rangle)\cup\Sigma);\{\mathsf{u}\}}$$

Fig. 18. Selected Inference Rules for Expressions

As usual, the inference rules are structural, *i.e.*, depend on the structure of the expression being typed; typically, the inference system does not have rules like **Weak**. Therefore, the inference rules must play also the role of the non-structural typing rules.

Rule **MethMinusI** uses the union of hot sets to check if all hot sets are either the

StartI

$$\frac{\Gamma \vdash \mathsf{e} : \mathsf{t} \;[]\; \Sigma; \mathscr{S}}{\Gamma \vdash \mathsf{e} : \mathsf{thread} \;[]\; \Sigma}$$

ParI

$$\frac{\Gamma \vdash P : \mathsf{thread} \;[]\; \Sigma \quad \Gamma \vdash P' : \mathsf{thread} \;[]\; \Sigma'}{\Gamma \vdash P \,|\, P' : \mathsf{thread} \;[]\; \Sigma \,|||\, \Sigma'}$$

Fig. 19. Inference Rules for Threads

same or empty. In rule **MethPlusI** all hot sets of the premises must be either empty or just contain the running channel u_1.

Rule **ReceiveI** introduces a standard type variable, since we do not know the type of the data that will be received. Rule **SendSI** introduces a partial session type variable, since we do not know the type of the channel that will be sent. We always assume the introduced variables to be fresh, *i.e.*, they cannot occur elsewhere in the current deduction.

In rule **ConnI** we do not know if the session environment inferred for e contains a premise for u, for this reason we use the extension of session environment schemes defined in Fig. 17.

An *inference substitution*, σ, maps standard type variables to standard type schemes, and partial session type variables to partial session type schemes. We use an inference substitution in rule **ConnI** in order to unify the shared session type s with $\mathsf{begin}.\rho\downarrow$, where $\rho\downarrow$ being inferred may contain variables. That is, we require $\mathsf{s} = \mathsf{begin}.\sigma(\rho\downarrow)$. We prescribe the domain of σ to be the set of type variables which occur in ρ.

We need some definitions for combining session environment schemes.

Given a finite set of pairs of standard type schemes and running session type schemes $\Xi = \{\langle \mathsf{t}_i; \mathsf{t}'_i\rangle \mid 1 \leq i \leq m\} \cup \{\langle \rho_j; \rho'_j\rangle \mid 1 \leq j \leq n\}$, an *equality solver* of Ξ is an inference substitution σ such that we have $\sigma(\mathsf{t}_i) = \sigma(\mathsf{t}'_i)$ for $1 \leq i \leq m$ and either $\sigma(\rho_j) = \sigma(\rho'_j)$, $\sigma(\rho_j\downarrow) = \sigma(\rho'_j)$ or $\sigma(\rho_j) = \sigma(\rho'_j\downarrow)$ for $1 \leq j \leq n$. The *most general equality solver* of Ξ, $\mathsf{E}(\Xi)$, is the solver σ such that if σ' is a solver of Ξ, then there is an inference substitution σ'' such that $\sigma = \sigma' \circ \sigma''$. It is routine to show that if a set has a solver, then it also has the more general one. For example the most general equality solver σ of $\{\langle \phi; \mathsf{bool}\rangle, \langle ?\phi.\mathsf{end}; \psi\rangle\}$ is defined by $\sigma(\phi) = \mathsf{bool}$ and $\sigma(\psi) = ?\mathsf{bool}$, while there is no solver for $\{\langle \phi; \mathsf{bool}\rangle, \langle ?\phi.\mathsf{end}; ?\mathsf{int}\rangle\}$. The most general equality solver is used in rules **SendIfI** (and **ReceiveIfI** which is left to Appendix C) in order to unify the types of the two branches.

Given a finite set of pairs of running session type schemes $\Xi = \{\langle \rho_i; \rho'_i\rangle \mid 1 \leq i \leq n\}$, a *duality solver* of Ξ is an inference substitution σ such that for $1 \leq i \leq n$, we have either $\sigma(\rho_i) = \overline{\sigma(\rho'_i)}$, $\sigma(\rho_i\downarrow) = \overline{\sigma(\rho'_i)}$, or $\sigma(\rho_i) = \overline{\sigma(\rho'_i\downarrow)}$. The *most general*

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset \vdash 5 : \mathsf{int} \;[]\; \emptyset;\emptyset}}{\emptyset \vdash \mathsf{x.send}\,(5) : \mathit{Object} \;[]\; \{\mathsf{x} : !\mathsf{int}\};\{\mathsf{x}\}}}{\emptyset \vdash \mathsf{c}_2.\mathsf{receiveS}\,(\mathsf{x})\{\mathsf{x.send}\,(5)\} : \mathit{Object} \;[]\; \{\mathsf{c}_2 : ?(!\mathsf{int.end})\};\{\mathsf{c}_2\}}}{\Gamma_1 \vdash \mathsf{e} : \mathit{Object} \;[]\; \emptyset;\emptyset} \qquad \dfrac{}{\Gamma_1 \vdash \mathsf{e}' : \phi \;[]\; \{\mathsf{c}_1 : ?\phi\};\{\mathsf{c}_1\}}$$

$$\dfrac{\dfrac{\Gamma_1 \vdash \mathsf{e} : \mathit{Object} \;[]\; \emptyset;\emptyset \qquad \Gamma_1 \vdash \mathsf{e}' : \phi \;[]\; \{\mathsf{c}_1 : ?\phi\};\{\mathsf{c}_1\}}{\Gamma_1 \vdash \mathsf{e};\mathsf{e}' : \phi \;[]\; \{\mathsf{c}_1 : ?\phi\};\{\mathsf{c}_1\}}}{\Gamma \vdash \mathsf{connect}\ \mathsf{c}_1\,\mathsf{begin}.?\mathsf{int}.\mathsf{end}\{\mathsf{e};\mathsf{e}'\} : \mathsf{int} \;[]\; \emptyset;\emptyset}$$

where $\mathsf{e} = \mathsf{connect}\ \mathsf{c}_2\,\mathsf{begin}.?(!\mathsf{int}.\mathsf{end}).\mathsf{end}\{\mathsf{c}_2.\mathsf{receiveS}\,(\mathsf{x})\{\mathsf{x.send}\,(5)\}\}$,
$\mathsf{e}' = \mathsf{c}_1.\mathsf{receive}$, $\Gamma = \{\mathsf{c}_1 : \mathsf{sch}, \mathsf{c}_2 : \mathsf{sch}\}$, $\Gamma_1 = \{\mathsf{c}_2 : \mathsf{sch}\}$.

Fig. 20. An Example of Inference

duality solver $\mathtt{D}(\Xi)$ is defined similarly to the most general equality solver. For example the most general duality solver σ of $\{\langle ?\phi; !\mathsf{bool}\rangle, \langle !\phi.\mathsf{end}; \psi\rangle\}$ is defined by $\sigma(\phi) = \mathsf{bool}$ and $\sigma(\psi) = ?\mathsf{bool}$.

We use the most general duality solver to define the *parallel composition of session environment schemes* as:

$$\Sigma ||| \Sigma' = \sigma(\Sigma \setminus \mathscr{D}(\Sigma)) \cup \sigma(\Sigma' \setminus \mathscr{D}(\Sigma)) \cup \{\mathsf{u} : \updownarrow \mid \mathsf{u} \in \mathscr{D}(\Sigma) \cap \mathscr{D}(\Sigma')\}$$

where $\sigma = \mathtt{D}(\{\langle \Sigma(\mathsf{u}); \Sigma'(\mathsf{u})\rangle \mid \mathsf{u} \in \mathscr{D}(\Sigma) \cap \mathscr{D}(\Sigma')\})$. Therefore $\Sigma ||| \Sigma'$ is undefined if there is no duality solver of $\{\langle \Sigma(\mathsf{u}); \Sigma'(\mathsf{u})\rangle \mid \mathsf{u} \in \mathscr{D}(\Sigma) \cap \mathscr{D}(\Sigma')\}$. We use the parallel composition of session environment schemes in rule **ParI**. Notice that by construction in the premises of this rule the set of type variables which occur in Σ and in Σ' are disjoint.

Note that the inference of Σ does not rely on $\mathscr{S}$, so that we can obtain the same result for the system without $\mathscr{S}$.

As an example we show the inference for the thread P_1 of Example 4.4 in Fig. 20.

We can show that inference computes the least session environments and hot sets, as stated in the following theorem, whose proof is given in Appendix C. We first need to introduce an order on session types and on session environments which takes into account the absence of weakening rules in the type inference. This order is a restriction of the weakening order of Definition 7.1.

Definition 8.1 (Inference Order)

(1) $\theta \Subset \theta'$ *is the smallest partial order such that*
- $\pi \Subset \pi.\mathsf{end}$*, and*
- $\varepsilon.\mathsf{end} \Subset \updownarrow$.

(2) $\Sigma \Subset \Sigma'$ *iff* $\forall \mathsf{u} \in \mathscr{D}(\Sigma) : \Sigma(\mathsf{u}) \Subset \Sigma'(\mathsf{u})$*, and* $\forall \mathsf{u} \in \mathscr{D}(\Sigma') \setminus \mathscr{D}(\Sigma) :\ \Sigma'(\mathsf{u}) \Subset \updownarrow$.

Clearly $\Sigma \Subset \Sigma'$ implies $\Sigma \preceq \Sigma'$, but the vice versa is not true. For example $\{\mathsf{c} : ?\mathsf{bool}\} \Subset \{\mathsf{c} :?\mathsf{bool}.\mathsf{end}, \mathsf{c}' :\updownarrow\}$, but $\{\mathsf{c} :?\mathsf{bool}\} \not\Subset \{\mathsf{c} : \mathsf{begin}.?\mathsf{bool}, \mathsf{c}' :\updownarrow\}$, also if $\{\mathsf{c} : ?\mathsf{bool}\}; \{\mathsf{c}\} \preceq \{\mathsf{c} : \mathsf{begin}.?\mathsf{bool}, \mathsf{c}' :\updownarrow\}; \emptyset$.

Theorem 8.2 *(1) If $\Gamma; \Sigma; \mathscr{S} \vdash \mathsf{e} : \mathsf{t}$ without using rule* **WeakB**, *then $\Gamma \vdash \mathsf{e} : \mathsf{t}' \; [] \; \Sigma'; \mathscr{S}'$ where $\sigma(\mathsf{t}') = \mathsf{t}$ and $\sigma(\Sigma') \Subset \Sigma$ for some inference substitution σ and $\mathscr{S}' \subseteq \mathscr{S}$.*

(2) If $\Gamma \vdash \mathsf{e} : \mathsf{t} \; [] \; \Sigma; \mathscr{S}$, then for all inference substitutions σ such that $\sigma(\Sigma)$ is a session environment and $\sigma(\mathsf{t})$ is a type, we get: $\Gamma; \sigma(\Sigma); \mathscr{S} \vdash \mathsf{e} : \sigma(\mathsf{t})$.

(3) If $\Gamma; \Sigma \vdash P : \mathsf{thread}$ without using rule **WeakB**, *then $\Gamma \vdash P : \mathsf{thread} \; [] \; \Sigma'$ where $\sigma(\Sigma') \Subset \Sigma$ for some inference substitution σ.*

(4) If $\Gamma \vdash P : \mathsf{thread} \; [] \; \Sigma$, then for all inference substitutions σ such that $\sigma(\Sigma)$ is a session environment, we get: $\Gamma; \sigma(\Sigma) \vdash P : \mathsf{thread}$.

Note that the above theorem assures that the present type system enjoys the principal type property in the classical sense of [31].

9 Related work

Systems for processes, Subject Reduction and Progress

Session types for the π-calculus are the subject of many works [6, 7, 11–13, 18, 27, 28, 32, 34, 35, 48]. More recently, sessions were incorporated into boxed ambients [26], and higher-order processes supporting code mobility [43].

In all previously mentioned papers on session types, typability guarantees the absence of run-time communication errors. However, not all of them have the subject reduction property: the problem emerges when sending and instantiating a live channel to a thread which already uses this channel to communicate, as in Example 4.4. This example can be translated into the calculi studied in [6, 28, 34, 51], and this issue has been discussed with some of the authors of these papers [1]. The recent work [54] analyses this issue in detail, comparing different reduction rules and typing systems appeared in the literature [6, 28, 34, 51].

MOOSE has been inspired by the previously mentioned papers, however, we believe that it has been the first calculus which guarantees absence of starvation on live channels also in presence of delegation (progress without delegation is only considered in [21]). For example, we can encode the counterpart of Example 5.4 in the calculi of [6, 28, 34, 51]. In the language of [51] we can type the parallel composition of the following processes (obtained by translating the threads of Example 5.4):

```
1 // fun1 x y =
2 let u = request x in
3  let w = request y in
4   let i = receive u in
5    let j = receive w in
6     close u; close w;
```

```
1 // fun2 x y =
2 let u = accept x in
3  let w = accept y in
4   send 5 on w;
5   send 6 on u;
6   close u; close w;
```

Note that in the above two interleaved sessions are established, however no session can proceed, because the progress of each is dependent on the progress of the other: before line 4 of the left hand process can reduce, line 5 of the right hand process must be made available in parallel; a similar dependency occurs between lines 4 and 5 of the right and left hand processes, respectively. Furthermore, observe that such deadlocks can also occur due to interdependencies among three or more processes, in which case they cannot be detected easily. We believe that such configurations are clearly undesirable, and for this reason our typing system rejects interleaved sessions.

The same problem arises in the calculi of [6, 28, 34], where the previous example is written as follows:

```
request x(u) in
 request y(w) in
   u?(i);
     w?(j );
```

```
accept x(u)
 accept y(w)
  w![5];
   u![6];
```

Note that by simply dropping the hot set, we can flexibly obtain a version of the typing system which preserves the type safety and type inference results, but allows deadlock on live channels like the above mentioned literature. In this sense, our system is not only theoretically sound, but also modular.

Clearly, allowing asynchronous communication enhances progress: for example the processes of Example 5.4 would not be stuck any more. Session types which take advantage of asynchronous communication are studied in [15] for a suitable variant of MOOSE. There the conditions for progress are relaxed, allowing arbitrary (non-blocking) outputs to appear inside nested sessions.

In recent work by some of the present authors [18], more flexible conditions for progress are studied in the context of a process language. In their system, interleaving is allowed by permitting hot sets to contain more than one element, and progress is ensured using a causality partial order, resulting in a significantly more fine-grained analysis. For instance the translation of Example 5.3 is typable in the type discipline of [18].

Advanced session types

An issue that arises with the use of sessions is how to group and distinguish different behaviours within a program or protocol. In [34] and subsequently in [29] the authors utilise labelled *branching* and *selection*; the first enables a process to offer alternative session parts indexed by labels, and the second is used dually to choose a part by selecting one of the available labels. In [27, 28, 34, 50], branching and selection are considered as an effective way to simulate methods of objects. Our conditional constructs are a simplification of branching and selection, therefore the same behaviour realised by branching types can also be expressed using our types. A different branching mechanism is proposed in [19, 23], where the choice on how to continue a session is made on the basis of the class of the object sent/received.

Session subtyping systems range from simple session subtyping [28] to more complex bounded session polymorphism [27], which enables parametric polymorphism of session types. Inspired by [27], [19] enhances the expressivity of session types in objects, by allowing bounded polymorphism for a suitable extension of MOOSE.

As another study on the enrichment of basic session types, in [6] the authors integrate the *correspondence assertions* of [30] with standard session types to reason about multi-party protocols comprising of standard interleaved sessions.

In this work, our purpose was to produce a reliable and extensible object-oriented core, and not to include everything in the first attempt; however, such richer type structures are attractive in an object-oriented framework. MOOSE can be used as a core extensible language incorporating other typing systems.

Linear typing systems

Session types for the π-calculus relate to linear typing systems [33, 39], whose main aim is to guarantee that a channel is used exactly or at most once within a term.

In the context of programming languages, [25] proposes a type system for checking protocols and resource usage in order to enforce linearity of variables in the presence of aliasing. They implemented the typing system in Vault [17], a low level C-like language. The main issue that they had to address is that a shared component should not refer to linear components, since aliasing of the shared component can result in non-linear usage of any linear elements to which it provides access. To relax this condition, they proposed operations for safe sharing, and for controlled linear usage. In our system non-interference is ensured by operational semantics in which substitution of shared with fresh channels takes place when reducing connect , and therefore we do not need explicit constructs for this purpose. Finally, note that the system of [25] is not readily applicable in a concurrent setting, and hence in channel-based communication.

Programming languages and sessions

In [51] the authors define a concurrent functional language with session primitives. Their language supports sending of channels and higher-order values that do not contain running sessions, and incorporates branching and selection, along with recursive sessions and channel sharing. Moreover, it incorporates the multi-threading primitive fork, whose operational semantics is similar to that of spawn. Finally, their system allows live channels as parameters to functions, and tracks aliasing of channels; as a result, their system is polymorphic.

In [49], the authors formalise an extension to CORBA interfaces based on session types, which are used to determine the order in which available operations can be invoked. The authors define *protocols* consisting of *sessions*, and use labelled branches and selection to model method invocation within each session. Labelled branches are also used to denote exceptions, and their system incorporates recursive session types. However, run-time checks are considered in order to check protocol conformance, and there is no formalisation in terms of operational semantics and type system.

More recently, a similar approach has been used in the Singularity OS [24]. Behaviour in this system is defined in *contracts*, that contain definitions that form a state machine of desired message exchange patterns. Messages encapsulate asynchronous method invocation, and consist of information on which method should be invoked, along with the actual arguments to use, when the message is received. Messages are exchanged using bidirectional channels, where each channel has two explicit endpoints. At the endpoints, the specific methods required for each state of the contract are defined. Asynchronous transmission is implemented using message queues at each endpoint. In our system, channels have generic send and receive operations, and communication is synchronous. Their system has the property that each endpoint can only be used by a single thread at a time, which corresponds to our property of bilinearity, and this ensures that messages at the endpoint queues are always ordered. Also, they allow to send channel endpoints, which corresponds to live channel communication in our system. When different messages can be received, they use a form of switch to group the program behaviours for each case, similarly to [49]. However, in contrast to the latter, contracts are verified statically.

We developed our formalism building upon previous experience with $\mathscr{L}_{doos}$ [21], a distributed object-oriented language with basic session capabilities. In the present work we have chosen to simplify the substrate to that of a concurrent calculus, and focus on the integration of advanced session types. In [21], as in all previous papers on session types, shared channels could only be associated with a single session type each, and therefore runtime checks were not required for connections; however, this assumption is not necessary, and it is orthogonal to the essence of our system which is the typing of a session body against a session type. In particular, in an open environment we cannot assume that the types of shared channels can be fixed in advance, and the runtime cost of checking compatibility is low, requiring one check of session duality (and possibly subtype checking).

In our new formulation we chose not to model RMI, and in fact, an interesting question is whether we can encode RMI as a form of degenerate session in the spirit of [49]. Also, we have now introduced more powerful primitives for thread and (shared) channel creation, along with the ability to delegate live sessions via method invocation and higher-order sessions. None of these features are considered in [21]. We discovered a flaw in the progress theorem in $\mathscr{L}_{doos}$ [21], and developed the new type system with hot sets in order to guard against the offending configurations.

More recently, [23] suggests an amalgamation of the session type and the object oriented paradigm whereby sessions are amalgamated with methods: class definitions contain therefore fields and session/method declarations. Generic classes and union types for the calculus of [23] are discussed in [10] and [4], respectively.

Behavioural types and Service-oriented computing

Behavioural types for processes (see [38]) have some similarities with sessions, but describe communications as types that resemble CCS processes. Hence, these systems capture the precise interleavings, and using additional tags (annotations) they achieve a fine-grained analysis of deadlock and liveness. Compared to our progress guarantee, the behavioural analysis is more detailed, but it is not straightforward how to adapt such techniques compositionally in a class-based object language without losing the appeal of being sufficiently simple for practical implementations.

In [2] a process language for service oriented computing is formalised, using a system of types similar, but simpler, to the behavioural types of [38]. Their system ensures a progress property for service clients, which seems natural since their sessions take place in nested scopes, and are not interleaved.

Objects implementing services are studied in [9] in an object-based formalism where communication is realised as a form of remote method invocation. Their system uses a language of spatial-behavioural types that can express sequencing and parallelism of usages on objects, recursive behaviours, and dynamic capabilities through owned types.

A different approach to the description of communication protocols is based on the notion of contract [8, 14]. The theory of contracts formalises the compatibility of a client to a service and the safe replacement of a service by another service by using behavioural equivalencies. An interesting comparison between contracts and session types is developed in [40].

Implementations

An early implementation of session types in Haskell is that of [44] where session types are mapped to existing type constructs, which can therefore be implemented without extending the language.

More recently, in [16], session types are considered for $F\sharp$, an implementation of a ML dialect. The work describes a system for ensuring security of multi-role sessions in the absence of trust. Session types are compiled to cryptographic protocols in a way such that during execution every party is guaranteed to play their role. Runtime verification is used to detect behaviour incompatible with a session.

The most relevant implementation is that of [36]. In this work the language Java is extended with basic session primitives for creating session-typed sockets and for performing communications governed by sessions based on our work [20], and also [19] and [15]. Sessions are defined by means of “protocol” declarations and static type checking is used to ensure safety, in combination with a dynamic agreement of session types between parties that are connecting over a session-typed socket. At the level of types our conditional types are replaced with the more general label-indexed branching and selection types found in the literature (see [34]), and the implementation also supports our session iteration types, session delegation, and subtyping. Communication is asynchronous and the implementation has been measured to have a very small performance overhead compared to untyped socket communication.

10 Conclusion and Future Work

This paper proposes the language MOOSE, a simple multi-threaded object-oriented language augmented with session communication primitives and types. MOOSE provides a clean object-oriented programming style for structural interaction protocols by prescribing channel usages as session types. We develop a typing system for MOOSE and prove type safety with respect to the operational semantics. We also show that in a well-typed MOOSE program, there will never be a communication error, starvation on live channels, nor an incorrect completion between two party interactions. These results demonstrate that a consistent integration of object-oriented language features and session types is possible where well typedness can guarantee the consistent composition of communication protocols. To our best knowledge, MOOSE is the first application of session types to a concurrent object-oriented class-based programming language, apart from [21]. Furthermore, type inference of session environments (Theorem 8.2), and the progress property on live channels with delegation (Theorem 7.10) have never been proved before in any work on session types including those in the π-calculus.

Exceptions and timeout

One feature not considered in our system, although important in practice, is exceptions; in particular, we did not provide any way for a session type to declare that it may throw a *checked* exception, so that when this occurs both communicating processes can execute predefined error-handling code. One obvious way to encode an exception would be to use a branch as in [49]. In addition, when a thread becomes blocked waiting for a session to commence, in our operational semantics, it will never escape the waiting state unless a connection occurs. In practice, this is unrealistic, but it could have been ameliorated by introducing a 'timeout' version of our basic connection primitive such as `connect(timeout)u s {e}`. However, controlling exceptions during session communication and realising timeout would be non-trivial since we wish to preserve the progress property on live channels. Therefore we plan to investigate these issues.

Acknowlegments Eduardo Bonelli, Adriana Compagnoni, Kohei Honda, Simon Gay, Pablo Garralda, Elsa Gunter, Antonio Ravara and Vasco Vasconcelos, discussed with us subject reduction and progress for systems with sessions types. Vasco Vasconcelos and the ECOOP reviewers gave useful suggestions. Alexander Ahern contributed to the initial design part which forms a basis of MOOSE. Mario Coppo pointed out a mistake in a previous progress proof and suggested a way out. Discussions with Marco Carbone, Kohei Honda, Robin Milner and the members of W3C Web Services Choreography Working Group for their collaboration on [13, 52] motivated the example in Section 2. Finally, we wish to thank the anonymous reviewers of the present submission for their feedback, which has lead to a strongly improved version. The work is partially supported by EPSRC GR/T03215, EPSRC GR/T03208, EPSRC EP/F003757 and IST2005-015905 MOBIUS.

References

[1] Personal Communication by E-mails between the authors of [6, 28, 34, 51].

[2] Lucia Acciai and Michele Boreale. A Type System for Client Progress in a Service-Oriented Calculus. In Pierpaolo Degano, Rocco De Nicola, and José Meseguer, editors, *Concurrency, Graphs and Models*, volume 5065 of *LNCS*, pages 642–658. Springer-Verlag, 2008.

[3] Alexander Ahern and Nobuko Yoshida. Formalising Java RMI with Explicit Code Mobility. In Ralph Johnson and Richard P. Gabriel, editors, *OOPSLA '05*, pages 403–422. ACM Press, 2005.

[4] Lorenzo Bettini, Sara Capecchi, Mariangiola Dezani-Ciancaglini, Elena Giachino, and Betti Venneri. Session and Union Types for Object Oriented Programming.

In Pierpaolo Degano, Rocco De Nicola, and José Meseguer, editors, *Concurrency, Graphs and Models*, volume 5065 of *LNCS*, pages 659–680. Springer-Verlag, 2008.

[5] Gavin Bierman, Matthew Parkinson, and Andrew Pitts. MJ: An Imperative Core Calculus for Java and Java with Effects. Technical Report 563, Univ. of Cambridge Computer Laboratory, 2003.

[6] Eduardo Bonelli, Adriana Compagnoni, and Elsa Gunter. Correspondence Assertions for Process Synchronization in Concurrent Communications. *Journal of Functional Programming*, 15(2):219–248, 2005.

[7] Eduardo Bonelli, Adriana Compagnoni, and Elsa Gunter. Typechecking Safe Process Synchronization. In Julian Rathke, editor, *FGUC'04*, volume 138 of *ENTCS*, pages 3–22. Elsevier, 2005.

[8] Mario Bravetti and Gianluigi Zavattaro. A Theory for Strong Service Compliance. In Amy L. Murphy and Jan Vitek, editors, *COORDINATION'07*, volume 4467 of *LNCS*, pages 96–112. Springer-Verlag, 2007.

[9] Luís Caires. Spatial-Behavioral Types, Distributed Services, and Resources. In Ugo Montanari, Donald Sannella, and Roberto Bruni, editors, *TGC'06*, volume 4661 of *LNCS*, pages 98–115. Springer-Verlag, 2006.

[10] Sara Capecchi, Mario Coppo, Mariangiola Dezani-Ciancaglini, Sophia Drossopoulou, and Elena Giachino. Amalgamating Sessions and Methods in Object Oriented Languages with Generics. *Theorical Computer Science*, 2008. To Appear.

[11] Marco Carbone, Kohei Honda, and Nobuko Yoshida. A Calculus of Global Interaction Based on Session Types. In Jean-Pierre Jouannaud and Ian Mackie, editors, *DMC'06*, volume 171 of *ENTCS*, pages 127–151. Elsevier, 2007.

[12] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured Communication-Centred Programming for Web Services. In Rocco De Nicola, editor, *ESOP'07*, volume 4421 of *LNCS*, pages 2–17. Springer-Verlag, 2007.

[13] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Theoretical Aspects of Communication-Centred Programming. In Catuscia Palamidessi and Frank D. Valencia, editors, *LIX'06*, volume 209 of *ETNCS*, pages 125–133. Elsevier, 2008.

[14] Giuseppe Castagna, Neil Gesbert, and Luca Padovani. A Theory of Contracts for Web Services. In Philip Wadler, editor, *POPL'08*, pages 261–272. ACM Press, 2008.

[15] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Asynchronous Session Types and Progress for Object-Oriented Languages. In Marcello Bonsangue and Einar Broch Johnsen, editors, *FMOODS'07*, volume 4468 of *LNCS*, pages 1–31. Springer-Verlag, 2007.

[16] Ricardo Corin, Pierre-Malo Deniélou, Cédric Fournet, Karthikeyan Bhargavan, and James Leifer. Secure Implementations for Typed Session Abstractions. In Riccardo Focardi, editor, *CSF'07*, pages 170–186. IEEE Computer Society, 2007.

[17] Robert DeLine and Manuel Fähndrich. Enforcing High-Level Protocols in Low-Level Software. In Michael Burke and Mary Lou Soffa, editors, *PLDI'01*, volume 36(5) of *SIGPLAN Notices*, pages 59–69. ACM Press, 2001.

[18] Mariangiola Dezani-Ciancaglini, Ugo de' Liguoro, and Nobuko Yoshida. On Progress for Structured Communications. In Gilles Barthe and Cédric Fournet, editors, *TGC'07*, volume 4912 of *LNCS*, pages 257–275, 2008.

[19] Mariangiola Dezani-Ciancaglini, Sophia Drossopoulou, Elena Giachino, and Nobuko Yoshida. Bounded Session Types for Object-Oriented Languages. In Frank de Boer and Marcello Bonsangue, editors, *FMCO'06*, volume 4709 of *LNCS*, pages 207–245. Springer-Verlag, 2007.

[20] Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. Session Types for Object-Oriented Languages. In Dave Thomas, editor, *ECOOP'06*, volume 4067 of *LNCS*, pages 328–352. Springer-Verlag, 2006.

[21] Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, Alex Ahern, and Sophia Drossopoulou. A Distributed Object Oriented Language with Session Types. In Rocco De Nicola and Davide Sangiorgi, editors, *TGC'05*, volume 3705 of *LNCS*, pages 299–318. Springer-Verlag, 2005.

[22] Sophia Drossopoulou. Advanced Issues in Object Oriented Languages Course Notes. http://www.doc.ic.ac.uk/˜scd/Teaching/AdvOO.html.

[23] Sophia
Drossopoulou, Mario Coppo, and Mariangiola Dezani-Ciancaglini. Amalgamating the Session Types and the Object Oriented Programming Paradigms. In *MPOOL'07*, 2007. http://homepages.fh-regensburg.de/∼mpool/mpool07/programme.html.

[24] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen C. Hunt, James R. Larus, , and Steven Levi. Language Support for Fast and Reliable Message-based Communication in Singularity OS. In Willy Zwaenepoel, editor, *EuroSys2006*, ACM SIGOPS, pages 177–190. ACM Press, 2006.

[25] Manuel Fähndrich and Robert DeLine. Adoption and Focus: Practical Linear Types for Imperative Programming. In Laurie J. Hendren, editor, *PLDI'02*, volume 37(5) of *SIGPLAN Notices*, pages 13–24. ACM Press, 2002.

[26] Pablo Garralda, Adriana Compagnoni, and Mariangiola Dezani-Ciancaglini. BASS: Boxed Ambients with Safe Sessions. In Michael Maher, editor, *PPDP'06*, pages 61–72. ACM Press, 2006.

[27] Simon Gay. Bounded Polymorphism in Session Types. *Mathematical Structures in Computer Science*, 18(5), 2008. To Appear.

[28] Simon Gay and Malcolm Hole. Subtyping for Session Types in the Pi-Calculus. *Acta Informatica*, 42(2/3):191–225, 2005.

[29] Simon Gay, Vasco T. Vasconcelos, and António Ravara. Session Types for Inter-Process Communication. TR 2003–133, Department of Computing, University of Glasgow, 2003.

[30] Andrew D. Gordon and Alan Jeffrey. Typing Correspondence Assertions for Communication Protocols. In Stephen Brooks and Michael Mislove, editors, *MFPS'01*, volume 45 of *ENTCS*, pages 379–409. Elsevier, 2001.

[31] J. Roger Hindley. The Principal Type Scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[32] Kohei Honda. Types for Dyadic Interaction. In Eike Best, editor, *CONCUR'93*, volume 715 of *LNCS*, pages 509–523. Springer-Verlag, 1993.

[33] Kohei Honda. Composing Processes. In Guy L. Steele, editor, *POPL'96*, pages 344–357. ACM Press, 1996.

[34] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language Primitives and Type Disciplines for Structured Communication-based Programming. In Chris Hankin, editor, *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer-Verlag, 1998.

[35] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Web Services, Mobile Processes and Types. *The Bulletin of the European Association for Theoretical Computer Science*, 91:165–185, 2007.

[36] Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based Distributed Programming in Java. In Jan Vitek, editor, *ECOOP'08*, volume 5142 of *LNCS*, pages 516–541. Springer-Verlag, 2008.

[37] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a Minimal Core Calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.

[38] Naoki Kobayashi. A New Type System for Deadlock-Free Processes. In Christel Baier and Holger Hermanns, editors, *CONCUR'06*, volume 4137 of *LNCS*, pages 233–247. Springer-Verlag, 2006.

[39] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the Pi-Calculus. *ACM TOPLAS*, 21(5):914–947, 1999.

[40] Cosimo Laneve and Luca Padovani. The Pairing of Contracts and Session Types. In Pierpaolo Degano, Rocco De Nicola, and José Meseguer, editors, *Concurrency, Graphs and Models*, volume 5065 of *LNCS*, pages 681–670. Springer-Verlag, 2008.

[41] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, Parts I and II. *Information and Computation*, 100(1), 1992.

[42] Dimitris Mostrous. Moose: a Minimal Object Oriented Language with Session Types. Master's thesis, Imperial College London, 2005.

[43] Dimitris Mostrous and Nobuko Yoshida. Two Session Typing Systems for Higher-order Mobile Processes. In Simona Ronchi Della Rocca, editor, *TLCA'07*, volume 4583 of *LNCS*, pages 321–335. Springer-Verlag, 2007.

[44] Matthias Neubauer and Peter Thiemann. An Implementation of Session Types. In Bharat Jayaraman, editor, *PADL'04*, volume 3057 of *LNCS*, pages 56–70. Springer-Verlag, 2004.

[45] Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable Extensibility via Nested Inheritance. In Doug Schmidt, editor, *OOPSLA'04*, pages 99–115. ACM Press, 2004.

[46] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[47] Stephen Sparkes. Conversation with Steve Ross-Talbot. *ACM Queue*, 4(2), 2006.

[48] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An Interaction-based Language and its Typing System. In C. Halatsis, D. Maritsas, G. Philokyprou, and S. Theodoridis, editors, *PARLE'94*, volume 817 of *LNCS*, pages 398–413. Springer-Verlag, 1994.

[49] Antonio Vallecillo, Vasco T. Vasconcelos, and António Ravara. Typing the Behavior of Objects and Components using Session Types. In Antonio Brogi and Jean-Marie Jacquet, editors, *FOCLASA'02*, volume 68(3) of *ENTCS*, pages 439–456. Elsevier, 2002.

[50] Vasco Vasconcelos. Typed Concurrent Objects. In Mario Tokoro and Remo Pareschi, editors, *ECOOP'94*, volume 821 of *LNCS*, pages 100–117. Springer-Verlag, 1994.

[51] Vasco T. Vasconcelos, Simon Gay, and António Ravara. Typechecking a Multithreaded Functional Language with Session Types. *Theorical Computer Science*, 368:64–87, 2006.

[52] Web Services Choreography Working Group. Web Services Choreography Description Language. http://www.w3.org/2002/ws/chor/.

[53] Nobuko Yoshida. Graph Types for Monadic Mobile Processes. In Vijay Chandru and V. Vinay, editors, *FSTTCS'96*, number 1180 in LNCS, pages 371–386. Springer-Verlag, 1996.

[54] Nobuko Yoshida and Vasco T. Vasconcelos. Language Primitives and Type Disciplines for Structured Communication-based Programming Revisited. In Maribel Fernández and Claude Kirchner, editors, *SecReT'06*, volume 171 of *ENTCS*, pages 73–93. Elsevier, 2007.

A Proof of Subject Reduction

A.1 Generation Lemmas

We will prove in Lemma 7.2 that $\preceq$ preserves the types of expressions. In Lemma A.6 we will show that $\preceq$ preserves also the types of threads.

Lemma 7.2 *If* $\Sigma;\mathscr{S} \preceq \Sigma';\mathscr{S}'$ *and* $\Gamma;\Sigma;\mathscr{S} \vdash \mathsf{e} : \mathsf{t}$, *then* $\Gamma;\Sigma';\mathscr{S}' \vdash \mathsf{e} : \mathsf{t}$.

Proof By induction on the number of basic steps to establish $\Sigma;\mathscr{S} \preceq \Sigma';\mathscr{S}'$ (in the sense of Definition 7.1), and application of the non-structural rules.

Lemma A.1 (Generation for Standard Expressions) *(1)* $\Gamma;\Sigma;\mathscr{S} \vdash \mathsf{x} : \mathsf{t}$ *implies* $\emptyset;\emptyset \preceq \Sigma;\mathscr{S}$ *and* $\mathsf{x} : \mathsf{t}' \in \Gamma$ *for some* $\mathsf{t}' <: \mathsf{t}$.

(2) $\Gamma;\Sigma;\mathscr{S} \vdash \mathsf{c} : \mathsf{t}$ *implies* $\emptyset;\emptyset \preceq \Sigma;\mathscr{S}$ *and* $\mathsf{c} : \mathsf{sch} \in \Gamma$ *and* $\mathsf{t} = (\mathsf{s},\overline{\mathsf{s}})$ *or* $\mathsf{t} = \mathsf{s}$.

(3) $\Gamma;\Sigma;\mathscr{S} \vdash \mathsf{null} : \mathsf{t}$ *implies* $\emptyset;\emptyset \preceq \Sigma;\mathscr{S}$.

(4) $\Gamma;\Sigma;\mathscr{S} \vdash \mathsf{v} : \mathsf{t}$ *with* $\mathsf{v} \in \{\mathsf{true},\mathsf{false}\}$ *implies* $\emptyset;\emptyset \preceq \Sigma;\mathscr{S}$ *and* $\mathsf{t} = \mathsf{bool}$.

(5) $\Gamma;\Sigma;\mathscr{S} \vdash \mathsf{o} : \mathsf{t}$ *implies* $\emptyset;\emptyset \preceq \Sigma;\mathscr{S}$ *and* $\mathsf{o} : C \in \Gamma$ *for some* $C <: \mathsf{t}$.

(6) $\Gamma;\Sigma;\mathscr{S} \vdash \mathsf{NullExc} : \mathsf{t}$ *implies* $\emptyset;\emptyset \preceq \Sigma;\mathscr{S}$.

(7) $\Gamma;\Sigma;\mathscr{S} \vdash \mathsf{this} : \mathsf{t}$ *implies* $\emptyset;\emptyset \preceq \Sigma;\mathscr{S}$ *and and* $\mathsf{this} : C \in \Gamma$ *for some* $C <: \mathsf{t}$.

(8) $\Gamma;\Sigma;\mathscr{S} \vdash \mathsf{e}_1;\mathsf{e}_2 : \mathsf{t}$ *implies* $\Sigma = \Sigma_1 \circ \Sigma_2$, *and* $\mathsf{t} = \mathsf{t}_2$ *and* $\Gamma;\Sigma_i;\mathscr{S} \vdash \mathsf{e}_i : \mathsf{t}_i$ *for some* Σ_i, t_i $(i \in \{1,2\})$.

(9) $\Gamma;\Sigma;\mathscr{S} \vdash \mathsf{e.f} \coloneqq \mathsf{e}' : \mathsf{t}$ *implies* $\Sigma = \Sigma_1 \circ \Sigma_2$, *and* $\Gamma;\Sigma_1;\mathscr{S} \vdash \mathsf{e} : C$ *and* $\Gamma;\Sigma_2;\mathscr{S} \vdash \mathsf{e}' : \mathsf{t}$ *with* $\mathsf{f}\,\mathsf{t} \in \mathsf{fields}(C)$ *for some* Σ_1, Σ_2, C.

(10) $\Gamma;\Sigma;\mathscr{S} \vdash \mathsf{e.f} : \mathsf{t}$ *implies* $\Gamma;\Sigma;\mathscr{S} \vdash \mathsf{e} : C$ *and* $\mathsf{f}\,\mathsf{t} \in \mathsf{fields}(C)$ *for some* C.

(11) $\Gamma;\Sigma;\mathscr{S} \vdash \mathsf{e.m}(\mathsf{e}_1,\ldots,\mathsf{e}_n) : \mathsf{t}$ *implies* $\Gamma;\Sigma_0;\mathscr{S}' \vdash \mathsf{e} : C$, *and* $\Gamma;\Sigma_i;\mathscr{S}' \vdash \mathsf{e}_i : \mathsf{t}_i$ *for* $1 \le i \le n-m$, *and* $\mathsf{e}_{n-m+j} = \mathsf{u}_j$ *for* $1 \le j \le m$, *and* $\Sigma_0 \circ \Sigma_1 \ldots \circ \Sigma_{n-m} \circ \{\mathsf{u}_1 : \rho_1, \ldots, \mathsf{u}_m : \rho_m\};\mathscr{S}' \preceq \Sigma;\mathscr{S}$ *and* $\mathsf{mtype}(\mathsf{m}, C) = \mathsf{t}_1, \ldots, \mathsf{t}_{n-m}, \rho_1, \ldots, \rho_m \xrightarrow{\ominus} \mathsf{t}$, *for some* m $(0 \le m \le n)$, $\mathscr{S}', \Sigma_i, \mathsf{t}_i, \mathsf{u}_j, \rho_j, C$ $(1 \le i \le n-m,\ 1 \le j \le m)$.

(12) $\Gamma;\Sigma;\mathscr{S} \vdash \mathsf{e.m}(\mathsf{e}_1,\ldots,\mathsf{e}_n) : \mathsf{t}$ *implies* $\Gamma;\Sigma_0;\{\mathsf{u}_1\} \vdash \mathsf{e} : C$, *and* $\Gamma;\Sigma_i;\{\mathsf{u}_1\} \vdash \mathsf{e}_i : \mathsf{t}_i$ *for* $1 \le i \le n-m$, *and* $\mathsf{e}_{n-m+j} = \mathsf{u}_j$ *for* $1 \le j \le m$, *and* $\Sigma_0 \circ \Sigma_1 \ldots \circ \Sigma_{n-m} \circ \{\mathsf{u}_1 : \rho_1, \ldots, \mathsf{u}_m : \rho_m\};\{\mathsf{u}_1\} \preceq \Sigma;\mathscr{S}$ *and* $\mathsf{mtype}(\mathsf{m}, C) = \mathsf{t}_1, \ldots, \mathsf{t}_{n-m}, \rho_1, \ldots, \rho_m \xrightarrow{\oplus} \mathsf{t}$, *for some* m $(1 \le m \le n)$, $\Sigma_i, \mathsf{t}_i, \mathsf{u}_j, \rho_j, C$ $(1 \le i \le n-m,\ 1 \le j \le m)$.

(13) $\Gamma;\Sigma;\mathscr{S} \vdash \mathsf{new}\ C : \mathsf{t}$ *implies* $\emptyset;\emptyset \preceq \Sigma;\mathscr{S}$ *and* $C <: \mathsf{t}$.

(14) $\Gamma;\Sigma;\mathscr{S} \vdash \mathsf{new}\ (\mathsf{s},\overline{\mathsf{s}}) : \mathsf{t}$ *implies* $\emptyset;\emptyset \preceq \Sigma;\mathscr{S}$ *and* $(\mathsf{s},\overline{\mathsf{s}}) <: \mathsf{t}$.

(15) $\Gamma;\Sigma;\mathscr{S} \vdash \mathsf{spawn}\{\ \mathsf{e}\ \} : \mathsf{t}$ *implies* $\Sigma';\mathscr{S}' \preceq \Sigma;\mathscr{S}$, *and ended*$(\Sigma')$ *and* $\mathsf{t} = \mathit{Object}$ *and* $\Gamma;\Sigma';\mathscr{S}' \vdash \mathsf{e} : \mathsf{t}'$ *for some* $\Sigma', \mathscr{S}', \mathsf{t}'$.

Proof By induction on typing derivations, then case analysis over the shape of the expression being typed, and then case analysis over the last rule applied. We just show two paradigmatic cases of the inductive step.

(12) If the expression being typed has the shape $\mathsf{e.m}(\mathsf{e}_1,\ldots,\mathsf{e}_n)$, then the last rule applied is **MethPlus**, **MethMinus** or one of the structural rules. We only consider the case where the last applied rule is **Consume**, then **MethPlus**:

$$\frac{\Gamma;\Sigma,\mathsf{u} : \varepsilon.\mathsf{end};\mathscr{S} \vdash \mathsf{e.m}(\mathsf{e}_1,\ldots,\mathsf{e}_n) : \mathsf{t}}{\Gamma;\Sigma,\mathsf{u} : \updownarrow;\mathscr{S} \vdash \mathsf{e.m}(\mathsf{e}_1,\ldots,\mathsf{e}_n) : \mathsf{t}}$$

By induction hypothesis we get $\Gamma;\Sigma_0;\mathscr{S} \vdash \mathsf{e} : C$, and $\Gamma;\Sigma_i;\mathscr{S} \vdash \mathsf{e}_i : \mathsf{t}_i$ for $1 \le i \le n-m$, and $\mathsf{e}_{n-m+j} = \mathsf{u}_j$ for $1 \le j \le m$, and $\Sigma_0 \circ \Sigma_1 \ldots \circ \Sigma_{n-m} \circ \{\mathsf{u}_1 : \rho_1, \ldots, \mathsf{u}_m : \rho_m\};\{\mathsf{u}_1\} \preceq \Sigma,\mathsf{u} : \varepsilon.\mathsf{end};\mathscr{S}$ and $\mathsf{mtype}(\mathsf{m}, C) = \mathsf{t}_1, \ldots, \mathsf{t}_{n-m}, \rho_1, \ldots, \rho_m \xrightarrow{\oplus} \mathsf{t}$, for some m $(1 \le m \le n)$, $\Sigma_i, \mathsf{t}_i, \mathsf{u}_j, \rho_j, C$ $(1 \le i \le n-m,\ 1 \le j \le m)$. By definition we also have that $\Sigma,\mathsf{u} : \varepsilon.\mathsf{end};\mathscr{S} \preceq \Sigma,\mathsf{u} : \updownarrow;\mathscr{S}$, and from transitivity of $\preceq$ we obtain that $\Sigma_0 \circ \Sigma_1 \ldots \circ \Sigma_{n-m} \circ \{\mathsf{u}_1 : \rho_1, \ldots, \mathsf{u}_m : \rho_m\};\{\mathsf{u}_1\} \preceq \Sigma,\mathsf{u} : \updownarrow;\mathscr{S}$.

(15) If the expression being typed has the shape $\mathsf{spawn}\{\ \mathsf{e}\ \}$, then the last rule applied is either **Spawn**, or one of the structural rules.

If last applied rule is **WeakB**, then

$$\frac{\Gamma;\Sigma,\mathsf{c}:\rho;\{\mathsf{c}\}\vdash \mathsf{spawn}\,\{\;\mathsf{e}\;\}:\mathsf{t}}{\Gamma;\Sigma,\mathsf{c}:\mathsf{begin}.\rho;\emptyset\vdash \mathsf{spawn}\,\{\;\mathsf{e}\;\}:\mathsf{t}}$$

By induction hypothesis there exist $\Sigma', \mathscr{S}'$, so that $\Sigma';\mathscr{S}' \preceq \Sigma,\mathsf{c}:\rho;\{\mathsf{c}\}$, and $\mathit{ended}(\Sigma')$ and $\mathsf{t} = \mathit{Object}$ and $\Gamma;\Sigma';\mathscr{S}' \vdash \mathsf{e}:\mathsf{t}'$. The rest follows from the fact that, by definition, $\Sigma,\mathsf{c}:\rho;\{\mathsf{c}\} \preceq \Sigma,\mathsf{c}:\mathsf{begin}.\rho;\emptyset$, and from transitivity of $\preceq$.

Lemma A.2 (Generation for Communication Expressions) *(1)* $\Gamma;\Sigma;\mathscr{S}\vdash \mathsf{connect\ u\ s}\,\{\mathsf{e}\}:\mathsf{t}$ *implies* $\mathsf{s} = \mathsf{begin}.\eta$, *and* $\Gamma;\emptyset;\emptyset\vdash\mathsf{u}:\mathsf{begin}.\eta$ *and* $\Gamma\setminus\mathsf{u};\Sigma',\mathsf{u}:\eta;\{\mathsf{u}\}\vdash\mathsf{e}:\mathsf{t}$, *and* $\Sigma';\emptyset\preceq\Sigma;\mathscr{S}$ *for some* η,Σ'.

(2) $\Gamma;\Sigma;\mathscr{S}\vdash\mathsf{u\,.receive}:\mathsf{t}$ *implies* $\{\mathsf{u}:?\mathsf{t}\};\{\mathsf{u}\}\preceq\Sigma;\mathscr{S}$.

(3) $\Gamma;\Sigma;\mathscr{S}\vdash\mathsf{u\,.send}\,(\mathsf{e}):\mathsf{t}$ *implies* $\mathsf{t} = \mathit{Object}$ *and* $\Gamma;\Sigma';\{\mathsf{u}\}\vdash\mathsf{e}:\mathsf{t}'$ *and* $\Sigma'\circ\{\mathsf{u}:!\mathsf{t}\};\{\mathsf{u}\}\preceq\Sigma;\mathscr{S}$ *for some* Σ',t'.

(4) $\Gamma;\ \Sigma;\mathscr{S}\vdash\mathsf{u\,.receiveS}\,(\mathsf{x})\{\mathsf{e}\}:\mathsf{t}$ *implies* $\mathsf{t} = \mathit{Object}$ *and* $\Gamma\setminus\mathsf{x}\,;\ \{\mathsf{x}:\eta\};\{\mathsf{x}\}\vdash\mathsf{e}:\mathsf{t}'$ *and* $\{\mathsf{u}:?(\eta)\};\{\mathsf{u}\}\preceq\Sigma;\mathscr{S}$ *for some* $\mathsf{t}',\eta\neq\varepsilon.\mathsf{end}$.

(5) $\Gamma;\ \Sigma;\mathscr{S}\vdash\mathsf{u\,.sendS}\,(\mathsf{u}')\,:\,\mathsf{t}$ *implies* $\mathsf{t} = \mathit{Object}$ *and* $\{\mathsf{u}':\eta,\mathsf{u}:!(\eta)\};\{\mathsf{u}\}\preceq\Sigma;\mathscr{S}$ *for some* $\eta\neq\varepsilon.\mathsf{end}$.

(6) $\Gamma;\Sigma;\mathscr{S}\vdash\mathsf{u\,.receiveIf}\,\{\mathsf{e}_1\}\{\mathsf{e}_2\}:\mathsf{t}$ *implies* $\Gamma;\Sigma',\mathsf{u}:\rho_i;\{\mathsf{u}\}\vdash\mathsf{e}_i:\mathsf{t}$ *(*$i\in\{1,2\}$*)* *and* $\Sigma',\mathsf{u}:?\langle\rho_1,\rho_2\rangle;\{\mathsf{u}\}\preceq\Sigma;\mathscr{S}$ *for some* Σ',ρ_1,ρ_2.

(7) $\Gamma;\Sigma;\mathscr{S}\vdash\mathsf{u\,.sendIf}\,(\mathsf{e})\{\mathsf{e}_1\}\{\mathsf{e}_2\}:\mathsf{t}$ *implies* $\Gamma;\Sigma_1;\{\mathsf{u}\}\vdash\mathsf{e}:\mathsf{bool}$ *and* $\Gamma;\Sigma_2,\mathsf{u}:\rho_i;\{\mathsf{u}\}\vdash\mathsf{e}_i:\mathsf{t}$ *(*$i\in\{1,2\}$*)* *and* $\Sigma_1\circ\Sigma_2,\mathsf{u}:!\langle\rho_1,\rho_2\rangle;\{\mathsf{u}\}\preceq\Sigma;\mathscr{S}$ *for some* $\Sigma_1,\Sigma_2,\rho_1,\rho_2$.

(8) $\Gamma;\Sigma;\mathscr{S}\vdash\mathsf{u\,.receiveWhile}\,\{\mathsf{e}\}:\mathsf{t}$ *implies* $\Gamma;\{\mathsf{u}:\pi\};\{\mathsf{u}\}\vdash\mathsf{e}:\mathsf{t}$ *and* $\{\mathsf{u}:?\langle\pi\rangle^*\};\{\mathsf{u}\}\preceq\Sigma;\mathscr{S}$ *for some* π.

(9) $\Gamma;\Sigma;\mathscr{S}\vdash\mathsf{u\,.sendWhile}\,(\mathsf{e})\{\mathsf{e}'\}:\mathsf{t}$ *implies* $\Gamma;\emptyset;\emptyset\vdash\mathsf{e}:\mathsf{bool}$ *and* $\Gamma;\{\mathsf{u}:\pi\};\{\mathsf{u}\}\vdash\mathsf{e}':\mathsf{t}$ *and* $\{\mathsf{u}:!\langle\pi\rangle^*\};\{\mathsf{u}\}\preceq\Sigma;\mathscr{S}$ *for some* π.

Proof Similar to the proof of Lemma A.1.

Lemma A.3 (Generation for Threads) *(1)* $\Gamma;\Sigma\vdash\mathsf{e}:\mathsf{thread}$ *implies* $\Gamma;\Sigma;\mathscr{S}\vdash\mathsf{e}:\mathsf{t}$ *for some* t.

(2) $\Gamma;\Sigma\vdash P_1\,|\,P_2:\mathsf{thread}$ *implies* $\Sigma=\Sigma_1\|\Sigma_2$ *and* $\Gamma;\Sigma_i\vdash P_i:\mathsf{thread}$ *(*$i\in\{1,2\}$*) for some* Σ_1,Σ_2.

Proof Similar to the proof of Lemma A.1.

A.2 Types Preservation under Structural Equivalence, and under Substitutions

As a convenient shorthand, for any two entities x and y which belong to a domain that includes $\perp$, we use the notation $x \triangleq y$ to indicate that x is defined if and only if y is defined, and if x is defined, then $x = y$.

In Lemma A.7 we show that structural equivalence of terms preserves types. To prove this, we first prove in Lemma A.4 the neutrality of element $\emptyset$, and associativity and commutativity of parallel composition of session environments. Moreover we show in Lemma A.5 various properties of $\preceq$, $\|$, and $\circ$ which easily follow from their definitions.

Lemma A.4 *(1)* $\Sigma_1 \| \emptyset = \Sigma_1 = \emptyset \| \Sigma_1$.
(2) $\Sigma_1 \| \Sigma_2 \triangleq \Sigma_2 \| \Sigma_1$.
(3) $\Sigma_1 \| (\Sigma_2 \| \Sigma_3) \triangleq (\Sigma_1 \| \Sigma_2) \| \Sigma_3$.

Proof Note that for any Σ, Σ', if $\Sigma \| \Sigma'$ is defined, then $\mathscr{D}(\Sigma \| \Sigma') = \mathscr{D}(\Sigma) \cup \mathscr{D}(\Sigma')$.
(1) follows from definition of $\|$.
For (2) show $\forall \mathsf{u} \in \mathscr{D}(\Sigma_1) \cup \mathscr{D}(\Sigma_2) : \ \Sigma_1(\mathsf{u}) \| \Sigma_2(\mathsf{u}) \triangleq \Sigma_2(\mathsf{u}) \| \Sigma_1(\mathsf{u})$. For (3) show $\forall \mathsf{u} \in \mathscr{D}(\Sigma_1) \cup \mathscr{D}(\Sigma_2) \cup \mathscr{D}(\Sigma_3) : \ \Sigma_1(\mathsf{u}) \| (\Sigma_2(\mathsf{u}) \| \Sigma_3(\mathsf{u})) \triangleq (\Sigma_1(\mathsf{u}) \| \Sigma_2(\mathsf{u})) \| \Sigma_3(\mathsf{u})$.

The next Lemma, *i.e.,* A.5, characterizes small modifications on operations that preserve well-formedness of the session environment composition, $\|$ and $\circ$, and also the preservation of the relationship $\preceq$. It will be used in the proof of Subject Reduction.

We define:

$$\Sigma[\mathsf{u} \mapsto \theta](\mathsf{u}') = \begin{cases} \theta & \text{if } \mathsf{u} = \mathsf{u}', \\ \Sigma(\mathsf{u}') & \text{otherwise.} \end{cases}$$

Lemma A.5 *(1)* $\emptyset \preceq \Sigma_1$, *and* $\Sigma_1 \| \Sigma_2$ *defined imply* $\Sigma_2 \preceq \Sigma_1 \| \Sigma_2$.
(2) $\Sigma_1 \| \Sigma_2 \preceq \Sigma$, *implies that there are* Σ'_1, Σ'_2 *such that* $\Sigma_1 \preceq \Sigma'_1$ *and* $\Sigma_2 \preceq \Sigma'_2$ *and* $\Sigma'_1 \| \Sigma'_2 = \Sigma$.
(3) $\Sigma_1 \preceq \Sigma'_1$, *and* $\Sigma'_1 \circ \Sigma_2$ *defined, imply* $\Sigma_1 \circ \Sigma_2$ *defined, and* $\Sigma_1 \circ \Sigma_2 \preceq \Sigma'_1 \circ \Sigma_2$.
(4) *ended*(Σ_1) *and* $(\Sigma_1 \cup \Sigma'_1) \circ \Sigma_2$ *defined imply*
  (a) $\Sigma'_1 \circ \Sigma_2$ *defined,*
  (b) $(\Sigma_1 \cup \Sigma'_1) \circ \Sigma_2 = \Sigma_1 \| \Sigma'_1 \circ \Sigma_2$.
(5) $\Sigma; \mathscr{S} \preceq \Sigma'; \mathscr{S}'$ *implies*
  (a) $\Sigma \setminus \mathsf{u}; \emptyset \preceq \Sigma' \setminus \mathsf{u}; \mathscr{S}'$,
  (b) $\Sigma \setminus \mathsf{u}; \mathscr{S} \preceq \Sigma' \setminus \mathsf{u}; \mathscr{S}'$ *when* $\mathscr{S} \neq \{\mathsf{u}\}$.
(6) $\{\mathsf{u} : \theta\}; \mathscr{S} \preceq \Sigma; \mathscr{S}'$ *implies*
  (a) $\Sigma(\mathsf{u}) \in \{\theta, \theta.\mathsf{end}, \mathsf{begin}.\theta, \mathsf{begin}.\theta.\mathsf{end}, \updownarrow\}$ *and* $\mathscr{R}(\Sigma \setminus \mathsf{u}) \subseteq \{\varepsilon, \varepsilon.\mathsf{end}, \mathsf{begin}.\varepsilon, \mathsf{begin}.\varepsilon.\mathsf{end}, \updownarrow\}$;
  (b) $\{\mathsf{u} : \theta'\}; \emptyset \preceq \Sigma[\mathsf{u} \mapsto \theta']; \mathscr{S}'$ *for all* θ';
  (c) $\{\mathsf{u} : \eta\}; \mathscr{S} \preceq \Sigma'[\mathsf{u} \mapsto \mathsf{begin}.\eta]; \mathscr{S}'$ *for all* η.
(7) $\{\mathsf{u} : \theta\}; \mathscr{S} \preceq \Sigma; \mathscr{S}'$ *and* $\Sigma \circ \Sigma'$ *defined imply*
  (a) $\Sigma[\mathsf{u} \mapsto \pi] \circ \Sigma'$ *defined for all* π;
  (b) $\Sigma[\mathsf{u} \mapsto \mathsf{begin}.\pi] \circ \Sigma'$ *defined for all* π;
  (c) $\Sigma'; \mathscr{S}' \preceq \Sigma[\mathsf{u} \mapsto \varepsilon] \circ \Sigma'; \mathscr{S}'$.
(8) $\Sigma_1 \circ \Sigma_2 \| \Sigma_3 \circ \Sigma_4$ *defined, and* $\{\mathsf{u} : \pi\} \preceq \Sigma_1$ *and* $\{\mathsf{u} : \pi'\} \preceq \Sigma_3$ *imply :*
  (a) $\pi = \overline{\pi'}$;
  (b) $\Sigma_1[\mathsf{u} \mapsto \pi''] \circ \Sigma_2 \| \Sigma_3[\mathsf{u} \mapsto \overline{\pi''}] \circ \Sigma_4 = \Sigma_1 \circ \Sigma_2 \| \Sigma_3 \circ \Sigma_4$, *for all* π'';

(c) $\Sigma_1[\mathsf{u} \mapsto \mathsf{begin}.\pi'']\circ\Sigma_2\|\Sigma_3[\mathsf{u} \mapsto \mathsf{begin}.\overline{\pi''}]\circ\Sigma_4 = \Sigma_1\circ\Sigma_2\|\Sigma_3\circ\Sigma_4$, *for all* π''.

Proof For (1) notice that $\emptyset \preceq \Sigma_1$ implies $\mathscr{R}(\Sigma_1) \subseteq \{\varepsilon, \varepsilon.\mathsf{end}, \mathsf{begin}.\varepsilon, \mathsf{begin}.\varepsilon.\mathsf{end}, \updownarrow$ $\}$ and that $\Sigma_1\|\Sigma_2$ defined implies $\Sigma_1(\mathsf{u}) = \overline{(\Sigma_2(\mathsf{u}))}$ for all $\mathsf{u} \in \mathscr{D}(\Sigma_1)\cap\mathscr{D}(\Sigma_2)$.
For (2) one can obtain Σ_1' and Σ_2' by applying to Σ_1 and Σ_2 the same transformations which build Σ from $\Sigma_1\|\Sigma_2$.
(3) follows easily from the definitions of $\preceq$ and of $\circ$.
(4a) is immediate. For (4b), *ended*(Σ_1) and $(\Sigma_1 \cup \Sigma_1')\circ\Sigma_2$ defined imply that $\mathscr{D}(\Sigma_1)\cap\mathscr{D}(\Sigma_2) = \emptyset$.
(5a) and (5b) follow from the definition of $\preceq$. The condition $\mathscr{S} \neq \{\mathsf{u}\}$ is necessary since for example $\{\mathsf{u}\,{:}!\mathsf{bool}\};\{\mathsf{u}\} \preceq \{\mathsf{u}:\mathsf{begin}.!\mathsf{bool}\};\emptyset$, but $\emptyset;\{\mathsf{u}\} \not\preceq \emptyset;\emptyset$.
(6a) follows from the definition of $\preceq$ and (6b), (6c) are consequences of (6a).
(6a) implies (7a), (7b) and (7c).
The definition of $\|$ and (6a) imply (8a). Points (8b), and (8c) follow from the observation that in all the equated session environments the predicates of u are $\updownarrow$.

Lemma A.6 *If* $\Sigma \preceq \Sigma'$ *and* $\Gamma;\Sigma \vdash P:\mathsf{thread}$, *then* $\Gamma;\Sigma' \vdash P:\mathsf{thread}$.

Proof By induction on derivations. If the last applied rule is **Start** use Lemma 7.2. If the last applied rule is **Par** use Lemma A.5(2) and induction.

Lemma A.7 (Preservation of Typing under Structural Equivalence) *If* $\Gamma;\Sigma \vdash P:\mathsf{thread}$ *and* $P \equiv P'$, *then* $\Gamma;\Sigma \vdash P':\mathsf{thread}$.

Proof By induction on the derivation of $\equiv$.
For the case where $P' = P\,|\,\mathsf{null}$, we use Lemma A.3(2), and obtain $\Sigma = \Sigma_1\|\Sigma_2$ and $\Gamma;\Sigma_1 \vdash P:\mathsf{thread}$ and $\Gamma;\Sigma_2 \vdash \mathsf{null}:\mathsf{thread}$. Using Lemma A.3(1) and Lemma A.1(3) we get $\Gamma;\Sigma_2;\mathscr{S} \vdash \mathsf{null}\ :\mathsf{t}_2$, and $\emptyset;\emptyset \preceq \Sigma_2;\mathscr{S}$. Using Lemma A.5(1), we obtain that $\Sigma_1 \preceq \Sigma$, and from that, with Lemma A.6 we obtain that $\Gamma;\Sigma;\mathscr{S} \vdash P:\mathsf{thread}$.
For the other two basic cases use Lemmas A.3(1) and A.4(2)-(3). For the induction case use Lemma A.3(1) and induction hypothesis.

We need a substitution lemma which takes into account not only the substitutions of variables by values, but also the substitutions of this by object identifiers and the substitutions of channel names and variables by fresh channel names. The proof by induction on derivations is standard.

Lemma A.8 (Preservation of Typing under Substitution) *(1) If* $\Gamma,\mathsf{x}:\mathsf{t}\,;\Sigma;\mathscr{S} \vdash \mathsf{e}:\mathsf{t}'$ *and* $\Gamma;\emptyset;\emptyset \vdash \mathsf{v}:\mathsf{t}$, *then* $\Gamma;\Sigma;\mathscr{S} \vdash \mathsf{e}[{}^{\mathsf{v}}/_{\mathsf{x}}]:\mathsf{t}'$.
(2) If $\Gamma,\mathsf{this}{:}C\,;\Sigma;\mathscr{S} \vdash \mathsf{e}:\mathsf{t}$ *and* $\Gamma;\emptyset;\emptyset \vdash \mathsf{o}:C$, *then* $\Gamma;\Sigma;\mathscr{S} \vdash \mathsf{e}[{}^{\mathsf{o}}/_{\mathsf{this}}]:\mathsf{t}$.
(3) If $\Gamma\backslash\mathsf{u}\,;\Sigma;\mathscr{S} \vdash \mathsf{e}:\mathsf{t}$ *and* c *is fresh, then* $\Gamma;\Sigma[{}^{\mathsf{c}}/_{\mathsf{u}}];\mathscr{S}[{}^{\mathsf{c}}/_{\mathsf{u}}] \vdash \mathsf{e}[{}^{\mathsf{c}}/_{\mathsf{u}}]:\mathsf{t}$.

A.3 Types in Subderivations, and Substitutions within Contexts

Lemma 7.3 [Subderivations]

If $\Gamma;\Sigma;\mathscr{S} \vdash E[\mathsf{e}]:\mathsf{t}$, *then there exist* $\Sigma_1,\Sigma_2,\mathsf{t}'$, *such that for all* x *fresh in* E,Γ, $\Sigma = \Sigma_1 \circ \Sigma_2$, *and* $\Gamma;\Sigma_1;\mathscr{S} \vdash \mathsf{e}:\mathsf{t}'$, *and* $\Gamma,\mathsf{x}:\mathsf{t}';\Sigma_2;\mathscr{S} \vdash E[\mathsf{x}]:\mathsf{t}$.

Proof By induction on E, and using Generation Lemmas. For example if $E = [\,];\mathsf{e}'$, then $\Gamma;\Sigma;\mathscr{S} \vdash \mathsf{e};\mathsf{e}':\mathsf{t}$ implies $\Sigma = \Sigma_1 \circ \Sigma_2$ and $\Gamma;\Sigma_1;\mathscr{S} \vdash \mathsf{e}:\mathsf{t}'$ and $\Gamma;\Sigma_2;\mathscr{S} \vdash \mathsf{e}':\mathsf{t}$ by Lemma A.1(8). Then we get $\Gamma,\mathsf{x}:\mathsf{t}';\Sigma_2;\mathscr{S} \vdash \mathsf{x};\mathsf{e}':\mathsf{t}$ by rules **Var** and **Seq**.

Lemma 7.4 [Context Substitution] *If* $\Gamma;\Sigma_1;\mathscr{S} \vdash \mathsf{e}:\mathsf{t}'$, *and* $\Gamma,\mathsf{x}:\mathsf{t}';\Sigma_2;\mathscr{S} \vdash E[\mathsf{x}]:\mathsf{t}$, *and* $\Sigma_1 \circ \Sigma_2$ *is defined, then* $\Gamma;\Sigma_1 \circ \Sigma_2;\mathscr{S} \vdash E[\mathsf{e}]:\mathsf{t}$.

Proof By induction on E, and using the Generation Lemmas.

A.4 Name Occurrence

Lemma A.9 formalises that a channel or object identifier that occurs in an expression must occur also in the typing environments of that expression.

Lemma A.9 (Name Occurrence) *(1) If* $\Gamma;\Sigma;\mathscr{S} \vdash \mathsf{e}:\mathsf{t}$ *and* $\mathsf{o} \in \mathsf{fv}(\mathsf{e})$, *then* $\mathsf{o} \in \mathscr{D}(\Gamma)$;

(2) If $\Gamma;\Sigma;\mathscr{S} \vdash \mathsf{e}:\mathsf{t}$ *and* $\mathsf{c} \in \mathsf{fv}(\mathsf{e})$, *then* $\mathsf{c} \in \mathscr{D}(\Gamma) \cup \mathscr{D}(\Sigma)$.

Proof By induction on the typing derivation.

We can then show:

Lemma 7.5 (Fresh Name)

(1) If $\Gamma;\Sigma;\mathscr{S} \vdash \mathsf{e};h$, *then*
   (a) $\mathsf{o} \in \mathsf{e} \Rightarrow \mathsf{o} \in h$;
   (b) $\mathsf{c} \in \mathsf{e} \Rightarrow \mathsf{o} \in h$.
(2) If $\Gamma;\Sigma \vdash P;h$, *then*
   (a) $\mathsf{o} \in P \Rightarrow \mathsf{o} \in h$;
   (b) $\mathsf{c} \in P \Rightarrow \mathsf{o} \in h$.

Proof The proof is by straightforward induction on the typing of the expression (thread): first, by appealing to the well-formed heap judgement, we show that any object or channel occurring in typing environments must occur in heaps which are well-formed with respect to those environments. Based on that, and using Lemma A.9 we obtain the occurrence result.

Theorem 7.6 (Subject Reduction).

(1) $\Gamma;\Sigma;\mathscr{S} \vdash \mathsf{e} : \mathsf{t}$, *and* $\Gamma;\Sigma \vdash h$, *and* $\mathsf{e}, h \longrightarrow \mathsf{e}', h'$ *imply* $\Gamma';\Sigma;\mathscr{S} \vdash \mathsf{e}' : \mathsf{t}$, *and* $\Gamma';\Sigma \vdash h'$, *with* $\Gamma \subseteq \Gamma'$.

(2) $\Gamma;\Sigma \vdash P;h$ *and* $P,h \longrightarrow P',h'$ *imply* $\Gamma';\Sigma' \vdash P;h'$ *with* $\Gamma \subseteq \Gamma'$ *and* $\Sigma \subseteq \Sigma'$.

Proof By induction on the reduction $\mathsf{e}, h \longrightarrow \mathsf{e}', h'$. We only consider the most interesting cases.

Rule **Spawn**. Therefore, the expression being reduced has the form $E[\mathsf{spawn}\,\{\ \mathsf{e}\ \}]$, and

(0) $h' = h$ and $P' = E[\mathsf{null}] \,|\, \mathsf{e}$.

Thus, together with the premises we obtain for some t:

(1) $\Gamma;\Sigma;\mathscr{S} \vdash E[\mathsf{spawn}\,\{\ \mathsf{e}\ \}] : \mathsf{t}$ (2) $\Gamma;\Sigma \vdash h$.

The aim of the next steps is to obtain types for e *and for* $E[\mathsf{null}]$.

Applying Lemma 7.3 on (1) we obtain, that $\exists \mathsf{t}', \Sigma_1, \Sigma_2$ with:

(3) $\Gamma;\Sigma_1;\mathscr{S} \vdash \mathsf{spawn}\,\{\ \mathsf{e}\ \} : \mathsf{t}'$, (4) $\Sigma = \Sigma_1 \circ \Sigma_2$, (5) $\Gamma, x : \mathsf{t}';\Sigma_2;\mathscr{S} \vdash E[x] : \mathsf{t}$.

From (3) and Generation Lemma (*i.e.*, A.1(15)), we obtain for some t'', Σ_1', $\mathscr{S}'$:

(6) $\mathsf{t}' = \mathit{Object}$, (7) $\Gamma;\Sigma_1';\mathscr{S}' \vdash \mathsf{e} : \mathsf{t}''$, (8) $\mathit{ended}(\Sigma_1')$, (9) $\Sigma_1';\mathscr{S}' \preceq \Sigma_1;\mathscr{S}$.

From (5), type rule **Null**, and Context Substitution Lemma (*i.e.*, 7.4), we obtain:

(10) $\Gamma;\Sigma_2;\mathscr{S} \vdash E[\mathsf{null}] : \mathsf{t}$.

From (10) and rule **Start**, and from (7) and rule **Start**, we obtain

(11) $\Gamma;\Sigma_2 \vdash E[\mathsf{null}] : \mathsf{thread}$, (12) $\Gamma;\Sigma_1' \vdash \mathsf{e} : \mathsf{thread}$.

From (11), (12) and rule **Par** we obtain:

(13) $\Gamma;\Sigma_1' \| \Sigma_2 \vdash \mathsf{e} \,|\, E[\mathsf{null}] : \mathsf{thread}$.

The aim of the next steps is to obtain types for $\mathsf{e}\ |E[\mathsf{null}]$ *in session environment* Σ.

From (4) we obtain that $\Sigma_1 \circ \Sigma_2$ is defined, and therefore, from (9) and Lemma A.5(3), we obtain

(14) $\Sigma_1' \circ \Sigma_2$ is defined, and $\Sigma_1' \circ \Sigma_2 \preceq \Sigma_1 \circ \Sigma_2$.

Also, from (8), Lemma A.5(4b), we obtain

(15) $\Sigma_1' \| \Sigma_2 = \Sigma_1' \circ \Sigma_2$.

Therefore, from (13), (14), (15), and Lemma A.6, we obtain

(16) $\Gamma;\Sigma \vdash \mathsf{e} \,|\, E[\mathsf{null}] : \mathsf{thread}$.

The case concludes by taking $\Sigma' = \Sigma$, $\Gamma' = \Gamma$ *and with (16) and (0).*

Rule **Connect**. Then, we have that

(0) $P = E_1[\mathsf{connect}\ \mathsf{c}\ \mathsf{s}\{\mathsf{e}_1\}] \,|\, E_2[\mathsf{connect}\ \mathsf{c}\ \overline{\mathsf{s}}\{\mathsf{e}_2\}]$,

(1) $h' = h :: \mathsf{c}'$, with c' is fresh in h,

(2) $P' = E_1[\mathsf{e}_1[\mathsf{c}'/\mathsf{c}]] \,|\, E_2[\mathsf{e}_2[\mathsf{c}'/\mathsf{c}]]$.

The aim of the next steps is to obtain types for e_1 *and for* e_2.
From premises, (0) and Lemma A.3(2), and A.3(1) we obtain for some $\Sigma_1, \Sigma_2, \mathscr{S}_1, \mathscr{S}_2, \mathsf{t}_1, \mathsf{t}_2$:

(3) $\Sigma = \Sigma_1 \| \Sigma_2$,

(4) $\Gamma; \Sigma_i; \mathscr{S}_i \vdash E_i[\mathsf{connect}\ \mathsf{c}\ \mathsf{s}_i\{\mathsf{e}_i\}]] : \mathsf{t}_i\ (i \in \{1,2\})$,

(5) $\Gamma; \Sigma \vdash h$.

where $\mathsf{s}_1 = \mathsf{s}$ and $\mathsf{s}_2 = \overline{\mathsf{s}}$.
From (4), applying Lemma 7.3, there exist Σ_{11}, Σ_{12}, Σ_{21}, Σ_{22}, $\mathsf{t}'_1, \mathsf{t}'_2$, such that:

(6) $\Sigma_i = \Sigma_{i1} \circ \Sigma_{i2}$,

(7) $\Gamma; \Sigma_{i1}; \mathscr{S}_i \vdash \mathsf{connect}\ \mathsf{c}\ \mathsf{s}_i\{\mathsf{e}_i\} : \mathsf{t}'_i\ (i \in \{1,2\})$,

(8) $\Gamma, \mathsf{x}_i : \mathsf{t}'_i; \Sigma_{i2}; \mathscr{S}_i \vdash E_i[\mathsf{x}_i] : \mathsf{t}_i\ (i \in \{1,2\})$.

From (7), and Lemmas A.2(1) we obtain for some Σ'_{11}, Σ'_{12}, η_1, η_2:

(9) $\Gamma; \emptyset; \emptyset \vdash \mathsf{c} : \mathsf{s}_i$, (10) $\mathsf{s}_i = \mathsf{begin}.\eta_i$,

(11) $\Gamma \setminus \mathsf{c}; \Sigma'_{i1}, \mathsf{c} : \eta_i; \{\mathsf{c}\} \vdash \mathsf{e}_i : \mathsf{t}'_i\ (i \in \{1,2\})$,

(12) $\Sigma'_{i1}; \emptyset \preceq \Sigma_{i1}; \mathscr{S}_i$.

The aim of the next steps is to obtain types for P' in a session environment Σ', so that $\Sigma \subseteq \Sigma'$.
From (1) and (11), and Lemma A.8(3), we get $\Gamma; \Sigma'_{i1}, \mathsf{c}' : \eta_i; \{\mathsf{c}'\} \vdash \mathsf{e}_i[{}^{\mathsf{c}'}\!/\mathsf{c}] : \mathsf{t}'_i$ $(i \in \{1,2\})$ which implies by rule **WeakB**:

(13) $\Gamma; \Sigma'_{i1}, \mathsf{c}' : \mathsf{s}_i; \emptyset \vdash \mathsf{e}_i[{}^{\mathsf{c}'}\!/\mathsf{c}] : \mathsf{t}'_i\ (i \in \{1,2\})$.

(12) implies $\Sigma'_{i1}, \mathsf{c}' : \mathsf{s}_i; \emptyset \preceq \Sigma_{i1}, \mathsf{c}' : \mathsf{s}_i; \mathscr{S}_i$ being c' fresh, and then by (13) and Lemma 7.2 we derive:

(14) $\Gamma; \Sigma_{i1}, \mathsf{c}' : \mathsf{s}_i; \mathscr{S}_i \vdash \mathsf{e}_i[{}^{\mathsf{c}'}\!/\mathsf{c}] : \mathsf{t}'_i\ (i \in \{1,2\})$.

From (14), (8) and Lemma 7.4, we obtain (notice that $(\Sigma_{i1}, \mathsf{c}' : \mathsf{s}_i) \circ \Sigma_{i2}$ is defined by (6) since c' is fresh):

(15) $\Gamma; (\Sigma_{i1}, \mathsf{c}' : \mathsf{s}_i) \circ \Sigma_{i2}; \mathscr{S}_i \vdash E_i[\mathsf{e}_i[{}^{\mathsf{c}'}\!/\mathsf{c}]] : \mathsf{t}'_i\ (i \in \{1,2\})$.

Applying rules **Start** and **Par** on (15), and also the fact that $(\Sigma_{11}, \mathsf{c}' : \mathsf{s}_1) \circ \Sigma_{12} \| (\Sigma_{21}, \mathsf{c}' : \mathsf{s}_2) \circ \Sigma_{22} = \Sigma, \mathsf{c}' : \updownarrow$, we obtain

(16) $\Gamma; \Sigma, \mathsf{c}' : \updownarrow \vdash E_1[\mathsf{e}_1[{}^{\mathsf{c}'}\!/\mathsf{c}]] \,|\, E_2[\mathsf{e}_2[{}^{\mathsf{c}'}\!/\mathsf{c}]] : \mathsf{thread}$.

Take

(17) $\Sigma' = \Sigma, \mathsf{c}' : \updownarrow$.

This gives, trivially that:

(18) $\Sigma \subseteq \Sigma'$.

Also, from (1) and (5) we obtain

(19) $\Gamma; \Sigma' \vdash h'$.

The case concludes by considering (16), (17), (18) and (19).

Rule **ComS**. Therefore, we have that

(0) $P = E_1[\mathsf{c}.\mathsf{send}(\mathsf{v})] \,|\, E_2[\mathsf{c}.\mathsf{receive}]$,

(1) $P' = E_1[\mathsf{null}] \,|\, E_2[\mathsf{v}]$, $h' = h$.

From (0), application of the premises, we obtain that $\Gamma; \Sigma \vdash E_1[\mathsf{c}.\mathsf{send}(\mathsf{v})] \,|\, E_2[\mathsf{c}.\mathsf{receive}] : \mathsf{thread}$, which gives by Lemma A.3(2) and (1) that for some $\Sigma_1, \Sigma_2, \mathscr{S}_1, \mathscr{S}_2, \mathsf{t}_1, \mathsf{t}_2$:

(2) $\Gamma; \Sigma_1; \mathscr{S}_1 \vdash E_1[\mathsf{c}.\mathsf{send}(\mathsf{v})] : \mathsf{t}_1$,

(3) $\Gamma;\Sigma_2;\mathscr{S}_2 \vdash E_2[\mathsf{c.receive}]:\mathsf{t}_2$,
(4) $\Sigma = \Sigma_1 \| \Sigma_2$.
By application of premises, we obtain that $\Gamma;\Sigma \vdash h$.

The aim of the next steps is to obtain types for $\mathsf{c.receive}$ *and* $\mathsf{c.send(v)}$, *and for* $E_1[\mathsf{x}]$ *and* $E_2[\mathsf{x}]$.
From (2) and Lemma 7.3, we obtain for some $\Sigma_{11},\Sigma_{12},\mathsf{t}'_1$:
(5) $\Gamma;\Sigma_{11};\mathscr{S}_1 \vdash \mathsf{c.send(v)}:\mathsf{t}'_1$,
(6) $\Gamma,\mathsf{x}:\mathsf{t}'_1;\Sigma_{12};\mathscr{S}_1 \vdash E_1[\mathsf{x}]:\mathsf{t}_1$,
(7) $\Sigma_1 = \Sigma_{11}\circ\Sigma_{12}$.
From (5) and Lemmas A.2(3) and A.1(2), (3), (4), (5), we obtain for some t''_1:
(8) $\Gamma;\emptyset;\emptyset \vdash \mathsf{v}:\mathsf{t}''_1$,
(9) $\{\mathsf{c}:!\mathsf{t}''_1\};\{\mathsf{c}\} \preceq \Sigma_{11};\mathscr{S}_1$.
From (3), and Lemma 7.3, we obtain for some $\Sigma_{21},\Sigma_{22},\mathsf{t}'_2$:
(10) $\Gamma;\Sigma_{21};\mathscr{S}_2 \vdash \mathsf{c.receive}:\mathsf{t}'_2$,
(11) $\Gamma,\mathsf{x}:\mathsf{t}'_2;\Sigma_{22};\mathscr{S}_2 \vdash E_2[\mathsf{x}]:\mathsf{t}_2$,
(12) $\Sigma_2 = \Sigma_{21}\circ\Sigma_{22}$.
From (10), by Lemma A.2(2), we obtain:
(13) $\{\mathsf{c}:?\mathsf{t}'_2\};\{\mathsf{c}\} \preceq \Sigma_{21};\mathscr{S}_2$.

The aim of the next steps is to obtain types for $E_1[\mathsf{null}]$ *and* $E_2[\mathsf{v}]$.
From (9), and (7), which gives that $\Sigma_{11}\circ\Sigma_{12}$ is defined, and Lemma A.5(7a) and (6b), we obtain:
(14) $\Sigma_{11}[\mathsf{c}\mapsto\varepsilon]\circ\Sigma_{12}$ is defined,
(15) $\{\mathsf{c}:\varepsilon\};\emptyset \preceq \Sigma_{11}[\mathsf{c}\mapsto\varepsilon];\mathscr{S}_1$.
By rules **Null**, and **WeakES** we obtain $\Gamma;\{\mathsf{c}:\varepsilon\};\emptyset \vdash \mathsf{null}:\mathsf{t}'_1$. Then, by (15) and Lemma 7.2 we obtain:
(16) $\Gamma;\Sigma_{11}[\mathsf{c}\mapsto\varepsilon];\mathscr{S}_1 \vdash \mathsf{null}:\mathsf{t}'_1$.
From (6), (14) , (16), and Lemma 7.4, we obtain:
(17) $\Gamma;\Sigma_{11}[\mathsf{c}\mapsto\varepsilon]\circ\Sigma_{12};\mathscr{S}_1 \vdash E_1[\mathsf{null}]:\mathsf{t}_1$.
From (4), (7), (12), (9), (13), and Lemma A.5(8a) we obtain that $\mathsf{t}''_1 = \mathsf{t}'_2$. Therefore, with (8) and (11) we obtain
(18) $\Gamma;\Sigma_{22};\mathscr{S}_2 \vdash E_2[\mathsf{v}]:\mathsf{t}_2$.
Furthermore, from (13), (12) and Lemma A.5(7c) we can deduce that $\Sigma_{22};\mathscr{S}_2 \preceq \Sigma_{21}[\mathsf{c}\mapsto\varepsilon]\circ\Sigma_{22};\mathscr{S}_2$. From that, (18) and application of Lemma 7.4, we obtain:
(19) $\Gamma;\Sigma_{21}[\mathsf{c}\mapsto\varepsilon]\circ\Sigma_{22};\mathscr{S}_2 \vdash E_2[\mathsf{v}]:\mathsf{t}_2$.
Furthermore, from (4), (7), (12), (9), (13) and Lemma A.5(8b), we obtain:
(20) $\Sigma_{11}[\mathsf{c}\mapsto\varepsilon]\circ\Sigma_{12}\|\Sigma_{21}[\mathsf{c}\mapsto\varepsilon]\circ\Sigma_{22} = \Sigma_{11}\circ\Sigma_{12}\|\Sigma_{21}\circ\Sigma_{22}$.
*The case concludes by applying rules **Par** and **Start** to (17) and (19) taking (20) into account.*

Rule **ComSS**. We have:
(0) $P = E_1[\mathsf{c.sendS(c')}] \mid E_2[\mathsf{c.receiveS(x)\{e\}}]$,
(1) $P' = E_1[\mathsf{null}] \mid \mathsf{e}[{}^{\mathsf{c}}\!/\!_{\mathsf{x}}] \mid E_2[\mathsf{null}]$,
(2) $h' = h$,
(3) $\Gamma;\,\Sigma \vdash P:\mathsf{thread}$.

From the premises, and using Lemma A.3(2) and (1), we obtain for some $\Sigma_1, \Sigma_2, \mathscr{S}_1, \mathscr{S}_2, \mathsf{t}_1, \mathsf{t}_2$:

(4) $\Gamma;\, \Sigma_1; \mathscr{S}_1 \vdash E_1[\mathsf{c}.\mathsf{sendS}\,(\mathsf{c}')] : \mathsf{t}_1$,

(5) $\Gamma;\, \Sigma_2; \mathscr{S}_2 \vdash E_2[\mathsf{c}.\mathsf{receiveS}\,(\mathsf{x})\{\mathsf{e}\}] : \mathsf{t}_2$,

(6) $\Sigma = \Sigma_1 \| \Sigma_2$.

The aim of the next steps is to obtain types for $E_1[\mathsf{null}]$, $E_2[\mathsf{null}]$, and $\mathsf{e}\,[{}^{\mathsf{c}}\!/_{\mathsf{x}}]$.

From (4), using Lemma 7.3 and Lemma A.2(5) we obtain for some $\Sigma_{11}, \Sigma_{12}, \mathsf{t}'_1, \eta \neq \varepsilon.\mathsf{end}$:

(7) $\Gamma;\, \Sigma_{11}; \mathscr{S}_1 \vdash \mathsf{c}.\mathsf{sendS}\,(\mathsf{c}') : \mathsf{t}'_1$,

(8) $\Gamma, \mathsf{y} : \mathsf{t}'_1;\, \Sigma_{12}; \mathscr{S}_1 \vdash E_1[\mathsf{y}] : \mathsf{t}_1$,

(9) $\Sigma_1 = \Sigma_{11} \circ \Sigma_{12}$,

(10) $\mathsf{t}'_1 = \mathit{Object}$,

(11) $\{\mathsf{c} :!(\eta), \mathsf{c}' : \eta\}; \{\mathsf{c}\} \preceq \Sigma_{11}; \mathscr{S}_1$.

(11) and Lemma A.5(5a) imply

(12) $\{\mathsf{c}' : \eta\}; \emptyset \preceq \Sigma_{11} \backslash \mathsf{c}; \mathscr{S}_1$,

which gives by $\eta \neq \varepsilon.\mathsf{end}$ and Lemma A.5(6a)

(13) $\Sigma_{11} = \Sigma'_{11}, \mathsf{c}' : \theta$ where $\theta \in \{\eta, \mathsf{begin}.\eta\}$.

(13) and (9) imply by Lemma A.5(4a)

(14) $\Sigma'_{11} \circ \Sigma_{12}$ defined.

(11) and (13) imply by Lemma A.5(5b)

(15) $\{\mathsf{c} :!(\eta)\}; \{\mathsf{c}\} \preceq \Sigma'_{11}; \mathscr{S}_1$.

Using rules **Null**, **WeakES** we obtain:

(16) $\Gamma;\, \{\mathsf{c} : \varepsilon\};\, \emptyset \vdash \mathsf{null} : \mathsf{t}'_1$.

By (15), (14), and Lemma A.5(7a) and (6b) respectively we have:

(17) $\Sigma'_{11}[\mathsf{c} \mapsto \varepsilon] \circ \Sigma_{12}$ defined,

(18) $\{\mathsf{c} : \varepsilon\}; \emptyset \preceq \Sigma'_{11}[\mathsf{c} \mapsto \varepsilon]; \mathscr{S}_1$.

From (18), (16), and using Lemma 7.2 we obtain:

(19) $\Gamma;\, \Sigma'_{11}[\mathsf{c} \mapsto \varepsilon];\, \mathscr{S}_1 \vdash \mathsf{null} : \mathsf{t}'_1$.

From (8), (19), (17) and Lemma 7.4, we obtain:

(20) $\Gamma; \Sigma'_{11}[\mathsf{c} \mapsto \varepsilon] \circ \Sigma_{12}; \mathscr{S}_1 \vdash E_1[\mathsf{null}] : \mathsf{t}_1$.

From (5), using Lemma 7.3 and Lemma A.2(4) we obtain for some $\Sigma_{21}, \Sigma_{22}, \mathsf{t}'_2, \eta' \neq \varepsilon.\mathsf{end}$:

(21) $\Gamma;\, \Sigma_{21}; \mathscr{S}_2 \vdash \mathsf{c}.\mathsf{receiveS}\,(\mathsf{x})\{\mathsf{e}\} : \mathsf{t}'_2$

(22) $\Gamma, \mathsf{y} : \mathsf{t}'_2;\, \Sigma_{22}; \mathscr{S}_2 \vdash E_2[\mathsf{y}] : \mathsf{t}_2$,

(23) $\Sigma_2 = \Sigma_{21} \circ \Sigma_{22}$,

(24) $\mathsf{t}'_2 = \mathit{Object}$,

(25) $\{\mathsf{c} :?(\eta')\}; \{\mathsf{c}\} \preceq \Sigma_{21}; \mathscr{S}_2$,

(26) $\Gamma \backslash \mathsf{x};\, \{\mathsf{x} : \eta'\}; \{\mathsf{x}\} \vdash \mathsf{e} : \mathsf{t}'$.

Similarly and simpler than the proof of (20) we can show:

(27) $\Gamma; \Sigma_{21}[\mathsf{c} \mapsto \varepsilon] \circ \Sigma_{22}; \mathscr{S}_2 \vdash E_2[\mathsf{null}] : \mathsf{t}_2$.

From (26) using Lemma A.8(3) we obtain:

(28) $\Gamma;\, \{\mathsf{c}' : \eta'\}; \{\mathsf{c}'\} \vdash \mathsf{e}\,[{}^{\mathsf{c}}\!/_{\mathsf{x}}] : \mathsf{t}'$.

The aim of the next steps is to show that the type of c used to type $\mathsf{c}.\mathsf{sendS}\,(\mathsf{c}')$ is dual to that used to type $\mathsf{c}.\mathsf{receiveS}\,(\mathsf{x})\{\mathsf{e}\}$, and that the parallel composition of the session environments used to type $E_1[\mathsf{null}]$, $E_2[\mathsf{null}]$, and $\mathsf{e}\,[{}^{\mathsf{c}}\!/_{\mathsf{x}}]$ is the same as Σ.

(13) and (9) imply by Lemma A.5(4b)

(29) $\Sigma_{11}\circ\Sigma_{12} = \mathsf{c}' : \theta \| \Sigma'_{11}\circ\Sigma_{12}$.

(6), (9), (23) and (29) imply:

(30) $\Sigma'_{11}\circ\Sigma_{12}\|\Sigma_{21}\circ\Sigma_{22}$ defined.

From (30), (15), (25) by Lemma A.5(8a) we get:

(31) $!(\eta) = \overline{?(\eta')}$,

which implies:

(32) $\eta = \eta'$.

Again from (30), (15), (25) by Lemma A.5(8b) we get:

(33) $\Sigma'_{11}[\mathsf{c} \mapsto \varepsilon]\circ\Sigma_{12}\|\Sigma_{21}[\mathsf{c} \mapsto \varepsilon]\circ\Sigma_{22} = \Sigma'_{11}\circ\Sigma_{12}\|\Sigma_{21}\circ\Sigma_{22}$.

(6), (9), (29), (23), and (33) imply:

(34) $\Sigma = \{\mathsf{c}' : \theta\}\|\Sigma'_{11}[\mathsf{c} \mapsto \varepsilon]\circ\Sigma_{12}\|\Sigma_{21}[\mathsf{c} \mapsto \varepsilon]\circ\Sigma_{22}$.

From (28), (13) and (32), possibly using **WeakB** and **Weak**, we derive:

(35) $\Gamma; \{\mathsf{c}' : \theta\}; \{\mathsf{c}'\} \vdash \mathsf{e}[{}^{\mathsf{c}'}\!/\!_{\mathsf{x}}] : \mathsf{t}'$.

The case concludes by applying rules **Par** *and* **Start** *to (20), (27), (35) by taking into account (34).*

Rule **ComSWhile**.Then, we have that:

(0) $P \;=\; E_1[\mathsf{c}.\mathsf{sendWhile}(\mathsf{e}_1)\{\mathsf{e}_2\}] \,|\, E_2[\mathsf{c}.\mathsf{receiveWhile}\{\mathsf{e}_3\}]$,

(1) $h' = h$,

(2) $P' \;=\; E_1[\mathsf{e}_5] \;|\; E_2[\mathsf{e}_6]$,

where we are using the shorthands:

(3) $\mathsf{e}_5 = \mathsf{c}.\mathsf{sendIf}(\mathsf{e}_1)\{\mathsf{e}_2; \mathsf{c}.\mathsf{sendWhile}(\mathsf{e}_1)\{\mathsf{e}_2\}\}\{\mathsf{null}\}$,

(4) $\mathsf{e}_6 = \mathsf{c}.\mathsf{receiveIf}\{\mathsf{e}_3; \mathsf{c}.\mathsf{receiveWhile}\{\mathsf{e}_3\}\}\{\mathsf{null}\}$.

From premises, (0) and Lemma A.3(2), and A.3(1) we obtain for some $\Sigma_1, \Sigma_2, \mathscr{S}_1, \mathscr{S}_2, \mathsf{t}_1, \mathsf{t}_2$:

(5) $\Gamma; \Sigma \vdash h$,

(6) $\Sigma = \Sigma_1\|\Sigma_2$,

(7) $\Gamma; \Sigma_1; \mathscr{S}_1 \vdash E_1[\mathsf{c}.\mathsf{sendWhile}(\mathsf{e}_1)\{\mathsf{e}_2\}] : \mathsf{t}_1$,

(8) $\Gamma; \Sigma_2; \mathscr{S}_2 \vdash E_2[\mathsf{c}.\mathsf{receiveWhile}\{\mathsf{e}_3\}] : \mathsf{t}_2$.

From (7), (8) applying Lemma 7.3, there exist Σ_{11}, Σ_{12}, Σ_{21}, Σ_{22}, t'_1, t'_2 so that:

(9) $\Sigma_1 = \Sigma_{11}\circ\Sigma_{12}, \quad \Sigma_2 = \Sigma_{21}\circ\Sigma_{22}$,

(10) $\Gamma; \Sigma_{11}; \mathscr{S}_1 \vdash \mathsf{c}.\mathsf{sendWhile}(\mathsf{e}_1)\{\mathsf{e}_2\} : \mathsf{t}'_1$,

(11) $\Gamma, \mathsf{x} : \mathsf{t}'_1; \Sigma_{12}; \mathscr{S}_1 \vdash E_1[\mathsf{x}] : \mathsf{t}_1$,

(12) $\Gamma; \Sigma_{21}; \mathscr{S}_2 \vdash \mathsf{c}.\mathsf{receiveWhile}\{\mathsf{e}_3\} : \mathsf{t}'_2$,

(13) $\Gamma, \mathsf{x} : \mathsf{t}'_2; \Sigma_{22}; \mathscr{S}_2 \vdash E_2[\mathsf{x}] : \mathsf{t}_2$.

The aim of the next steps is to find types for e_2, *and* e_5, *and* $E_1[\mathsf{e}_5]$.

From (10), and Lemma A.2(9), we obtain for some π_1:

(14) $\{\mathsf{c} : !\langle\pi_1\rangle^*\}; \{\mathsf{c}\} \preceq \Sigma_{11}; \mathscr{S}_1$,

(15) $\Gamma; \emptyset; \emptyset \vdash \mathsf{e}_1 : \mathsf{bool}$,

(16) $\Gamma; \{\mathsf{c} : \pi_1\}; \{\mathsf{c}\} \vdash \mathsf{e}_2 : \mathsf{t}'_1$.

We will be using π_2 as a shorthand defined as follows:

(17) $\pi_2 = !\langle\pi_1.!\langle\pi_1\rangle^*, \varepsilon\rangle$.

By application of type rules **Null**, **Weak**, **SendIf**, **Seq**, **SendWhile** on (15) and (16), and using the shorthands (3) and (17) we obtain:

(18) $\Gamma;\{\mathsf{c}:\pi_2\};\{\mathsf{c}\}\vdash \mathsf{e}_5:\mathsf{t}'_1$.
From (14), and application of Lemma A.5(6c), we obtain that:
(20) $\{\mathsf{c}:\pi_2\};\{\mathsf{c}\}\preceq \Sigma_{11}[\mathsf{c}\mapsto \mathsf{begin}.\pi_2];\mathscr{S}_1$.
By application of Lemma 7.2 on (18) and (20), we obtain:
(21) $\Gamma;\Sigma_{11}[\mathsf{c}\mapsto \mathsf{begin}.\pi_2];\mathscr{S}_1\vdash \mathsf{e}_5:\mathsf{t}'_1$.
By (9), we have that $\Sigma_{11}\circ\Sigma_{12}$ is defined, and therefore, by (14) and application of Lemma A.5(7b) we also obtain that $\Sigma_{11}[\mathsf{c}\mapsto \mathsf{begin}.\pi_2]\circ\Sigma_{12}$ is defined. Therefore by applying Lemma 7.4 on (11) and (21) we obtain:
(22) $\Gamma;\Sigma_{11}[\mathsf{c}\mapsto \mathsf{begin}.\pi_2]\circ\Sigma_{12};\mathscr{S}_1\vdash E_1[\mathsf{e}_5]:\mathsf{t}'_1$.
The aim of the next steps is to find types for e_3*, and* e_6*, and* $E_2[\mathsf{e}_6]$.
By arguments similar to those used to get (14) and (16), we obtain from (12) for some π_3:
(23) $\{\mathsf{c}:?\langle\pi_3\rangle^*\};\{\mathsf{c}\}\preceq \Sigma_{21};\mathscr{S}_2$,
(24) $\Gamma;\{\mathsf{c}:\pi_3\};\{\mathsf{c}\}\vdash \mathsf{e}_3:\mathsf{t}'_2$.
We use the shorthand
(25) $\pi_4 = ?\langle\pi_3.?\langle\pi_3\rangle^*,\varepsilon\rangle$.
Then, by arguments similar to those used to get (22), we obtain that:
(26) $\Gamma;\Sigma_{21}[\mathsf{c}\mapsto \mathsf{begin}.\pi_4]\circ\Sigma_{22};\mathscr{S}_1\vdash E_2[\mathsf{e}_6]:\mathsf{t}_2$.
The aim of the next steps is to show that the type of c *used to type* e_5 *is dual to that used to type* e_6*, and that the parallel composition of the session environments used to type* $E_1[\mathsf{e}_5]$ *and* $E_2[\mathsf{e}_6]$ *is the same as* Σ*.*
Because of (14), (23), being $\Sigma_{11}\circ\Sigma_{12}\|\Sigma_{21}\circ\Sigma_{22}$ defined, and by Lemma A.5(8a) we obtain that:
(27) $\pi_1=\overline{\pi_3}$,
which implies:
(28) $\pi_2=\overline{\pi_4}$.
Therefore, using (14), (23), being $\Sigma_{11}\circ\Sigma_{12}\|\Sigma_{21}\circ\Sigma_{22}$ defined, and by Lemma A.5(8c) we obtain that:
(29) $\Sigma_{11}[\mathsf{c}\mapsto \mathsf{begin}.\pi_2]\circ\Sigma_{12}\parallel\Sigma_{21}[\mathsf{c}\mapsto \mathsf{begin}.\pi_4]\circ\Sigma_{22} \;=\; \Sigma_{11}\circ\Sigma_{12}\parallel\Sigma_{21}\circ\Sigma_{22}$
The case concludes by applying rules **Par** *and* **Start** *to (22), (26), and taking into account (29), (9), and (6).*

B Proof of Theorems 7.10 and 7.13

We start from basic properties of live channels. In Section 3 we used the notion of live channel in an informal way; here we need to give a precise, formal definition.

Definition B.1 *A channel* c *is* live *in a process* P *if* $P=C[\mathsf{e}]$*, the expression* e *is either a session expression with subject* c *or a command sending the channel* c*, and there are no contexts* $C_1[\,]$*,* $C_2[\,]$ *such that* $P\equiv C_1[\mathsf{connect}\ \mathsf{c}\ \mathsf{s}\,\{C_2[\mathsf{e}]\}]$.

For example, `c1` and `c2` are live in `o.f;c1.send(3);c2.receive`; also, `c1` and `c2` are live in `c2.sendS(c1){...}`; but `c1` is *not* live in `connect c1!int{c1.send(3)}`; finally, a channel variable is never live.

Lemma B.2 *(1) If* $\Gamma;\Sigma \vdash \mathsf{e} : \mathsf{thread}$ *and* c *is live in* e*, then* $c : \theta \in \Sigma$ *for some* $\theta \notin \{\varepsilon, \varepsilon.\mathsf{end}, \mathsf{begin}.\varepsilon, \mathsf{begin}.\varepsilon.\mathsf{end}, \updownarrow\}$.

(2) Assume P_0 *is* Γ*-initial and* $P_0, h_\Gamma \twoheadrightarrow P, h$*. Then* $\Gamma;\Sigma \vdash P;h$ *for some* Γ, Σ*, such that all predicates in* Σ *are* $\updownarrow$.

Proof (1) By definition $\mathsf{e} \equiv C[\mathsf{e}']$ where e' is either a session expression with subject c or a command sending the channel c. In the first case by Lemma A.2(2), (3), (4), (5), (6), (7), (8), and (9) and in the second case by Lemma A.2(5) the session environment for typing e' must contain a premise with subject c and predicate different from $\varepsilon, \varepsilon.\mathsf{end}, \mathsf{begin}.\varepsilon, \mathsf{begin}.\varepsilon.\mathsf{end}, \mathsf{begin}.\varepsilon.\mathsf{end}, \updownarrow$. The proof is then by structural induction on $C[\,]$ taking into account that $C[\,] \neq C_1[\mathsf{connect}\ \mathsf{c}\ \mathsf{s}\,\{C_2[\,]\}]$ for all $C_1[\,], C_2[\,]$.

(2) P_0 initial implies that it is typed with the empty session environment. Looking at the proof of the Subject Reduction Theorem for threads it is clear that **Connect** is the only rule in which one needs to add premises to the session environments. Moreover the added premise is of the shape $\mathsf{c} :\updownarrow$ where c is the fresh created channel.

Lemma B.3 *(1) If* $E[\mathsf{spawn}\,\{\ \mathsf{e}\ \}]$ *is well typed, then no live channel occurs both in* $E[\,]$ *and in* e.

(2) If $E[\mathsf{c}.\mathsf{receiveS}\,(\mathsf{x})\{\mathsf{e}\}]$ *is well typed, then no live channel occurs in* e.

Proof (1) If $E[\mathsf{spawn}\,\{\ \mathsf{e}\ \}]$ is well typed, then by Lemma 7.3 there are Γ, Σ, $\mathscr{S}$, t, Σ_1, Σ_2, t' such that $\Gamma;\Sigma;\mathscr{S} \vdash E[\mathsf{spawn}\,\{\ \mathsf{e}\ \}] : \mathsf{t}$ and $\Gamma;\Sigma_1;\mathscr{S} \vdash \mathsf{spawn}\,\{\ \mathsf{e}\ \} : \mathsf{t}'$ and $\Gamma, \mathsf{x} : \mathsf{t}';\Sigma_2;\mathscr{S} \vdash E[\mathsf{x}] : \mathsf{t}$ and $\Sigma = \Sigma_1 \circ \Sigma_2$. By Lemma A.1(15), there are $\Sigma';\mathscr{S}'$ such that $\Sigma';\mathscr{S}' \preceq \Sigma_1;\mathscr{S}$ and $\mathit{ended}(\Sigma')$. By Lemma B.2(1) a live channel in e must occur in the domain of Σ', and therefore it cannot occur in the domain of Σ_2. Because $\Sigma_1 \circ \Sigma_2$ is defined, and from $\mathit{ended}(\Sigma')$, it follows that no live channel in e can occur in $E[\mathsf{x}]$.

(2) If $E[\mathsf{c}.\mathsf{receiveS}\,(\mathsf{x})\{\mathsf{e}\}]$ is well typed, then by Lemma 7.3 there are Γ, Σ, $\mathscr{S}$, t, $\Sigma_1, \Sigma_2, \mathsf{t}'$ such that $\Gamma;\Sigma;\mathscr{S} \vdash E[\mathsf{c}.\mathsf{receiveS}\,(\mathsf{x})\{\mathsf{e}\}] : \mathsf{t}$ and $\Gamma;\Sigma_1;\mathscr{S} \vdash \mathsf{c}.\mathsf{receiveS}\,(\mathsf{x})\{\mathsf{e}\} : \mathsf{t}'$ and $\Gamma, \mathsf{y} : \mathsf{t}';\Sigma_2;\mathscr{S} \vdash E[\mathsf{y}] : \mathsf{t}$ and $\Sigma = \Sigma_1 \circ \Sigma_2$. By Lemma A.2(4), there is η such that $\{\mathsf{c} : \eta\};\{\mathsf{c}\} \preceq \Sigma_1;\mathscr{S}$, which implies the thesis by Lemma B.2(1).

Lemma 7.8 *Assume* P_0 *is* Γ*-initial and* $P_0, h_\Gamma \twoheadrightarrow P, h$*. Then each live channel occurs exactly in two threads in* P.

Proof By induction on $\twoheadrightarrow$. The base case is trivial, since there are no live channels in the typing environments of an initial thread. The **Connect** rule creates a new live channel in two different threads. By Lemma B.3(1) the live channels which occur in $E_1[\mathsf{spawn}\,\{\ \mathsf{e}\ \}]$ are split between $E_1[\mathsf{null}]$ and e. By Lemma B.3(2) all the live channels which occur in $E_1[\mathsf{c}.\mathsf{receiveS}(\mathsf{x})\{\mathsf{e}\}]$ are in $E_1[\mathsf{null}]$.

We say e is *irreducible* if $\mathsf{e} \not\longrightarrow$. The key in showing progress is the natural correspondence between irreducible session expressions and partial session types formalised in the following definition.

Definition B.4 *Define $\propto$ between irreducible session expressions and parts of session types as follows:*

$$\mathsf{c}.\mathsf{receive} \propto ?\mathsf{t} \quad \mathsf{c}.\mathsf{send}(\mathsf{v}) \propto !\mathsf{t} \quad \mathsf{c}.\mathsf{receiveS}(\mathsf{x})\{\mathsf{e}\} \propto ?(\eta) \quad \mathsf{c}.\mathsf{sendS}(\mathsf{c}') \propto !(\eta)$$
$$\mathsf{c}.\mathsf{receiveIf}\{\mathsf{e}_1\}\{\mathsf{e}_2\} \propto ?\langle\rho_1,\rho_2\rangle \quad \mathsf{c}.\mathsf{sendIf}(\mathsf{v})\{\mathsf{e}_1\}\{\mathsf{e}_2\} \propto !\langle\rho_1,\rho_2\rangle$$
$$\mathsf{c}.\mathsf{receiveWhile}\{\mathsf{e}\} \propto ?\langle\pi\rangle^* \quad \mathsf{c}.\mathsf{sendWhile}(\mathsf{v})\{\mathsf{e}\} \propto !\langle\pi\rangle^*$$

Notice, that the relation $\mathsf{e} \propto \pi$ reflects the "shape" of the session, rather than the precise types involved. For example, $\mathsf{e} \propto ?\mathsf{t}$ implies $\mathsf{e} \propto ?\mathsf{t}'$ for any type t'.

The following proposition is immediate from the definition of $\propto$.

Proposition B.5 *If $\mathsf{e} \propto \pi$ and $\mathsf{e}' \propto \overline{\pi}$, then e and e' are dual of each other.*

Using the Generation Lemmas and Lemma 7.3 we can show the correspondence between an irreducible session expression inside an evaluation context and the type of the live channel which is the subject of the expression.

Lemma B.6 *Let e be an irreducible session expression with subject c and $\Gamma;\Sigma \vdash E[\mathsf{e}]:\mathsf{thread}$. Then $\mathsf{e} \propto \pi$ and $\Sigma(\mathsf{c}) \in \{\pi, \mathsf{begin}.\pi, \pi.\mathsf{end}, \pi.\rho, \mathsf{begin}.\pi.\rho\}$ for some π,ρ.*

Proof By Lemmas A.3(1) and 7.3 we get $\Gamma;\Sigma';\mathscr{S} \vdash \mathsf{e}:\mathsf{t}'$ for some $\Sigma' \preceq \Sigma$ and t'. By Lemma A.1(2), (3), (4), (5) the session environments in the typing of values are always $\succeq \emptyset$. Then from Lemma A.2(2), (3), (4), (5), (6), (7), (8), (9), we get $\mathsf{e} \propto \pi$ and $\Sigma(\mathsf{c}) \in \{\pi, \mathsf{begin}.\pi, \pi.\mathsf{end}, \pi.\rho, \mathsf{begin}.\pi.\rho\}$ for some π,ρ.

The following three lemmas state a relationship between hot sets and subjects of session expressions and of method calls. In these lemmas we consider typing of initial threads so that rule **WeakB** has never been applied. In fact rule **WeakB** introduces a session type starting by begin in the session environment, which can never be discharged in order to obtain an empty session environment.

Lemma B.7 *Let e be a session expression or a method call with subject u and rule* **WeakB** *be never applied in the considered typings.*

(1) The expression e must be typed with hot set $\{\mathsf{u}\}$.
(2) If $\Gamma;\Sigma;\mathscr{S} \vdash C[\mathsf{e}]:\mathsf{t}$, and $\mathscr{S} \neq \{\mathsf{u}\}$, then either $C[\,] = C_1[\mathsf{connect}\ \mathsf{u}\ \mathsf{s}\{C_2[\,]\}]$ or $C[\,] = C_1[\mathsf{u}'.\mathsf{receiveS}(\mathsf{x})\{C_2[\,]\}]$ and $\mathsf{u} = \mathsf{x}$, i.e., e occurs in the body either

of a $\mathsf{connect}$ *or of a* $\mathsf{receiveS}$ *expression, and in the last case* $\mathsf{u} = \mathsf{x}$.

Proof (1) Immediate from Lemmas A.2(2), (3), (4), (5), (6), (7), (8), (9) and A.1(12).

(2) From (1) we get that e must be typed with hot set $\{\mathsf{u}\}$. Then the claim follows by observing that the only typing rules different from **WeakB** which change non-empty hot sets are **Conn**, **ReceiveS**.

Notice that Lemma B.7 does not hold if we allow rule **WeakB**, since for example we can derive $\emptyset; \{\mathsf{c} : \mathsf{begin.!bool}\}; \emptyset \vdash \mathsf{c.send}\,(\mathsf{true}) : \mathit{Object}$.

Lemma 7.9 *If* $\mathsf{connect}\ \mathsf{u}\ \mathsf{s}\{\mathsf{e}\}$ *is an expression which is well typed without using rule* **WeakB** *and* $\mathsf{e} = C[\mathsf{e}']$, *where* e' *is a session expression or a method call with subject* u', *then one of the following conditions holds:*

(1) $\mathsf{u} = \mathsf{u}'$;
(2) $C[\,] = C_1[\mathsf{connect}\ \mathsf{u}'\mathsf{s}'\{C_2[\,]\}]$;
(3) $C[\,] = C_1[\mathsf{u}''.\mathsf{receiveS}\,(\mathsf{x})\{C_2[\,]\}]$ *and* $\mathsf{u}' = \mathsf{x}$.

Proof From Lemma B.7(1) we get that e' must be typed with hot set $\{\mathsf{u}'\}$. From the typing rule **Conn** we get that e must be typed with hot set $\{\mathsf{u}\}$. So we conclude using Lemma B.7(2).

Lemma B.8 *(1) If* $\mathsf{t}\ \mathsf{m}\ (\widetilde{\mathsf{t}\ \mathsf{x}}, \widetilde{\rho\ \mathsf{y}})\ \{\mathsf{e}\}$ *is ok in some class,* $\mathsf{mtype}(\mathsf{m}, C) = \mathsf{t}_1, \ldots, \mathsf{t}_n, \rho_1, \ldots, \rho_m \xrightarrow{\ominus} \mathsf{t}$ *and* $\mathsf{e} = C[\mathsf{e}']$, *where* e' *is a session expression or a method call with subject* u, *then one of the following conditions holds:*
(a) $C[\,] = C_1[\mathsf{connect}\ \mathsf{u}\ \mathsf{s}\{C_2[\,]\}]$;
(b) $C[\,] = C_1[\mathsf{u}'.\mathsf{receiveS}\,(\mathsf{x})\{C_2[\,]\}]$ *and* $\mathsf{u} = \mathsf{x}$.
(2) If $\mathsf{t}\ \mathsf{m}\ (\widetilde{\mathsf{t}\ \mathsf{x}}, \widetilde{\rho\ \mathsf{y}})\ \{\mathsf{e}\}$ *is ok in some class,* $\mathsf{mtype}(\mathsf{m}, C) = \mathsf{t}_1, \ldots, \mathsf{t}_n, \rho_1, \ldots, \rho_m \xrightarrow{\oplus} \mathsf{t}$ *and* $\mathsf{e} = C[\mathsf{e}']$, *where* e' *is a session expression or a method call with subject* u, *then one of the following conditions holds:*
(a) $\mathsf{u} = \mathsf{y}_1$;
(b) $C[\,] = C_1[\mathsf{connect}\ \mathsf{u}\ \mathsf{s}\{C_2[\,]\}]$;
(c) $C[\,] = C_1[\mathsf{u}'.\mathsf{receiveS}\,(\mathsf{x})\{C_2[\,]\}]$ *and* $\mathsf{u} = \mathsf{x}$.

Proof The proof is similar to that of Lemma 7.9, taking into account that rules **MMinus** – **ok** and **MPlus** – **ok** do not allow to use rule **WeakB** in typing e and that these rules require respectively the empty set and the set $\{\mathsf{y}_1\}$ as hot sets of e.

The following definition shows the order in which expressions are reduced.

Definition B.9 *Let* e *be an expression and* e_1, e_2 *be two subexpressions of* e
e_1 precedes e_2 *in* e *iff* $\mathsf{e} = C[\mathsf{e}']$ *and* $\mathsf{e}' = E[\mathsf{e}_1] = C'[\mathsf{e}_2]$
for some contexts $C[\,]$, $E[\,]$ *and* $C'[\,]$.

Notice that any expression precedes itself since we can choose all contexts as the empty one.

In the following we convene that the fresh channels created reducing a thread take successive numbers according to the order of creation, i.e. they are $\mathtt{c}_0, \mathtt{c}_1, \ldots$. This means that if $P, h \longrightarrow\!\!\!\!\rightarrow Q, h' \longrightarrow\!\!\!\!\rightarrow R, h''$ and $\mathtt{c}_i$ is a channel created in the reduction $P, h \longrightarrow\!\!\!\!\rightarrow Q, h'$, and $\mathtt{c}_j$ is a channel created in the reduction $Q, h' \longrightarrow\!\!\!\!\rightarrow R, h''$, then $i < j$. We convene also that the names $\mathtt{c}_0, \mathtt{c}_1, \ldots$ are reserved for live channels.

The following lemma shows that the subject of a session expression inside an evaluation context is always the latest created channel which occurs in the whole expression.

Lemma B.10 *Let P_0 be Γ-initial and $P_0, h_\Gamma \longrightarrow\!\!\!\!\rightarrow \mathtt{e} \,|\, P, h$. If $\mathtt{e}'$ precedes $\mathtt{e}''$ in $\mathtt{e}$, and $\mathtt{e}'$ is a session expression or method call with subject $\mathtt{c}_i$, then $i \geq j$ for all live channels $\mathtt{c}_j$ which occur in $\mathtt{e}'$.*

Proof The proof is by induction on $\longrightarrow$ and by cases on the last applied reduction rule. We only consider some interesting cases.
Let the last applied rule be **Meth**:

$$E_0[\mathtt{o}.\mathsf{m}(\tilde{\mathtt{v}})] \,|\, P', h \longrightarrow E_0[\mathtt{e}_0[\mathtt{o}/\mathtt{this}][\tilde{\mathtt{v}}/\tilde{\mathtt{x}}]] \,|\, P', h$$

since $h(\mathtt{o}) = (C, \ldots)$, $\mathsf{mbody}(\mathsf{m}, C) = (\tilde{\mathtt{x}}, \mathtt{e}_0)$ and $\mathsf{mtype}(\mathsf{m}, C) = \mathtt{t}_1, \ldots, \mathtt{t}_n, \rho_1, \ldots, \rho_m \overset{\circledcirc}{\rightarrow} \mathtt{t}$.
If $\circledcirc = \ominus$, then by Lemma B.8(1) all session expressions or method calls which occur in $\mathtt{e}_0[\mathtt{o}/\mathtt{this}][\tilde{\mathtt{v}}/\tilde{\mathtt{x}}]$ have subjects which cannot be live channels. For the session expressions or method calls which occur in $E_0[\,]$ induction hypothesis applies.
If $\circledcirc = \oplus$ let $\mathtt{c}_l$ be the live channel which is the subject of the method call. By induction $l \geq k$ for all $\mathtt{c}_k$ which occur in $E[\mathtt{o}.\mathsf{m}(\tilde{\mathtt{v}})]$. By Lemma B.8(2) the subjects of all session expressions and method calls inside $\mathtt{e}_0[\mathtt{o}/\mathtt{this}][\tilde{\mathtt{v}}/\tilde{\mathtt{x}}]$ which are live channels are the channel $\mathtt{c}_l$. If $\mathtt{e} = E_0[\mathtt{e}_0[\mathtt{o}/\mathtt{this}][\tilde{\mathtt{v}}/\tilde{\mathtt{x}}]]$, then either $\mathtt{e}'$ and $\mathtt{e}''$ are both sub-expressions of $\mathtt{e}_0[\mathtt{o}/\mathtt{this}][\tilde{\mathtt{v}}/\tilde{\mathtt{x}}]$, or $\mathtt{e}'$ is a sub-expressions of $\mathtt{e}_0[\mathtt{o}/\mathtt{this}][\tilde{\mathtt{v}}/\tilde{\mathtt{x}}]$ and $\mathtt{e}''$ is a sub-expressions of $E_0[\,]$, or $\mathtt{e}'$ and $\mathtt{e}''$ are both sub-expressions of $E_0[\,]$. In the first case $\mathtt{c}_l$ is both the subject of $\mathtt{e}'$ and the only live channel which occurs in $\mathtt{e}''$, in the second case the subject of $\mathtt{e}'$ is $\mathtt{c}_l$ and $l \geq k$ for all the live channels $\mathtt{c}_k$ which occur in $\mathtt{e}''$, and in the third case induction hypothesis applies.
Let the last applied rule be **Connect**:

$$E_1[\mathsf{connect}\ \mathtt{c}\ \mathtt{s}\{\mathtt{e}_1\}] \,|\, E_2[\mathsf{connect}\ \mathtt{c}\ \overline{\mathtt{s}}\{\mathtt{e}_2\}] \,|\, P', h \longrightarrow E_1[\mathtt{e}_1[\mathtt{c}_l/\mathtt{c}]] \,|\, E_2[\mathtt{e}_2[\mathtt{c}_l/\mathtt{c}]] \,|\, P', h :: \mathtt{c}' \quad \mathtt{c}_l \notin h$$

where by construction $l > k$ for all $\mathtt{c}_k$ which occur in h. Notice that $\mathtt{e}_1$ and $\mathtt{e}_2$ have never been reduced by definition of evaluation context, and so they can be typed without using rule **WeakB**. Therefore by Lemma 7.9 the subjects of all session expressions and method calls inside $\mathtt{e}_1[\mathtt{c}_l/\mathtt{c}]$ and $\mathtt{e}_2[\mathtt{c}_l/\mathtt{c}]$ which are live channels are the channel $\mathtt{c}_l$. We can conclude as in previous case.

If the last applied rule is **ComS**:

$$E_1[\mathsf{c}_l.\mathsf{send}\,(\mathsf{v})] \mid E_2[\mathsf{c}_l.\mathsf{receive}] \mid P', h \longrightarrow E_1[\mathsf{null}] \mid E_2[\mathsf{v}] \mid P', h$$

then all session expressions or method calls which occur in $E_1[\mathsf{null}] \mid E_2[\mathsf{v}]$ occur also in $E_1[\,] \mid E_2[\,]$, so induction hypothesis applies.
If the last applied rule is **ComSS**:

$$E_1[\mathsf{c}_l.\mathsf{sendS}\,(\mathsf{c}_k)] \;\mid\; E_2[\mathsf{c}_l.\mathsf{receiveS}\,(\mathsf{x})\{\mathsf{e}_0\}] \mid P', h \longrightarrow E_1[\mathsf{null}] \;\mid\; \mathsf{e}_0[{}^{\mathsf{c}_k}\!/\!_{\mathsf{x}}] \;\mid\; E_2[\mathsf{null}] \mid P', h$$

then no live channel occurs in e_0 by Lemma B.3(2). Therefore if $\mathsf{e} = \mathsf{e}_0[{}^{\mathsf{c}_k}\!/\!_{\mathsf{x}}]$, then c_k is both the subject of e' and the only live channel which occurs in e''. The proof for $E_1[\mathsf{null}]$ and $E_2[\mathsf{null}]$ is as in the case of rule **ComS**.

Now we prove the progress property. The following proof of Theorem 7.10 argues that if the configuration does not contain waiting connects or null pointer errors, but contains an irreducible session expression e_1, then by subject reduction and well-formedness of the session environment, the rest of the thread independently moves or it contains the dual of that irreducible expression, e_2. Then by Lemma B.6, we get $\mathsf{e}_1 \propto \pi$ and $\mathsf{e}_2 \propto \overline{\pi}$. Therefore e_1 and e_2 are session expressions dual of each other and they can communicate.

Theorem 7.10 (Progress) *Assume P_0 is Γ-initial and $P_0, h_\Gamma \longrightarrow\!\!\!\!\rightarrow P, h$. Then one of the following holds.*

- *In P, all expressions are values, i.e., $P \equiv \prod_{0 \leq \mathsf{i} < \mathsf{n}} \mathsf{v}_\mathsf{i}$;*
- *$P, h \longrightarrow P', h'$;*
- *P throws a null pointer exception, i.e., $P \equiv \mathsf{NullExc} \mid Q$; or*
- *P stops with a connect waiting for its dual instruction, i.e., $P \equiv E[\mathsf{connect}\ \mathsf{c}\ \mathsf{s}\,\{\mathsf{e}\}] \mid Q$.*

Proof Suppose $P \equiv \mathsf{NullExc} \mid Q$ or $P \equiv E[\mathsf{connect}\ \mathsf{c}\ \mathsf{s}\,\{\mathsf{e}\}] \mid Q$. Then the proof is immediate. Also $P \equiv \mathsf{e} \mid Q$ with $\mathsf{e}, h \longrightarrow \mathsf{e}', h'$ is easy, since we get $P, h \longrightarrow \mathsf{e}' \mid Q, h'$.

The only interesting case is $P \equiv V \mid Q$, where V is a parallel of values and Q is a parallel of evaluation contexts containing irreducible session expressions. Let $Q \equiv \prod_{1 \leq \mathsf{j} \leq \mathsf{n}} E_\mathsf{j}\,[\mathsf{e}_\mathsf{j}]$. Let c_i be the live channel with the higher index which occurs in P. By Lemma 7.8 c_i occurs exactly in two threads in P. By definition of $\equiv$, without loss of generality, we can assume that c_i occurs in $E_1\,[\mathsf{e}_1]$ and $E_2\,[\mathsf{e}_2]$. Then by Lemma B.10 c_i is the subject of e_1 and e_2. By Subject Reduction we have $\Gamma; \Sigma \vdash P; h$. This implies $\Sigma = \Sigma_1 \| \ldots \| \Sigma_\mathsf{n}$ and $\Gamma; \Sigma_\mathsf{j} \vdash E_\mathsf{j}\,[\mathsf{e}_\mathsf{j}] : \mathsf{thread}$ by Lemma A.3(2). By Lemma B.2(2) $\Sigma(\mathsf{c}_i) = \updownarrow$ and by definition of $\|$ the channel c_i occurs exactly in two session environments between $\Sigma_1, \ldots, \Sigma_\mathsf{n}$ with dual running session types different from ε. By Lemma B.2(1) c_i occurs in Σ_1 and in Σ_2 and then by above $\Sigma_1(\mathsf{c}_i) = \overline{\Sigma_2(\mathsf{c}_i)}$. Lemma B.6 gives $\mathsf{e}_1 \propto \pi$ and $\mathsf{e}_2 \propto \overline{\pi}$ for some π. Therefore e_1 and e_2 are session expressions dual of each other by Proposition B.5 and they can communicate.

Theorem 7.13 (Communication-Order Preservation) *Let P_0 be Γ-initial. Assume that $P_0, h_\Gamma \longrightarrow\!\!\!\!\rightarrow E[\mathsf{e}_0] \,|\, Q, h \longrightarrow P', h'$ where e_0 is an irreducible session expression with subject c. Then:*

(1) $P' \equiv E[\mathsf{e}_0] \,|\, Q'$, *or*
(2) $Q \equiv E'[\mathsf{e}'_0] \,|\, R$ *with* e'_0 *dual of* e_0 *and*
 (a) $E[\mathsf{e}_0] \,|\, E'[\mathsf{e}'_0] \,|\, R, h \longrightarrow \mathsf{e} \,|\, \mathsf{e}' \,|\, R', h'$*;*
 (b) $\Gamma;\Sigma, \mathsf{c} : \theta \vdash E[\mathsf{e}_0] : \mathsf{thread}$ *and* $\Gamma;\Sigma', \mathsf{c} : \overline{\theta} \vdash E'[\mathsf{e}'_0] : \mathsf{thread}$*; and*
 (c) $\Gamma;\hat{\Sigma}, \mathsf{c} : \theta' \vdash \mathsf{e} : \mathsf{thread}$ *and* $\Gamma;\hat{\Sigma}', \mathsf{c} : \overline{\theta}' \vdash \mathsf{e}' : \mathsf{thread}$ *with* $\theta' \sqsubseteq \theta$.

Proof By the proof of the Progress Theorem (Theorem 7.10) if the reduction step

$$E[\mathsf{e}_0] \,|\, Q, h \longrightarrow P', h'$$

does not reduce Q alone, then $Q \equiv E'[\mathsf{e}'_0] \,|\, R$ with e'_0 dual of e_0. Thus we have:

$$E[\mathsf{e}_0] \,|\, E'[\mathsf{e}'_0] \,|\, R, h \longrightarrow \mathsf{e} \,|\, \mathsf{e}' \,|\, R', h'$$

which shows (a).

For (b) by the Subject Reduction Theorem (Theorem 7.6) $\Gamma;\check{\Sigma} \vdash E[\mathsf{e}_0] \,|\, E'[\mathsf{e}'_0] \,|\, R : \mathsf{thread}$, which implies by Lemma A.3(2) $\check{\Sigma} = \Sigma_1 \| \Sigma_2 \| \Sigma_3$ and $\Gamma;\Sigma_1 \vdash E[\mathsf{e}_0] : \mathsf{thread}$ and $\Gamma;\Sigma_2 \vdash E'[\mathsf{e}'_0] : \mathsf{thread}$ and $\Gamma;\Sigma_3 \vdash R : \mathsf{thread}$. Again by the proof of the Progress Theorem the channel c which is the subject of e_0 and e'_0 has dual running session types in Σ_1 and Σ_2. We have then $\Sigma_1 = \Sigma, \mathsf{c} : \theta$ and $\Sigma_2 = \Sigma', \mathsf{c} : \overline{\theta}$ for some Σ, Σ', θ.

For (c) we consider only two interesting cases, the proofs in all other cases being similar. We assume $\theta = \pi.\rho$, the proof for θ of different shapes being almost the same.
Let $\mathsf{e}_0 \equiv \mathsf{c.receive}$ and $\mathsf{e}'_0 \equiv \mathsf{c.send}(\mathsf{v})$ and $\pi = ?\mathsf{t}$. Then we have $\mathsf{e} \equiv E[\mathsf{v}]$ and $\mathsf{e}' \equiv E'[\mathsf{null}]$ and $R' \equiv R$ by the reduction rule **ComS**. From the proof the Subject Reduction Theorem we get $\Gamma;\Sigma, \mathsf{c} : \rho \vdash \mathsf{e} : \mathsf{thread}$ and $\Gamma;\Sigma', \mathsf{c} : \overline{\rho} \vdash \mathsf{e}' : \mathsf{thread}$.
Let $\mathsf{e}_0 \equiv \mathsf{c.receiveS}(\mathsf{x})\{\mathsf{e}\}$ and $\mathsf{e}'_0 \equiv \mathsf{c.sendS}(\mathsf{c}')$ and $\pi = ?(\eta)$. Then we have $\mathsf{e} \equiv E[\mathsf{null}]$ and $\mathsf{e}' \equiv E'[\mathsf{null}]$ and $R' \equiv \mathsf{e}[\mathsf{c}'/\mathsf{x}] \,|\, R$ by the reduction rule **ComSS**. From the proof the Subject Reduction Theorem we get $\Gamma;\Sigma'', \mathsf{c} : \rho \vdash \mathsf{e} : \mathsf{thread}$ for some $\Sigma'' \subseteq \Sigma$ and $\Gamma;\Sigma', \mathsf{c} : \overline{\rho} \vdash \mathsf{e}' : \mathsf{thread}$.

C Proof of Theorem 8.2

We list the omitted inference rules in Fig. C.1 and Fig. C.2.

An environment session scheme is *ε-free* if all its predicates are running session type schemes different from ε. The following lemma states that:

ChanI
$$\frac{\Gamma, \mathsf{c}:\mathsf{sch} \vdash \mathsf{ok}}{\Gamma, \mathsf{c}:\mathsf{sch} \vdash \mathsf{c}:\phi \;[]\; \emptyset;\emptyset}$$

NullI
$$\frac{\Gamma \vdash \mathsf{ok} \quad \vdash \mathsf{t}:\mathtt{tp}}{\Gamma \vdash \mathsf{null}:\mathsf{t} \;[]\; \emptyset;\emptyset}$$

OidI
$$\frac{\Gamma, \mathsf{o}:C \vdash \mathsf{ok} \quad C <: D}{\Gamma, \mathsf{o}:C \vdash \mathsf{o}:D \;[]\; \emptyset;\emptyset}$$

TrueI
$$\frac{\Gamma \vdash \mathsf{ok}}{\Gamma \vdash \mathsf{true}:\mathsf{bool} \;[]\; \emptyset;\emptyset}$$

FalseI
$$\frac{\Gamma \vdash \mathsf{ok}}{\Gamma \vdash \mathsf{false}:\mathsf{bool} \;[]\; \emptyset;\emptyset}$$

VarI
$$\frac{\Gamma, \mathsf{x}:\mathsf{t} \vdash \mathsf{ok}}{\Gamma, \mathsf{x}:\mathsf{t} \vdash \mathsf{x}:\mathsf{t} \;[]\; \emptyset;\emptyset}$$

ThisI
$$\frac{\Gamma, \mathsf{this}:C \vdash \mathsf{ok} \quad C <: D}{\Gamma, \mathsf{this}:C \vdash \mathsf{this}:D \;[]\; \emptyset;\emptyset}$$

FldI
$$\frac{\Gamma \vdash \mathsf{e}:C \;[]\; \Sigma;\mathscr{S} \quad \mathsf{f}\,\mathsf{t} \in \mathsf{fields}(C)}{\Gamma \vdash \mathsf{e.f}:\mathsf{t} \;[]\; \Sigma;\mathscr{S}}$$

SeqI
$$\frac{\Gamma \vdash \mathsf{e}:\mathsf{t} \;[]\; \Sigma;\mathscr{S} \qquad \Gamma \vdash \mathsf{e}':\mathsf{t}' \;[]\; \Sigma';\mathscr{S}'}{\Gamma \vdash \mathsf{e};\mathsf{e}':\mathsf{t}' \;[]\; \Sigma\circ\Sigma';\mathscr{S} \uplus \mathscr{S}'}$$

FldAssI
$$\frac{\Gamma \vdash \mathsf{e}:C \;[]\; \Sigma;\mathscr{S} \qquad \Gamma \vdash \mathsf{e}':\mathsf{t} \;[]\; \Sigma';\mathscr{S}' \quad \mathsf{f}\,\mathsf{t} \in \mathsf{fields}(C)}{\Gamma \vdash \mathsf{e.f} := \mathsf{e}':\mathsf{t} \;[]\; \Sigma\circ\Sigma';\mathscr{S} \uplus \mathscr{S}'}$$

NewCI
$$\frac{\Gamma \vdash \mathsf{ok} \quad C \in \mathscr{D}(\mathtt{CT}) \quad C <: D}{\Gamma \vdash \mathsf{new}\ C:D \;[]\; \emptyset;\emptyset}$$

NewSI
$$\frac{\Gamma \vdash \mathsf{ok}}{\Gamma \vdash \mathsf{new}\ (\mathsf{s},\bar{\mathsf{s}}):(\mathsf{s},\bar{\mathsf{s}}) \;[]\; \emptyset;\emptyset}$$

SpawnI
$$\frac{\Gamma \vdash \mathsf{e}:\mathsf{t} \;[]\; \Sigma;\mathscr{S}}{\Gamma \vdash \mathsf{spawn}\{\ \mathsf{e}\ \}:\mathit{Object} \;[]\; \Sigma\!\downarrow;\mathscr{S}}$$

NullPEI
$$\frac{\Gamma \vdash \mathsf{ok} \quad \vdash \mathsf{t}:\mathtt{tp}}{\Gamma \vdash \mathsf{NullExc}:\mathsf{t} \;[]\; \emptyset;\emptyset}$$

Fig. C.1. Inference Rules for Values and Standard Expressions II

- session environment schemes obtained by applying inference substitutions to inferred session environment schemes for expressions are ε-free and they never contain $\updownarrow$ as predicate, and
- session environment schemes obtained by applying inference substitutions to inferred session environment schemes for threads are ε-free.

The proof by induction on deductions is standard.

Lemma C.1 *(1) If* $\Gamma \vdash \mathsf{e}:\mathsf{t} \;[]\; \Sigma;\mathscr{S}$ *and* σ *is an inference substitution, then* $\sigma(\Sigma)$ *is* ε*-free and* $\mathsf{u}:\rho \in \sigma(\Sigma)$ *implies* $\rho \neq \updownarrow$*;*
(2) If $\Gamma \vdash P:\mathtt{thread} \;[]\; \Sigma$ *and* σ *is an inference substitution, then* $\sigma(\Sigma)$ *is* ε*-free.*

ReceiveIfI

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{e}_i : \mathsf{t}_i \mathrel{[]} \Sigma_i ; \mathscr{S}_i \quad \Sigma_i(\!(\mathsf{u})\!) = \rho_i \quad \mathscr{S}_i \subseteq \{\mathsf{u}\} \quad i \in \{1,2\} \\ \sigma = \mathsf{E}(\{\langle \mathsf{t}_1 ; \mathsf{t}_2 \rangle\} \cup \{\langle \Sigma_1(\mathsf{u}') ; \Sigma_2(\mathsf{u}')\rangle \mid \forall \mathsf{u}' \neq \mathsf{u} . \mathsf{u}' \in \mathscr{D}(\Sigma_1) \cap \mathscr{D}(\Sigma_2)\} \cup \\ \{\langle \Sigma_i(\mathsf{u}') ; \varepsilon.\mathsf{end}\rangle \mid \forall \mathsf{u}' \neq \mathsf{u} . \mathsf{u}' \in \mathscr{D}(\Sigma_i) \ \&\ \mathsf{u}' \notin \mathscr{D}(\Sigma_j)\ i,j \in \{1,2\}\}) \\ \rho_i' = \begin{cases} \rho_i \downarrow & \text{if } \rho_j \text{ is an ended session type scheme,} \\ \rho_i & \text{otherwise} \end{cases} \quad i \neq j,\ i,j \in \{1,2\} \\ \Sigma = \{\mathsf{u}' : \varepsilon.\mathsf{end} \mid \forall \mathsf{u}' \neq \mathsf{u} . \mathsf{u}' \in \mathscr{D}(\Sigma_2) \ \&\ \mathsf{u}' \notin \mathscr{D}(\Sigma_1)\}\end{array}}{\Gamma \vdash \mathsf{u}.\mathsf{receiveIf}\{\mathsf{e}_1\}\{\mathsf{e}_2\} : \sigma(\mathsf{t}) \mathrel{[]} \sigma(\Sigma_1 \setminus \mathsf{u}, \mathsf{u} : ?\langle \rho_1', \rho_2' \rangle) \cup \Sigma ; \{\mathsf{u}\}}$$

SendWhileI

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{e} : \mathsf{t}_0 \mathrel{[]} \emptyset ; \emptyset \quad \Gamma \vdash \mathsf{e}' : \mathsf{t} \mathrel{[]} \Sigma ; \mathscr{S} \\ \Sigma \subseteq \{\mathsf{u} : \pi\} \quad \mathscr{S} \subseteq \{\mathsf{u}\} \quad \sigma = \mathsf{E}(\{\langle \mathsf{t}_0 ; \mathsf{bool}\rangle\})\end{array}}{\Gamma \vdash \mathsf{u}.\mathsf{sendWhile}(\mathsf{e})\{\mathsf{e}'\} : \sigma(\mathsf{t}) \mathrel{[]} \sigma(\{\mathsf{u} : !\langle \pi \rangle^*\}) ; \{\mathsf{u}\}}$$

ReceiveWhileI

$$\frac{\Gamma \vdash \mathsf{e} : \mathsf{t} \mathrel{[]} \Sigma ; \mathscr{S} \quad \Sigma \subseteq \{\mathsf{u} : \pi\} \quad \mathscr{S} \subseteq \{\mathsf{u}\}}{\Gamma \vdash \mathsf{u}.\mathsf{receiveWhile}\{\mathsf{e}\} : \mathsf{t} \mathrel{[]} \{\mathsf{u} : ?\langle \pi \rangle^*\} ; \{\mathsf{u}\}}$$

Fig. C.2. Inference Rules for Communication Expressions II

A second lemma gives useful properties of inference substitutions and $\Subset$.

Lemma C.2 *(1) If Σ is a session environment scheme and $\sigma(\Sigma)$ is a session environment, then $\sigma(\Sigma \setminus \mathsf{u}) = \sigma(\Sigma) \setminus \mathsf{u}$.*

(2) If Σ is a session environment scheme and $\sigma(\Sigma)$ is a session environment, then $\sigma(\Sigma\downarrow) = \sigma(\Sigma)\downarrow$.

(3) If Σ, Σ' are session environments and $\Sigma' \Subset \Sigma, \mathsf{u} : \eta$, then $\Sigma' \setminus \mathsf{u} \Subset \Sigma$.

(4) If Σ, Σ' are session environment schemes and $\sigma(\Sigma), \sigma(\Sigma')$ are session environments and $\sigma(\Sigma) \| \sigma(\Sigma')$ is defined, then $\sigma(\Sigma \mathbin{|||} \Sigma') = \sigma(\Sigma) \| \sigma(\Sigma')$.

(5) If $\Sigma_1, \Sigma_2, \Sigma_1', \Sigma_2'$ are session environments and $\Sigma_1' \| \Sigma_2'$ is defined and $\Sigma_1' \cup \Sigma_2'$ is ε-free, then $\Sigma_1 \Subset \Sigma_1'$ and $\Sigma_2 \Subset \Sigma_2'$ imply $\Sigma_1 \| \Sigma_2 \Subset \Sigma_1' \| \Sigma_2'$.

Proof (1) and (2) are immediate.

(3) follows from the definitions of $\Subset$.

(4) Easy by definition of $|||$ on session environment schemes.

(5) By induction on Σ_1. The basic case, $\Sigma_1 = \emptyset$, is trivial. For the induction case, $\Sigma_1 = \Sigma_0, \mathsf{u} : \updownarrow$ is trivial too. For $\Sigma_1 = \Sigma_0, \mathsf{u} : \rho$, we need to consider different subcases. We always have $\Sigma_1'(\mathsf{u}) \in \{\rho, \rho\downarrow\}$ by definition of $\Subset$.

If $\mathsf{u} \notin \mathscr{D}(\Sigma_2')$, then $\mathsf{u} \notin \mathscr{D}(\Sigma_2)$ and $\Sigma_1 \| \Sigma_2(\mathsf{u}) = \Sigma_1'(\mathsf{u})$.

If $\mathsf{u} : \rho' \in \Sigma_2$, then $\Sigma'_2(\mathsf{u}) \in \{\rho', \rho' \downarrow\}$. Being $\Sigma'_1 \| \Sigma'_2$ defined, we have $\rho = \overline{\rho'}$, or $\rho \downarrow = \overline{\rho'}$, or $\rho = \overline{\rho' \downarrow}$. In all cases we get $\Sigma_1 \| \Sigma_2(\mathsf{u}) = \Sigma'_1 \| \Sigma'_2(\mathsf{u}) = \updownarrow$.

If $\mathsf{u} \notin \mathscr{D}(\Sigma_2)$ and $\mathsf{u} : \rho' \in \Sigma'_2$, then as in previous case $\rho = \overline{\rho'}$, or $\rho \downarrow = \overline{\rho'}$. Being $\Sigma'_1 \| \Sigma'_2$ defined, we have $\rho' \neq \updownarrow$ and we conclude $\rho' = \varepsilon.\mathsf{end}$, since $\Sigma'_1 \cup \Sigma'_2$ is ε-free. So $\Sigma_1 \| \Sigma_2(\mathsf{u}) = \varepsilon.\mathsf{end}$, which implies $\Sigma'_1 \| \Sigma'_2(\mathsf{u}) = \updownarrow$.

Theorem 8.2 *(1) If $\Gamma; \Sigma; \mathscr{S} \vdash \mathsf{e} : \mathsf{t}$ without using rule* **WeakB**, *then $\Gamma \vdash \mathsf{e} : \mathsf{t}' \;[\,]\; \Sigma'; \mathscr{S}'$ where $\sigma(\mathsf{t}') = \mathsf{t}$ and $\sigma(\Sigma') \Subset \Sigma$ for some inference substitution σ and $\mathscr{S}' \subseteq \mathscr{S}$.*

(2) If $\Gamma \vdash \mathsf{e} : \mathsf{t} \;[\,]\; \Sigma; \mathscr{S}$, then for all inference substitutions σ such that $\sigma(\Sigma)$ is a session environment and $\sigma(\mathsf{t})$ is a type, we get: $\Gamma; \sigma(\Sigma); \mathscr{S} \vdash \mathsf{e} : \sigma(\mathsf{t})$.

(3) If $\Gamma; \Sigma \vdash P : \mathsf{thread}$ without using rule **WeakB**, *then $\Gamma \vdash P : \mathsf{thread} \;[\,]\; \Sigma'$ where $\sigma(\Sigma') \Subset \Sigma$ for some inference substitution σ.*

(4) If $\Gamma \vdash P : \mathsf{thread} \;[\,]\; \Sigma$, then for all inference substitutions σ such that $\sigma(\Sigma)$ is a session environment, we get: $\Gamma; \sigma(\Sigma) \vdash P : \mathsf{thread}$.

Proof The proofs of all points are by induction on derivations and we only consider the more interesting cases.

(1) If the last applied rule is

Conn

$$\frac{\Gamma; \emptyset; \emptyset \vdash \mathsf{u} : \mathsf{begin}.\eta \qquad \Gamma \setminus \mathsf{u}\, ; \Sigma, \mathsf{u} : \eta; \{\mathsf{u}\} \vdash \mathsf{e} : \mathsf{t}}{\Gamma; \Sigma; \emptyset \vdash \mathsf{connect}\ \mathsf{u}\ \mathsf{begin}.\eta\ \{\mathsf{e}\} : \mathsf{t}}$$

by induction hypothesis we have

$$\Gamma \setminus \mathsf{u} \vdash \mathsf{e} : \mathsf{t}' \;[\,]\; \Sigma'; \mathscr{S}' \tag{C.1}$$

with $\sigma(\mathsf{t}') = \mathsf{t}$ and $\sigma(\Sigma') \Subset \Sigma, \mathsf{u} : \eta$ for some inference substitution σ and $\mathscr{S}' \subseteq \mathscr{S}$. Let $\Sigma'(\!(\mathsf{u})\!) = \rho$, we get $\sigma(\rho \downarrow) = \eta$. If σ_ρ is the restriction of σ to the type variables which occur in ρ we have also $\sigma_\rho(\rho \downarrow) = \eta$. If u is a variable, then $\Gamma(\mathsf{u}) <: \mathsf{s}$ by Lemma A.1(1). If u is a name, then $\Gamma(\mathsf{u}) = \mathsf{sch}$ by Lemma A.1(2). By applying rule **ConnI** to (C.1) we derive

$$\Gamma \vdash \mathsf{connect}\ \mathsf{u}\ \mathsf{begin}.\eta\ \{\mathsf{e}\} : \sigma_\rho(\mathsf{t}') \;[\,]\; \sigma_\rho(\Sigma') \setminus \mathsf{u}\, ; \emptyset$$

If σ_- is the restriction of σ to the type variables which do not occur in ρ (and so $\sigma_- \circ \sigma_\rho = \sigma$) we get $\sigma_-(\sigma_\rho(\mathsf{t}')) = \mathsf{t}$ and $\sigma_-(\sigma_\rho(\Sigma' \setminus \mathsf{u})) = \sigma_-(\sigma_\rho(\Sigma')) \setminus \mathsf{u} \Subset \Sigma$ by Lemma C.2(1) and (3), and this concludes the proof of this case.

If the last applied rule is

ReceiveS

$$\frac{\Gamma \setminus \mathsf{x}\, ; \{\mathsf{x} : \eta\}; \{\mathsf{x}\} \vdash \mathsf{e} : \mathsf{t} \qquad \eta \neq \varepsilon.\mathsf{end}}{\Gamma; \{\mathsf{u} : ?(\eta)\}; \{\mathsf{u}\} \vdash \mathsf{u}.\mathsf{receiveS}\,(\mathsf{x})\{\mathsf{e}\} : \mathit{Object}}$$

by induction hypothesis we have

$$\Gamma \setminus \mathsf{x} \vdash \mathsf{e} : \mathsf{t}' \; [] \; \Sigma'; \mathscr{S}' \tag{C.2}$$

with $\sigma(\mathsf{t}') = \mathsf{t}$ and $\sigma(\Sigma') \Subset \{\mathsf{x} : \eta\}$ for some inference substitution σ and $\mathscr{S}' \subseteq \{\mathsf{x}\}$. From $\sigma(\Sigma') \Subset \{\mathsf{x} : \eta\}$ we get $\Sigma' = \{\mathsf{x} : \rho\}$, and $\sigma(\rho\downarrow) = \eta$ for some $\rho \neq \varepsilon$. We can conclude by applying rule **ReceiveSI** to (C.2).

If the last applied rule is

$$\textbf{SendIf}\quad \frac{\Gamma; \Sigma_0; \{\mathsf{u}\} \vdash \mathsf{e} : \mathsf{bool} \qquad \Gamma; \Sigma, \mathsf{u} : \rho_i; \{\mathsf{u}\} \vdash \mathsf{e}_i : \mathsf{t} \quad i \in \{1,2\}}{\Gamma; \Sigma_0 \circ \{\Sigma, \mathsf{u} :!\langle \rho_1, \rho_2 \rangle\}; \{\mathsf{u}\} \vdash \mathsf{u}.\mathsf{sendIf}(\mathsf{e})\{\mathsf{e}_1\}\{\mathsf{e}_2\} : \mathsf{t}}$$

by induction hypothesis we have

$$\Gamma \vdash \mathsf{e} : \mathsf{t}_0 \; [] \; \Sigma_0'; \mathscr{S}_0 \quad \Gamma \vdash \mathsf{e}_i : \mathsf{t}_i \; [] \; \Sigma_i; \mathscr{S}_i \quad i \in \{1,2\} \tag{C.3}$$

with $\sigma_0(\mathsf{t}_0) = \mathsf{bool}$, $\sigma_0(\Sigma_0') \Subset \Sigma_0$, $\sigma_i(\mathsf{t}_i) = \mathsf{t}$, $\sigma_i(\Sigma_i) \Subset \Sigma, \mathsf{u} : \rho_i$ for some inference substitution σ_0, σ_i, and $\mathscr{S}_0 \subseteq \{\mathsf{u}\}$, $\mathscr{S}_i \subseteq \{\mathsf{u}\}$, $i \in \{1,2\}$. From $\sigma_i(\Sigma_i) \Subset \Sigma, \mathsf{u} : \rho_i$ we get $\sigma_i(\Sigma_i)(\!(\mathsf{u})\!) \Subset \rho_i$. By Lemma C.2(2) $\sigma_i(\Sigma_i) \Subset \Sigma, \mathsf{u} : \rho_i$ implies $\sigma_i(\Sigma_i) \setminus \mathsf{u} \Subset \Sigma$, and then either $\sigma_1(\Sigma_1)(\mathsf{u}') = \sigma_2(\Sigma_2)(\mathsf{u}')$, $\sigma_1(\Sigma_1)(\mathsf{u}')\downarrow = \sigma_2(\Sigma_2)(\mathsf{u}')$, or $\sigma_1(\Sigma_1)(\mathsf{u}') = \sigma_2(\Sigma_2)(\mathsf{u}')\downarrow$ for all $\mathsf{u}' \neq \mathsf{u}.\mathsf{u}' \in \mathscr{D}(\Sigma_1) \cap \mathscr{D}(\Sigma_2)$. Moreover by Lemma C.1(1) $\mathsf{u}' \neq \mathsf{u}$, and $\mathsf{u}' : \rho \in \sigma_1(\Sigma_1)$, and $\mathsf{u}' \notin \mathscr{D}(\Sigma_2)$ imply $\rho = \varepsilon.\mathsf{end}$. Symmetrically $\mathsf{u}' \neq \mathsf{u}$, and $\mathsf{u}' : \rho \in \sigma_2(\Sigma_2)$, and $\mathsf{u}' \notin \mathscr{D}(\Sigma_1)$ imply $\rho = \varepsilon.\mathsf{end}$. Since by assumption the variables in $\Sigma_0, \Sigma_1, \Sigma_2$ are disjoint, then $\sigma_0 \circ \sigma_1 \circ \sigma_2$ is defined. Let $\sigma = \mathsf{E}(\{\langle \Sigma_1(\mathsf{u}'); \Sigma_2(\mathsf{u}') \rangle \mid \forall \mathsf{u}' \neq \mathsf{u}.\mathsf{u}' \in \mathscr{D}(\Sigma_1) \cap \mathscr{D}(\Sigma_2)\} \cup \{\langle \Sigma_1(\mathsf{u}'); \varepsilon.\mathsf{end} \rangle \mid \forall \mathsf{u}' \neq \mathsf{u}.\mathsf{u}' \in \mathscr{D}(\Sigma_1) \;\&\; \mathsf{u}' \notin \mathscr{D}(\Sigma_2)\} \cup \{\langle \Sigma_2(\mathsf{u}'); \varepsilon.\mathsf{end} \rangle \mid \forall \mathsf{u}' \neq \mathsf{u}.\mathsf{u}' \in \mathscr{D}(\Sigma_2) \;\&\; \mathsf{u}' \notin \mathscr{D}(\Sigma_1)\})$: by construction there is σ' such that $\sigma \circ \sigma' = \sigma_0 \circ \sigma_1 \circ \sigma_2$. We conclude by applying rule **SendIfI** to (C.3).

(2) If the last applied rule is

ConnI

$$\frac{\Gamma \setminus \mathsf{u} \vdash \mathsf{e} : \mathsf{t} \; [] \; \Sigma; \mathscr{S} \quad \Sigma(\!(\mathsf{u})\!) = \rho \quad \mathsf{s} = \mathsf{begin}.\sigma(\rho\downarrow) \quad \mathscr{S} \subseteq \{\mathsf{u}\} \quad \begin{array}{l}\text{if } \mathsf{u} \text{ is a variable } \Gamma(\mathsf{u}) <: \mathsf{s} \\ \text{if } \mathsf{u} \text{ is a name } \Gamma(\mathsf{u}) = \mathsf{sch}\end{array}}{\Gamma \vdash \mathsf{connect}\ \mathsf{u}\ \mathsf{s}\ \{\mathsf{e}\} : \sigma(\mathsf{t}) \; [] \; \sigma(\Sigma) \setminus \mathsf{u}; \emptyset}$$

by induction hypothesis for all σ' we have

$$\Gamma \setminus \mathsf{u}; \sigma'(\Sigma); \mathscr{S} \vdash \mathsf{e} : \sigma'(\mathsf{t})$$

and this holds in particular for those inference substitutions σ' such that $\sigma' = \sigma'' \circ \sigma$ for some σ''. If $S = \emptyset$ by rule **Weak** we get $\Gamma \setminus \mathsf{u}; \sigma'(\Sigma); \{\mathsf{u}\} \vdash \mathsf{e} : \sigma'(\mathsf{t})$. Let $\Sigma' = \sigma'(\Sigma)$ if $\mathsf{u} \in \mathscr{D}(\Sigma)$ and $\Sigma' = \sigma'(\Sigma), \mathsf{u} : \varepsilon.\mathsf{end}$ otherwise. In both cases (using rules

WeakES and **WeakE** in the second case) we get

$$\Gamma \setminus \mathsf{u}; \Sigma'; \{\mathsf{u}\} \vdash \mathsf{e} : \sigma'(\mathsf{t}) \tag{C.4}$$

If u is a channel name, then $\Gamma(\mathsf{u}) = \mathsf{sch}$ and $\Gamma;\emptyset;\emptyset \vdash \mathsf{u} : \mathsf{s}$ by rules **Chan** and **Sub**. If u is a variable name, then $\Gamma(\mathsf{u}) <: \mathsf{s}$ implies $\Gamma;\emptyset;\emptyset \vdash \mathsf{u} : \mathsf{s}$ by rules **Var** and **Sub**. We can conclude applying rule **Conn** to (C.4).

If the last applied rule is

ReceiveSI

$$\frac{\Gamma\setminus\mathsf{x} \vdash \mathsf{e} : \mathsf{t}\ [\,]\ \{\mathsf{x} : \rho\}; \mathscr{S} \quad \mathscr{S} \subseteq \{\mathsf{x}\} \quad \rho \neq \varepsilon}{\Gamma \vdash \mathsf{u}.\mathsf{receiveS}(\mathsf{x})\{\mathsf{e}\} : \mathit{Object}\ [\,]\ \{\mathsf{u} : ?(\rho\downarrow)\}; \{\mathsf{u}\}}$$

by induction hypothesis for all σ we have

$$\Gamma \setminus \mathsf{x}; \{\mathsf{x} : \sigma(\rho)\}; \mathscr{S} \vdash \mathsf{e} : \sigma(\mathsf{t})$$

If $S = \emptyset$ by rule **Weak** we get $\Gamma \setminus \mathsf{x}; \{\mathsf{x} : \sigma(\rho)\}; \{\mathsf{x}\} \vdash \mathsf{e} : \sigma(\mathsf{t})$. By rule **WeakE** we can derive

$$\Gamma \setminus \mathsf{x}; \{\mathsf{x} : \sigma(\rho)\downarrow\}; \{\mathsf{x}\} \vdash \mathsf{e} : \sigma(\mathsf{t}) \tag{C.5}$$

where $\rho\downarrow \neq \varepsilon.\mathsf{end}$, and we can conclude applying rule **ReceiveS** to (C.5).

If the last applied rule is

SendIfI

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{e} : \mathsf{t}_0\ [\,]\ \Sigma_0; \mathscr{S}_0 \quad \Gamma \vdash \mathsf{e}_i : \mathsf{t}_i\ [\,]\ \Sigma_i; \mathscr{S}_i \quad \Sigma_i(\!(\mathsf{u})\!) = \rho_i \quad \mathscr{S}_j \subseteq \{\mathsf{u}\} \quad i \in \{1,2\} \quad j \in \{0,1,2\} \\ \sigma = \mathsf{E}(\{\langle \mathsf{t}_1; \mathsf{t}_2\rangle, \langle \mathsf{t}_0; \mathsf{bool}\rangle\} \cup \{\langle \Sigma_1(\mathsf{u}'); \Sigma_2(\mathsf{u}')\rangle \mid \forall \mathsf{u}' \neq \mathsf{u}.\mathsf{u}' \in \mathscr{D}(\Sigma_1) \cap \mathscr{D}(\Sigma_2)\} \cup \\ \{\langle \Sigma_1(\mathsf{u}'); \varepsilon.\mathsf{end}\rangle \mid \forall \mathsf{u}' \neq \mathsf{u}.\mathsf{u}' \in \mathscr{D}(\Sigma_1)\ \&\ \mathsf{u}' \notin \mathscr{D}(\Sigma_2)\} \cup \\ \{\langle \Sigma_2(\mathsf{u}'); \varepsilon.\mathsf{end}\rangle \mid \forall \mathsf{u}' \neq \mathsf{u}.\mathsf{u}' \in \mathscr{D}(\Sigma_2)\ \&\ \mathsf{u}' \notin \mathscr{D}(\Sigma_1)\}) \\ \rho_1' = \begin{cases} \rho_1\downarrow & \text{if } \rho_2 \text{ is an ended session type scheme,} \\ \rho_1 & \text{otherwise} \end{cases} \\ \rho_2' = \begin{cases} \rho_2\downarrow & \text{if } \rho_1 \text{ is an ended session type scheme,} \\ \rho_2 & \text{otherwise} \end{cases} \\ \Sigma = \{\mathsf{u}' : \varepsilon.\mathsf{end} \mid \forall \mathsf{u}' \neq \mathsf{u}.\mathsf{u}' \in \mathscr{D}(\Sigma_2)\ \&\ \mathsf{u}' \notin \mathscr{D}(\Sigma_1)\}\end{array}}{\Gamma \vdash \mathsf{u}.\mathsf{sendIf}(\mathsf{e})\{\mathsf{e}_1\}\{\mathsf{e}_2\} : \sigma(\mathsf{t})\ [\,]\ \sigma(\Sigma_0)\circ(\sigma(\Sigma_1 \setminus \mathsf{u}, \mathsf{u} :!\langle \rho_1', \rho_2'\rangle) \cup \Sigma); \{\mathsf{u}\}}$$

by induction hypothesis for all σ' we have

$$\Gamma; \sigma'(\Sigma_0); \mathscr{S}_0 \vdash \mathsf{e} : \sigma'(\mathsf{t}_0) \quad \Gamma; \sigma'(\Sigma_i); \mathscr{S}_i \vdash \mathsf{e}_i : \sigma'(\mathsf{t}_i)$$

and this holds in particular for those inference substitutions σ' such that $\sigma' = \sigma'' \circ \sigma$ for some σ''. Using rules **Weak**, **WeakES** and **WeakE** we can derive

$$\Gamma; \sigma'(\Sigma_0); \{\mathsf{u}\} \vdash \mathsf{e} : \sigma'(\mathsf{t}_0) \quad \Gamma; \sigma'(\Sigma_1 \setminus \mathsf{u}, \mathsf{u} : \rho_i') \cup \Sigma; \{\mathsf{u}\} \vdash \mathsf{e}_i : \sigma'(\mathsf{t}_i). \tag{C.6}$$

We can conclude applying rule **SendIf** to (C.6).

(3) If the last applied rule is

Par

$$\frac{\Gamma;\Sigma_1 \vdash P_1 : \mathsf{thread} \quad \Gamma;\Sigma_2 \vdash P_2 : \mathsf{thread}}{\Gamma;\Sigma_1 \| \Sigma_2 \vdash P_1 \,|\, P_2 : \mathsf{thread}}$$

by induction hypothesis we have

$$\Gamma \vdash P_1 : \mathsf{thread} \ \sqcup \ \Sigma_1' \quad \Gamma \vdash P_2 : \mathsf{thread} \ \sqcup \ \Sigma_2'$$

and there are σ_1, σ_2 such that $\sigma_i(\Sigma_i') \Subset \Sigma_i$ for $i \in \{1,2\}$. By rule **ParI** we get

$$\Gamma \vdash P_1 \,|\, P_2 : \mathsf{thread} \ \sqcup \ \Sigma_1' \mathbin{|||} \Sigma_2'.$$

By Lemma C.1(2) ε is not a predicate in $\sigma_1(\Sigma_1') \cup \sigma_2(\Sigma_2')$. By Lemma C.2(5) $\sigma_i(\Sigma_i') \Subset \Sigma_i$ for $i \in \{1,2\}$ imply $\sigma_1(\Sigma_1') \| \sigma_2(\Sigma_2') \Subset \Sigma_1 \| \Sigma_2$. By construction the sets of variables occurring in Σ_1' and Σ_2' are disjoint. Let $\sigma = \sigma_1 \circ \sigma_2$, then $\sigma_1(\Sigma_1') \| \sigma_2(\Sigma_2') = \sigma(\Sigma_1') \| \sigma(\Sigma_2') = \sigma(\Sigma_1' \mathbin{|||} \Sigma_2')$ by Lemma C.2(4).

(4) If the last applied rule is

ParI

$$\frac{\Gamma \vdash P_1 : \mathsf{thread} \ \sqcup \ \Sigma_1 \quad \Gamma \vdash P_2 : \mathsf{thread} \ \sqcup \ \Sigma_2}{\Gamma \vdash P_1 \,|\, P_2 : \mathsf{thread} \ \sqcup \ \Sigma_1 \mathbin{|||} \Sigma_2}$$

by induction hypothesis for all σ we have:

$$\Gamma;\sigma(\Sigma_1) \vdash P_1 : \mathsf{thread} \quad \Gamma;\sigma(\Sigma_2) \vdash P_2 : \mathsf{thread}$$

Being $\sigma(\Sigma_1) \| \sigma(\Sigma_2) = \sigma(\Sigma_1 \mathbin{|||} \Sigma_2)$ by Lemma C.2(4), we conclude by applying rule **Par**.