Session Types for Object-Oriented Languages

Mariangiola Dezani-Ciancaglini[1], Dimitris Mostrous[2],
Nobuko Yoshida[2], and Sophia Drossopoulou[2]

[1] Dipartimento di Informatica, Università di Torino
[2] Department of Computing, Imperial College London

Abstract. A session takes place between two parties; after establishing a connection, each party interleaves local computations with communications (sending or receiving) with the other party. 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 the language 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 whether we can offer a high-level language abstraction that enables tractable description of type-safe communication patterns seamlessly integrated in 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, and 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 [16], with the aim 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 completed. Thus, session types specify the communication behaviour of a piece of software, and can be used to verify the safety, in terms of communication, of the composition of several pieces of software executing in parallel. Session types have been studied for several different settings, *i.e.*, for π-calculus-based formalisms [4, 5, 11, 16, 18, 24], for CORBA [25], for functional languages [13, 27], and recently, for a W3C standard description language for web services called Choreography Description Language (CDL) [6, 28].

In this paper we study the incorporation of session types into object-oriented languages. To our knowledge, except for our earlier work [8], 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.

In the design of MOOSE, we were guided by our wish that it should have 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 [22], as well as those from the WC3 standard documents [6, 28]. In order to achieve expressivity, we introduced the ability to spawn new threads, to send and receive sessions (*i.e.*, higher-order sessions), conditional, and iterative sessions.

type preservation We wanted to be able to guarantee that execution preserves types, *i.e.*, to obtain the subject reduction property, and this 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. In order to ensure linear usage of channels, we had to forbid their aliasing, and therefore, these cannot be stored in the fields of objects.

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, 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 can ensure it. The combination of higher-order sessions, spawn and the requirement to prevent deadlock during the session posed the major challenge for our type system.

The remaining sections are organised as follows. Section 2 illustrates the basic ideas of MOOSE through an example. Section 3 defines the syntax of the language. Section 4 presents the operational semantics. Section 5 describes design decisions. Section 6 illustrates the typing system. Section 7 is devoted to basic theorems on type safety and communication safety. Section 8 discusses the related work, and Section 9 concludes. The Appendix gives complete definitions and proofs. Also more detailed explanations and examples can be found in [22].

2 Example: Business Protocol

We describe a typical collaboration pattern that appears in many web service business protocols [6, 28] 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 and deadlock-freedom during the session.

In Fig. 1 we show the sequence diagram for the protocol which models the purchasing of items as follows: 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. If the price received is acceptable then the Seller 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 will await for the Buyer's address, before responding with the delivery date. If the price is not acceptable, then the interaction terminates.

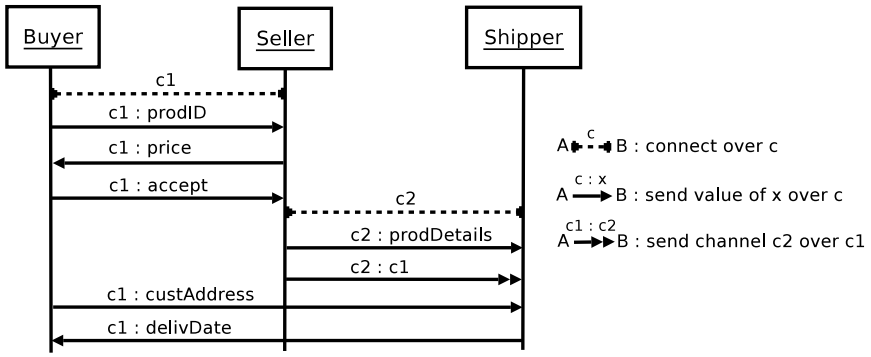


Fig. 1. Sequence diagram for item purchasing protocol.

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 $\overline{\texttt{BuyProduct}}$ for the *dual* type, which is constructed by taking `BuyProduct` and changing occurrences of `!` to `?` and vice versa; hence, 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, $\overline{\texttt{BuyProduct}}$ 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 former thread's decision. The session type `RequestDelivery` describes sending a `ProductDetails` instance, followed by sending a 'live' session channel of type `?Address.!DeliveryDetails.end`.

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

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

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 communication, 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 session bodies, therefore the objects can proceed to perform their interactions without the possibility of external interference.

In the body of method `buy` once the session has started, 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.*, `c1.receive <= maxPrice` , then `true` is sent and the first branch of the conditional, starting on line `9` is taken. 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 former 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.

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

The above example shows how MOOSE achieves deadlock-freedom during a session: because sessions take place between threads with complementary communication

```
1  class Buyer {
2
3     Address addr;
4
5     void buy( String prodID, double maxPrice ) {
6        connect c1 BuyProduct {
7           c1.send( prodID );
8           c1.sendIf( c1.receive <= maxPrice ) {
9                c1.send( addr );
10               DeliveryDetails delivDetails := c1.receive;
11          }{ null; /* buyer rejects price, end of protocol */ }
12       } /* End connect */
13    } /* End method buy */
14 }
15
16 class Seller {
17    void sell() {
18       connect c1 BuyProduct {
19          String prodID := c1.receive;
20          double price := getPrice( prodID ); // implem. omitted
21          c1.send( price );
22          c1.receiveIf { // buyer accepts price
23              ProductDetails prodDetails := new ProductDetails();
24              // ... init prodDetails with prodID, size, etc
25              spawn { connect c2 RequestDelivery {
26                      c2.send( prodDetails ); c2.sendS( c1 );} }
27              }{ null; /* receiveIf : buyer rejects */ }
28          }
29     }
30 }
31
32 class Shipper {
33     void delivery() {
34        connect c2 RequestDelivery {
35             ProductDetails prodDetails := c2.receive;
36             c2.receiveS( x ) {
37             Address custAddress := x.receive;
38             DeliveryDetails delivDetails := new DeliveryDetails();
39             //... set state of delivDetails
40             x.send( delivDetails ); }
41        }
42     }
43 }
```

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

patterns, whenever we have `c.send(v)` eventually an expression of the shape `c.receive` will appear in the other thread, unless the thread diverges or an exception occurs. Likewise for the other communication expressions. Therefore, no session will remain in-

(type) $\mathsf{t} ::= C \mid \mathsf{bool} \mid \mathsf{s} \mid (\mathsf{s}, \overline{\mathsf{s}})$

(direction) $\dagger ::= ! \mid ?$

(session) $\mathsf{s} ::= \mathsf{begin}.\rho \qquad \rho ::= \pi.\mathsf{end} \mid \pi.\dagger\langle \rho, \rho \rangle \qquad \eta ::= \pi \mid \rho$
$\pi ::= \varepsilon \mid \dagger \mathsf{t} \mid \dagger(\rho) \mid \dagger\langle \pi, \pi \rangle \mid \dagger\langle \pi \rangle^* \mid \pi.\pi$

(class) $\mathit{class} ::= \mathsf{class}\ C\ \mathsf{extends}\ C\ \{\ \tilde{\mathsf{f}}\,\tilde{\mathsf{t}}\ \ \tilde{\mathit{meth}}\ \}$

(method) $\mathit{meth} ::= \mathsf{t}\,\mathsf{m}\ (\tilde{\mathsf{t}}\ \tilde{\mathsf{x}})\ \{\mathsf{e}\} \mid \mathsf{t}\,\mathsf{m}\ (\tilde{\mathsf{t}}\ \tilde{\mathsf{x}}, \eta\,\mathsf{x})\ \{\mathsf{e}\}$

(expression) $\mathsf{e} ::= \mathsf{x} \mid \mathsf{v} \mid \mathsf{this} \mid \mathsf{e}; \mathsf{e} \mid \mathsf{e}.\mathsf{f} := \mathsf{e} \mid \mathsf{e}.\mathsf{f} \mid \mathsf{e}.\mathsf{m}(\tilde{\mathsf{e}}) \mid \mathsf{new}\ C \mid \mathsf{new}\ (\mathsf{s}, \overline{\mathsf{s}})$
$\mid$ NullExc $\mid \mathsf{spawn}\,\{\,\mathsf{e}\,\} \mid \mathsf{connect}\ \mathsf{u}\ \mathsf{s}\,\{\mathsf{e}\}$
$\mid \mathsf{u}.\mathsf{receive} \mid \mathsf{u}.\mathsf{send}(\mathsf{e}) \mid \mathsf{u}.\mathsf{receiveS}(\mathsf{x})\{\mathsf{e}\} \mid \mathsf{u}.\mathsf{sendS}(\mathsf{u})$
$\mid \mathsf{u}.\mathsf{receiveIf}\,\{\mathsf{e}\}\{\mathsf{e}\} \mid \mathsf{u}.\mathsf{sendIf}(\mathsf{e})\{\mathsf{e}\}\{\mathsf{e}\}$
$\mid \mathsf{u}.\mathsf{receiveWhile}\,\{\mathsf{e}\} \mid \mathsf{u}.\mathsf{sendWhile}(\mathsf{e})\{\mathsf{e}\}$

(identifier) $\mathsf{u} ::= \mathsf{c} \mid \mathsf{x}$

(value) $\mathsf{v} ::= \mathsf{c} \mid \mathsf{null} \mid \mathsf{true} \mid \mathsf{false} \mid$ o

(thread) $P ::=$ e $\mid P \mid P$

(heap) $h ::= \emptyset \mid h \cdot [\mathsf{o} \mapsto (C, \tilde{\mathsf{f}} : \tilde{\mathsf{v}})] \mid h \cdot \mathsf{c}$

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

complete, because for every action we can guarantee that the co-action will eventually appear and the communication will take place, again, unless there is divergence or an exception. Note that exception means a null-pointer exception or a connection error.

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 [19] with the addition of imperative and communication primitives similar to those from [2, 5, 8, 16, 18, 27]. We designed MOOSE as a multi-threaded concurrent language for simplicity of presentation although it can be extended to model distribution; see § 8.

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 active; data may be transmitted through such live 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*.

Types The metavariable t ranges over types for expressions, C ranges over class names and s ranges over session types.

Session types are the types of shared channels. Each 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.

Session parts, ranged over π, represent communications and their sequencing; the symbol ! means *output*, while ? means *input*, and $\dagger$ is a convenient abbreviation that ranges over $\{!,?\}$. As expected, $!\mathsf{t}$ and $?\mathsf{t}$ respectively express the sending and reception of a value of type t, while $!(\rho)$ and $?(\rho)$ represent the exchange of a live channel, and therefore of an active session, with remaining communications determined by type ρ.

The *conditional* session part has the shape $\dagger\langle \pi_1, \pi_2 \rangle$: when $\dagger$ is ! this type 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 that the boolean that determines the branch is to be received instead.

The *iterative* session part $\dagger\langle \pi \rangle^*$ 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 following session parts, if any.

Session parts can be composed into sequences using '.', hence forming longer session parts inductively; note that we use ε for the empty sequence.

A *complete session part* is a session part concatenated either with end or with a conditional whose branches in turn are both complete session parts. We use ρ to range over complete session parts.

Each session type s has one corresponding *dual*, denoted $\overline{\mathsf{s}}$, which is obtained by replacing each ! by ? and vice versa. Note that the carried type itself is *not* dualised, for example $\overline{\mathsf{begin}.?(\rho).\mathsf{end}} = \mathsf{begin}.!(\rho).\mathsf{end}$. The pair $(\mathsf{s}, \overline{\mathsf{s}})$ is the type of a shared channel in which both directions of communication (*i.e.*, s and $\overline{\mathsf{s}}$) can occur. This type is used in the expression $\mathsf{new}\ (\mathsf{s}, \overline{\mathsf{s}})$ to build a fresh channel which can be used to establish a private session, as we will see in Example 4.1.

Class and method declarations Class and method declarations are as expected, except for the restriction that at most one parameter can be a live channel. Example 5.4 shows that allowing multiple live channels as parameters can lead to deadlock.

User syntax The syntax of user expressions e, e' is standard with the exception of the channel constructor $\mathsf{new}\ (\mathsf{s}, \overline{\mathsf{s}})$ discussed above, and the *communication expressions*, *i.e.*, $\mathsf{connect}\ \mathsf{u}\ \mathsf{s}\{\mathsf{e}\}$ and all the expressions in the last three lines. The first line gives parameter, value, the receiver this, sequence of expressions, assignment to fields, field access, method call, object and channel creation. The values are channels, null, and the literals true and false. Thread creation is declared using $\mathsf{spawn}\{\ \mathsf{e}\ \}$, in which the expression e is called the *thread body*.

The expression $\mathsf{connect}\ \mathsf{u}\ \mathsf{s}\{\mathsf{e}\}$ starts a session: the channel u appears within the term $\{\mathsf{e}\}$ as the subject of 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 "$\mathsf{u}.\_$"; we call u the *subject* of such expressions. In the above explanation session expressions are pairwise coupled: we say that expressions in the same pair and with the same subject are *dual* of each other.

The first pair is for exchange of values (which can be shared channels): $\mathsf{u}.\mathsf{receive}$ receives a value via u, while $\mathsf{u}.\mathsf{send}(\mathsf{e})$ evaluates e and sends the result over u.

The second pair expresses live channel exchange: in $\mathsf{u}.\mathsf{receiveS}(\mathsf{x})\{\mathsf{e}\}$ the received channel will be bound to x within e, in which x is used for communications. The expression $\mathsf{u}.\mathsf{sendS}(\mathsf{u}')$ sends the channel u' over u.

The third pair is for *conditional* communication: $\mathsf{u}.\mathsf{receiveIf}\{\mathsf{e}\}\{\mathsf{e}'\}$ receives a boolean value via channel u, and if it is true continues with e, otherwise with e'; $\mathsf{u}.\mathsf{sendIf}(\mathsf{e})\{\mathsf{e}'\}\{\mathsf{e}''\}$ 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: $\mathsf{u}.\mathsf{receiveWhile}\{\mathsf{e}\}$ receives a boolean value via channel u, and if it is true continues with e and iterates, otherwise ends; $\mathsf{u}.\mathsf{sendWhile}(\mathsf{e})\{\mathsf{e}'\}$ 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.

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

Heaps, ranged over h, are built inductively using the heap composition operator '$\cdot$', and contain mappings of object identifiers to instances of classes, and channels. In particular, a heap will contain the set of objects and *fresh* channels, both shared and live, that have been created since the beginning of execution. The heap produced by composing $h \cdot [\mathsf{o} \mapsto (C, \tilde{\mathsf{f}} : \tilde{\mathsf{v}})]$ will map o to the object $(C\tilde{\mathsf{f}} : \tilde{\mathsf{v}})$, where C is the class name and $\tilde{\mathsf{f}} : \tilde{\mathsf{v}}$ is a representation for the vector of distinct mappings from field names to their values for this instance. The heap produced by composing $h \cdot \mathsf{c}$ 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 access is denoted by $h(\mathsf{o})$ for objects, and returns the object pointed to in the heap by the given identifier, or $\bot$ if there is no such mapping. Heap update for objects is written $h[\mathsf{o} \mapsto (C, \tilde{\mathsf{f}} : \tilde{\mathsf{v}})]$; note that in this case we assume that the object identifier, whose mapping is to be updated, is already in the heap.

4 Operational Semantics

This section presents the operational semantics of MOOSE, which is inspired by the standard small step call-by-value reduction of [2, 3, 23] and mainly of [8]. We only discuss the more interesting rules. We start by listing the evaluation contexts.

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

Notice that $\mathsf{connect}\ \mathsf{u}\ \mathsf{s}\{E\}$, $\mathsf{u}.\mathsf{receiveS}(\mathsf{x})\{E\}$, $\mathsf{u}.\mathsf{sendIf}(\mathsf{e})\{E\}\{\mathsf{e}\}$, $\mathsf{u}.\mathsf{sendIf}(\mathsf{e})\{\mathsf{e}\}\{E\}$, $\mathsf{u}.\mathsf{receiveIf}\{E\}\{\mathsf{e}\}$, $\mathsf{u}.\mathsf{receiveIf}\{\mathsf{e}\}\{E\}$, $\mathsf{u}.\mathsf{receiveWhile}\{E\}$, and $\mathsf{u}.\mathsf{sendWhile}(\mathsf{e})\{E\}$ 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.

$$\textbf{Fld}\ \frac{h(\mathsf{o}) = (C, \tilde{\mathsf{f}} : \tilde{\mathsf{v}})}{\mathsf{o}.\mathsf{f}_\mathsf{i}, h \longrightarrow \mathsf{v}_\mathsf{i}, h} \qquad \textbf{Seq}\ \frac{\mathsf{e}_1, h \longrightarrow \mathsf{v}, h'}{\mathsf{e}_1; \mathsf{e}_2, h \longrightarrow \mathsf{e}_2, h'} \qquad \textbf{FldAss}\ \frac{h' = h[\mathsf{o} \mapsto h(\mathsf{o})[\mathsf{f} \mapsto \mathsf{v}]]}{\mathsf{o}.\mathsf{f} := \mathsf{v}, h \longrightarrow \mathsf{v}, h'}$$

$$\textbf{NewC}\ \frac{\mathsf{fields}(C) = \tilde{\mathsf{f}}\tilde{\mathsf{t}} \quad \mathsf{o} \notin h}{\mathsf{new}\ C, h \longrightarrow \mathsf{o}, h \cdot [\mathsf{o} \mapsto (C, \tilde{\mathsf{f}} : \tilde{\mathsf{init}}(\mathsf{t}))]} \qquad \textbf{NewS}\ \frac{\mathsf{c} \notin h}{\mathsf{new}\ (\mathsf{s}, \overline{\mathsf{s}}), h \longrightarrow \mathsf{c}, h \cdot \mathsf{c}} \qquad \textbf{Cong}\ \frac{\mathsf{e}, h \longrightarrow \mathsf{e}', h'}{E[\mathsf{e}], h \longrightarrow E[\mathsf{e}'], h'}$$

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

$$\textbf{NullFldAss}\ \mathsf{null}.\mathsf{f} := \mathsf{v}, h \longrightarrow \mathsf{NullExc}, h \qquad \textbf{NullFld}\ \mathsf{null}.\mathsf{f}, h \longrightarrow \mathsf{NullExc}, h \qquad \textbf{NullMeth}\ \mathsf{null}.\mathsf{m}(\tilde{\mathsf{v}}), h \longrightarrow \mathsf{NullExc}, h$$

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

Fig. 5. Expression Reduction

Expressions Fig. 5 shows the rules for execution of expressions which correspond to the sequential part of the language. These are standard [3, 9, 19], but for the addition of a fresh live 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 $\tilde{\mathsf{v}}$.

Threads The reduction rules for threads, shown in Fig. 6, are given modulo the standard structural equivalence rules of the π-calculus [21], written $\equiv$. We define *multi-step* reduction as: $\longrightarrow\!\!\!\!\rightarrow \stackrel{\mathsf{def}}{=} (\longrightarrow \cup \equiv)^*$.

When $\mathsf{spawn}\{\ \mathsf{e}\ \}$ 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'.

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 variable x within 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 4.2.

Struct $P \mid \mathsf{null} \equiv P \qquad P \mid P_0 \equiv P_0 \mid P \qquad P \mid (P_0 \mid P_1) \equiv (P \mid P_0) \mid P_1$

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

Par
$$\frac{P, h \longrightarrow P', h'}{P \mid P_0, h \longrightarrow P' \mid P_0, h'}$$

Str
$$\frac{P_1, h \longrightarrow P_2, h' \quad P_i \equiv P_i' \quad i \in \{1,2\}}{P_1', h \longrightarrow P_2', h'}$$

Connect
$E_1[\mathsf{connect\ c\ s}\{\mathsf{e}_1\}] \mid E_2[\mathsf{connect\ c}\ \overline{\mathsf{s}}\{\mathsf{e}_2\}], h \longrightarrow E_1[\mathsf{e}_1[\mathsf{c}'/\mathsf{c}]] \mid E_2[\mathsf{e}_2[\mathsf{c}'/\mathsf{c}]], h \cdot \mathsf{c}' \quad \mathsf{c}' \notin h$

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

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

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

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

ComSWhile
$E_1[\mathsf{c.sendWhile}(\mathsf{e})\{\mathsf{e}_1\}] \mid E_2[\mathsf{c.receiveWhile}\ \{\mathsf{e}_2\}], h \longrightarrow$
$E_1[\mathsf{c.sendIf}(\mathsf{e})\{\mathsf{e}_1; \mathsf{c.sendWhile}(\mathsf{e})\{\mathsf{e}_1\}\}\{\mathsf{null}\}] \mid E_2[\mathsf{c.receiveIf}\ \{\mathsf{e}_2; \mathsf{c.receiveWhile}\ \{\mathsf{e}_2\}\}\{\mathsf{null}\}], h$

Fig. 6. Thread Communication

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 [6, 8]).

The following two examples justify some aspects of our operational semantics.

Example 4.1. demonstrates asynchronous call-backs using new channel creation and spawn. When the method is called with $\mathsf{o.m}(\ \mathsf{new}\ \ (\mathsf{s}, \overline{\mathsf{s}}))$, then after line 2, there will be two threads, which will then communicate over the private channel x; the first thread will execute e, and the second thread will execute e'.

```
1  int m( (s,s̄) x ) {
2      spawn { connect x s { e } };
3      connect x s̄ { e′ }
4  }
```

Example 4.2. 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 the threads P_1 and P_2 in the table below reduce to

$$\mathsf{c}'_1.\mathsf{send}(5); \mathsf{c}'_1.\mathsf{receive} \mid \mathsf{null}, \quad h \cdot \mathsf{c}1'$$

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

```
1 connect c1 begin.?int.end {
2  connect c2 begin.?(!int.end).end {
3   c2.receiveS(x) { x.send(5)} };
4   c1.receive
5 }
```

P_1

```
1 connect c1 begin.!int.end{
2  connect c2 begin.!(!int.end).end {
3   c2.sendS(c1)
4  }
5 }
```

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.

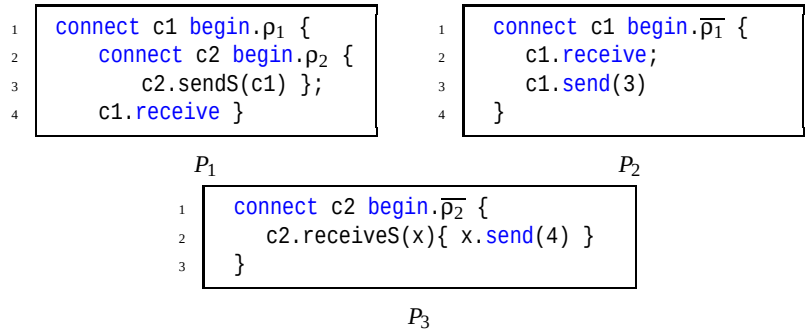
Type preservation In order to achieve subject reduction, we need to ensure that at any time during execution, no more than two threads have access to the same live channel, and also, that no thread has aliases (*i.e.,* more than one reference) to a live channel.

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 in the threads P_1, P_2 and P_3 the variables x, y and z, all point to the same live channel c in heap h.

$$\underbrace{\mathsf{x}.\mathsf{send}(3);\mathsf{x}.\mathsf{send}(\mathsf{true})}_{P_1} \mid \underbrace{\mathsf{y}.\mathsf{send}(4);\mathsf{y}.\mathsf{send}(\mathsf{false})}_{P_2} \mid \underbrace{\mathsf{z}.\mathsf{receive};\mathsf{z}.\mathsf{receive}}_{P_3}, \quad h$$

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. Therefore, we need to distinguish a *shared* channel, *i.e.,* one where a connection has not been established yet, from a *live* channel, *i.e.,* one where a connection has been established. In order to make this distinction, shared channel types start with begin. 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 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 ρ_1 and ρ_2,



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

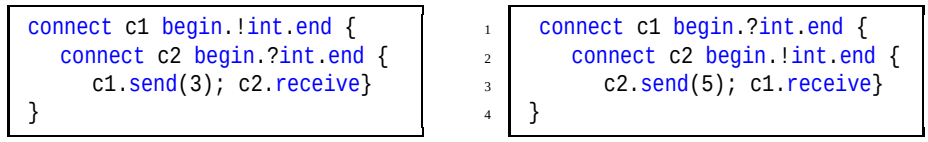
$$\mathsf{c}_1'.\mathsf{receive} \mid \mathsf{c}_1'.\mathsf{receive}\,;\,\mathsf{c}_1'.\mathsf{send}\,(3) \mid \mathsf{c}_1'.\mathsf{send}\,(4), \quad h \cdot \{\mathsf{c}_1', \mathsf{c}_2'\}$$

where c_1' and c_2' are the fresh live channels that replaced respectively c_1 and c_2 when the sessions began. Clearly, this configuration violates the bilinearity condition.

We therefore need a notion of whether a live channel has been *consumed*, *i.e.*, whether it can still be 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, the types do distinguish consumed channels, because the type of a closed channel is suffixed with end. In § 6.1 we show that P_1 is type incorrect for any ρ_1 and ρ_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. 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. demonstrates how session interleaving may cause deadlocks.



In the above example we have indefinite waiting after establishing the connection, because P_1 cannot progress unless P_2 reaches the statement c_1.receive, and P_2 cannot progress unless P_1 reaches the statement c_2.receive, and so we have a deadlock at a communication point. Note that *nesting* of sessions, does not affect progress. Let us consider the following processes.

$$P_1' = \mathsf{connect}\ \mathsf{c}_1\ \mathsf{begin}.?\mathsf{int}.\mathsf{end}\{\mathsf{c}_1.\mathsf{receive}\,;\mathsf{connect}\ \mathsf{c}_2\ \mathsf{begin}.!\mathsf{int}.\mathsf{end}\{\mathsf{c}_2.\mathsf{send}\,(5)\}\}$$
$$P_2' = \mathsf{connect}\ \mathsf{c}_1\ \mathsf{begin}.!\mathsf{int}.\mathsf{end}\{\mathsf{c}_1.\mathsf{send}\,(3)\,;\mathsf{connect}\ \mathsf{c}_2\ \mathsf{begin}.?\mathsf{int}.\mathsf{end}\{\mathsf{c}_2.\mathsf{receive}\}\}$$
$$P_3' = \mathsf{connect}\ \mathsf{c}_1\ \mathsf{begin}.!\mathsf{int}.\mathsf{end}\{\mathsf{connect}\ \mathsf{c}_2\ \mathsf{begin}.?\mathsf{int}.\mathsf{end}\{\mathsf{c}_2.\mathsf{receive}\}\,;\mathsf{c}_1.\mathsf{send}\,(3)\}$$

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 $\mathtt{c}_2$. However, deadlock at connection points is acceptable, since it will disappear by putting a suitable 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. In § 6.1, we show that P_1 and P_2 are type incorrect.

Example 5.4. demonstrates that allowing methods with multiple live channel parameters may cause interleaving. Consider a method m of class C with two parameters x and y both of type ?int and body x.receive;y.receive. In this case the two threads P_1 and P_2 below produce a deadlock due to the interleaving of sessions.

```
1 connect c1 begin.!int.end {
2   connect c2 begin.!int.end {
3     c1.send(3); c2.send(3)}
4 }
```

P_1

```
1 connect c1 begin.?int.end {
2   connect c2 begin.?int.end {
3       new C.m(c2,c1)}
4 }
```

P_2

In order to avoid problems like the above, we restrict the number of live channel parameters to at most one.

We argue that the above conditions on live channels are not that restrictive. First, we can represent most of the communication protocols in the session types literature, as well as traditional synchronisation [22, § 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, we will obtain a more relaxed type system which allows deadlock on live channels, but still preserves type safety.

6 Type System

We type expressions and threads with respect to a fixed class table, so only the classes declared in this table are types. We use the same table to judge subtyping $<:$ on class names: we assume the subtyping between classes causes no cycle as in [19]. In addition, we have $(\mathtt{s},\overline{\mathtt{s}}) <: \mathtt{s}$ and $(\mathtt{s},\overline{\mathtt{s}}) <: \overline{\mathtt{s}}$, as in standard π-calculus channel subtyping rules [17]: a channel on which both communication directions are allowed may also transmit data following only one of the two directions.

The typing judgements for threads have two environments, *i.e.*, they have the shape:

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

where the *standard environment* Γ associates types to this, parameters and objects, while the *session environment* Σ contains only judgements for live channels. These environments are defined as follows, under the condition that no subject occurs twice.

$$\Gamma ::= \emptyset \mid \Gamma,\mathtt{x}:\mathtt{t} \mid \Gamma,\mathtt{this}:C \mid \Gamma,\mathtt{o}:C \qquad \Sigma ::= \emptyset \mid \Sigma,\mathtt{u}:\eta \mid \Sigma,\mathtt{u}:\updownarrow$$

When typing expressions we need also to take into account which is the unique (if any) channel identifier currently used to communicate data. This is necessary in order

to avoid session interleaving. Therefore we record a third set, the *hot set* $\mathcal{S}$, which can be either empty or can contain a single channel identifier belonging to the session environment. Thus the typing judgements for expressions have the shape:

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

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

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

Spawn

$$\frac{\Gamma;\Sigma;\mathcal{S} \vdash \mathsf{e} : \mathsf{t} \quad \mathit{closed}(\Sigma)}{\Gamma;\Sigma;\mathcal{S} \vdash \mathsf{spawn}\{\ \mathsf{e}\ \} : \mathit{Object}}$$

Weak

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

Meth

$$\frac{\Gamma;\Sigma_0;\mathcal{S} \vdash \mathsf{e} : C \qquad \Gamma;\Sigma_i;\mathcal{S} \vdash \mathsf{e}_i : \mathsf{t}_i \quad i \in \{1 \ldots n\}}{\Gamma;\Sigma_0.\Sigma_1 \ldots \Sigma_n;\mathcal{S} \vdash \mathsf{e}.\mathsf{m}(\mathsf{e}_1 \ldots \mathsf{e}_n) : \mathsf{t}} \quad \mathsf{mtype}(\mathsf{m}, C) = \mathsf{t}_1 \ldots \mathsf{t}_n \to \mathsf{t}$$

MethLin

$$\frac{\Gamma;\Sigma_0;\{\mathsf{u}\} \vdash \mathsf{e} : C \qquad \Gamma;\Sigma_i;\{\mathsf{u}\} \vdash \mathsf{e}_i : \mathsf{t}_i \quad i \in \{1 \ldots n\}}{\Gamma;\Sigma_0.\Sigma_1 \ldots \Sigma_n.\{\mathsf{u} : \eta\};\{\mathsf{u}\} \vdash \mathsf{e}.\mathsf{m}(\mathsf{e}_1 \ldots \mathsf{e}_n, \mathsf{u}) : \mathsf{t}} \quad \mathsf{mtype}(\mathsf{m}, C) = \mathsf{t}_1 \ldots \mathsf{t}_n, \eta \to \mathsf{t}$$

Fig. 7. Some Typing Rules for Standard Expressions

Expressions We highlight the interesting typing rules for expressions in Fig. 7 and Fig. 8. Looking at these rules two observations on hot sets are immediate:

- in all the rules but **Conn**, **ReceiveS** and **Weak** the hot sets of all the premises and of the conclusion coincide;
- in all the 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 all communications use the same live channel, *i.e.*, that sessions are not interleaved. In rule **Conn** instead the live channel becomes shared and therefore in the conclusion the hot set is empty. Since $\mathsf{u}.\mathsf{receiveS}(\mathsf{x})\{\mathsf{e}\}$ in rule **ReceiveS** receives along the live channel u a channel that will be replaced to x, the hot set of the premise is $\{\mathsf{x}\}$ while that of the conclusion is $\{\mathsf{u}\}$. Lastly rule **Weak** allows to replace an empty hot set by a set containing an arbitrary element of the domain of the session environment.

The session environments of the conclusions are obtained from those of the premises and possibly other session environments using the *concatenation* defined below.

- $\eta.\eta' = \eta.\eta'$ if $\eta = \pi$ or $\eta' = \varepsilon$ otherwise $\eta.\eta' = \bot$.
- $\Sigma.\Sigma' = \Sigma \setminus \mathsf{dom}(\Sigma') \cup \Sigma' \setminus \mathsf{dom}(\Sigma) \cup \{\mathsf{u} : \Sigma(\mathsf{u}).\Sigma'(\mathsf{u}) \mid \mathsf{u} \in \mathsf{dom}(\Sigma) \cap \mathsf{dom}(\Sigma')\}$

Conn

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

Send

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

Receive

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

ReceiveS

$$\frac{\Gamma \setminus \mathsf{x}\ ;\ \Sigma, \mathsf{x} : \rho; \{\mathsf{x}\} \vdash \mathsf{e} : \mathsf{t} \quad \mathit{closed}(\Sigma)}{\Gamma;\{\mathsf{u} : ?(\rho)\}.\Sigma;\{\mathsf{u}\} \vdash \mathsf{u}.\mathsf{receiveS}(\mathsf{x})\{\mathsf{e}\} : \mathit{Object}}$$

SendS

$$\frac{\Gamma \vdash \mathsf{ok} \quad \vdash \rho : \mathsf{tp}}{\Gamma;\{\mathsf{u}' : \rho, \mathsf{u} : !(\rho)\};\{\mathsf{u}\} \vdash \mathsf{u}.\mathsf{sendS}(\mathsf{u}')\ :\ \mathit{Object}}$$

ReceiveIf

$$\frac{\Gamma;\Sigma, \mathsf{u} : \eta_i;\{\mathsf{u}\} \vdash \mathsf{e}_i : \mathsf{t} \quad i \in \{1,2\}}{\Gamma;\Sigma, \mathsf{u} :?\langle \eta_1, \eta_2 \rangle;\{\mathsf{u}\} \vdash \mathsf{u}.\mathsf{receiveIf}\{\mathsf{e}_1\}\{\mathsf{e}_2\} : \mathsf{t}}$$

SendIf

$$\frac{\Gamma;\Sigma_1;\{\mathsf{u}\} \vdash \mathsf{e} : \mathsf{bool} \qquad \Gamma;\Sigma_2, \mathsf{u} : \eta_i;\{\mathsf{u}\} \vdash \mathsf{e}_i : \mathsf{t} \quad i \in \{1,2\}}{\Gamma;\Sigma_1.\{\Sigma_2, \mathsf{u} :!\langle \eta_1, \eta_2 \rangle\};\{\mathsf{u}\} \vdash \mathsf{u}.\mathsf{sendIf}(\mathsf{e})\{\mathsf{e}_1\}\{\mathsf{e}_2\} : \mathsf{t}}$$

ReceiveWhile

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

SendWhile

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

Fig. 8. Typing Rules for Communication Expressions

The concatenation of two live channel types η and η' is the unique live channel type (if it exists) which prescribes all the communications of η followed by all those of η'. The extension to session environments is straightforward. The typing rules concatenate the session environments to take into account the order of execution of expressions.

In the following we discuss the three 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 complete live channel types. This is necessary in order to avoid configurations that break the bilinearity condition, such as $\mathsf{spawn}\{\ \mathsf{c}.\mathsf{send}\,(1)\ \};\mathsf{c}.\mathsf{send}\,(\mathsf{true})$. To guarantee the consumption we define

$$\mathit{closed}(\Sigma) = \forall \mathsf{u} : \eta \in \Sigma\ \exists \rho.\ \eta = \rho$$

Rule **MethLin** retrieves the type of the method m from the class table using the auxiliary function $\mathsf{mtype}(\mathtt{m}, C)$. This rule expects the last actual parameter u 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 $\{\mathsf{u}\}$. The session environments of the premises are also concatenated with $\{\mathsf{u} : \eta\}$ which represents the communication protocol of the live channel u during the execution of the method body.

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 (u) can be typed with the appropriate shared session type ($\mathsf{begin}.\rho$). The second premise ensures that the session body can be typed in the restricted environment $\Gamma \setminus \mathsf{u}$ with a session environment containing $\mathsf{u} : \rho$ and with hot set $\{\mathsf{u}\}$.

Methods The following rules define well-formed methods.

$$\textbf{M-ok}\quad \frac{\mathtt{this}:C,\tilde{x}:\tilde{t}\,;\,\emptyset;\,\emptyset\vdash e:t}{t\,m\ (\tilde{t}\,\tilde{x})\ \{e\}:\mathtt{ok}\ \mathtt{in}\ C} \qquad \textbf{MLin-ok}\quad \frac{\mathtt{this}:C,\tilde{x}:\tilde{t}\,;\,x:\eta;\,\{x\}\vdash e:t}{t\,m\ (\tilde{t}\,\tilde{x},\eta x)\ \{e\}:\mathtt{ok}\ \mathtt{in}\ C}$$

Rule **M-ok** checks that a method that does not have live channel parameters is well-formed, by type-checking its body and succeeding with both an empty session environment and an empty hot set *i.e.*, it ensures that no channel can be used outside the scope of a session within the method body. Rule **MLin-ok**, performs the same check but requires that the last parameter is a live channel which is the element of the hot set in the typing of the method body.

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 premises using the *parallel composition* defined below.

- $\eta\|\eta' = \updownarrow$ if $\eta = \overline{\eta'}$ otherwise $\eta\|\eta' = \bot$; and $\updownarrow \| \eta = \eta \| \updownarrow = \updownarrow \| \updownarrow = \bot$.
- $\Sigma\|\Sigma' = \Sigma \setminus \mathsf{dom}(\Sigma') \cup \Sigma' \setminus \mathsf{dom}(\Sigma) \cup \{u : \Sigma(u) \,\|\, \Sigma'(u) \mid u \in \mathsf{dom}(\Sigma) \cap \mathsf{dom}(\Sigma')\}$

Using the above operator, the typing rules for processes are straightforward. 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.

$$\textbf{Start}\quad \frac{\Gamma;\Sigma;\mathcal{S}\vdash e:t}{\Gamma;\Sigma\vdash e:\mathsf{thread}} \qquad \textbf{Par}\quad \frac{\Gamma;\Sigma\vdash P:\mathsf{thread}\quad \Gamma;\Sigma'\vdash P':\mathsf{thread}}{\Gamma;\Sigma\|\Sigma'\vdash P\mid P':\mathsf{thread}}$$

6.1 Justifying Examples

In this subsection we discuss the typing of the threads shown in § 5.

Example 5.1: The thread $P_1 \mid P_2$ 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;
- **Seq** to type the composition of these two expressions requires concatenation of the corresponding two 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: Neither thread can be typed. For example, to type the expressions in line 3 in P_1 using rule **Send**, $\{c_1\}$ and $\{c_2\}$ should be the hot sets, respectively. Thus rule **Seq** cannot type the composition of these two expressions, since this rule requires the premises to share the same hot set.

Example 5.4: It is clear from the rules **Meth** and **MethLin** that a method can have at most one live parameter, so the method is not typable.

6.2 Type Inference

We have shown that the type system gives enough flexibility to typecheck interesting protocols. On the other hand, the structure of the typing judgements is somewhat complex: ideally, one would like to be able to have only standard environments, without having to worry about sessions environments and hot sets. We show in fact that this is the case, by giving appropriate inference rules. The inference judgements derived by these rules have the shapes

$$\Gamma \vdash \mathsf{e} : \mathsf{t} \parallel \Sigma; \mathcal{S} \qquad \text{and} \qquad \Gamma \vdash P : \mathsf{thread} \parallel \Sigma$$

respectively for expressions and threads. This means that sessions environments and hot sets are derived instead than assumed. Fig. 9 gives some inference rules. All rules are applicable only if all the sets in the conclusion are defined.

ConnI

$$\frac{\Gamma \vdash \mathsf{e} : \mathsf{t} \parallel \Sigma; \mathcal{S} \quad \Sigma(\!(\mathsf{u})\!) = \eta \quad \mathsf{s} = \mathsf{begin}.\sigma(\eta\downarrow) \quad \mathsf{u} \notin \mathsf{dom}(\Gamma) \quad \Gamma' = \Gamma \text{ if } \mathsf{u} \text{ is a name else } \Gamma, \mathsf{u} : \mathsf{s}}{\Gamma' \vdash \mathsf{connect}\ \mathsf{u}\ \mathsf{s}\ \{\mathsf{e}\} : \sigma(\mathsf{t}) \parallel \sigma(\Sigma) \setminus \mathsf{u}; \emptyset}$$

ReceiveI

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

ReceiveSI

$$\frac{\Gamma \vdash \mathsf{e} : \mathsf{t} \parallel \Sigma; \mathcal{S} \quad \mathsf{x} \notin \Gamma \quad \mathcal{S} \subseteq \{\mathsf{x}\} \quad \Sigma(\!(\mathsf{x})\!) = \eta}{\Gamma \vdash \mathsf{u}.\mathsf{receiveS}\,(\mathsf{x})\{\mathsf{e}\} : \mathit{Object} \parallel \{\mathsf{u} : ?(\eta\downarrow)\}.\Sigma\downarrow; \{\mathsf{u}\}}$$

SendSI

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

Fig. 9. Some Inference Rules

We extend the syntax of types with two kinds of variables, *type variables* and *part of session type variables*, ranged over respectively by ϕ and ψ. The type variables stand for types and the part of session type variables stand for part of session types. Rule **ReceiveI** introduces type variables since we do not know the type of the data that will be received. Rule **SendSI** introduces part of session type variables since we do not know the type of the channel that will be send.

Comparing the typing with the inference rules it should be clear that we only have to take into account the absence of non-structural rules as usual (*i.e.*, rules which do not depend on the shape of the subject). We only comment some notation. In rule **ConnI** we do not know if the session environment inferred for e contains a premise for u: thus we define

$$\Sigma(\!(\mathsf{u})\!) = \text{ if } \mathsf{u} \in \mathsf{dom}(\Sigma) \text{ then } \Sigma(\mathsf{u}) \text{ else } \varepsilon.$$

In the same rule the operator $\downarrow$ appends end to η if η is a session part, propagates inside the final branches of η if η is of the shape $\pi.\natural\langle\eta_1, \eta_2\rangle$, and does nothing otherwise. These operators are also used in rule **ReceiveSI**.

An *inference substitution* is a substitution mapping type variables to types and part of session type variables to part of session types. We use σ to range over inference substitutions. We use inference substitution only in rule **ConnI** in order to unify the session type s with $\mathsf{begin}.\eta$ where $\eta\downarrow$ being inferred may contain variables. That is, we require $\mathsf{s} = \mathsf{begin}.\sigma(\eta\downarrow)$.

The following proposition (whose proof by induction on deductions is standard) relates the type system and the inference system as expected, showing that the inference computes the least sessions environments and hot sets.

Proposition 6.1. *1. If* $\Gamma;\Sigma;\mathcal{S} \vdash \mathsf{e}:\mathsf{t}$ *then* $\Gamma \vdash \mathsf{e}:\mathsf{t}' \parallel \Sigma';\mathcal{S}'$ *where* $\sigma(\mathsf{t}') = \mathsf{t}$ *and* $\sigma(\Sigma') \subseteq \Sigma$ *for some inference substitution* σ *and* $\mathcal{S}' \subseteq \mathcal{S}$.
2. If $\Gamma \vdash \mathsf{e}:\mathsf{t} \parallel \Sigma;\mathcal{S}$ *then for all inference substitutions* σ *we get:* $\Gamma;\sigma(\Sigma);\mathcal{S} \vdash \mathsf{e}:\sigma(\mathsf{t})$.
3. If $\Gamma;\Sigma \vdash P:\mathsf{thread}$ *then* $\Gamma \vdash P:\mathsf{thread} \parallel \Sigma'$ *where* $\sigma(\Sigma') \subseteq \Sigma$ *for some inference substitution* σ.
4. If $\Gamma \vdash P:\mathsf{thread} \parallel \Sigma$ *then for all inference substitutions* σ *we get:* $\Gamma;\sigma(\Sigma) \vdash P:\mathsf{thread}$.

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

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset \vdash 5:\mathsf{int} \parallel \emptyset;\emptyset}}{\emptyset \vdash \mathsf{x}.\mathsf{send}(5):\mathit{Object} \parallel \{\mathsf{x}:!\mathsf{int}\};\{\mathsf{x}\}}}{\emptyset \vdash \mathsf{c}_2.\mathsf{receiveS}(\mathsf{x})\{\mathsf{x}.\mathsf{send}(5)\}:\mathit{Object} \parallel \{\mathsf{c}_2:?(!\mathsf{int}.\mathsf{end})\};\{\mathsf{c}_2\}} \quad \dfrac{}{\emptyset \vdash \mathsf{e}:\mathit{Object} \parallel \emptyset;\emptyset}}{\dfrac{\emptyset \vdash \mathsf{c}_1.\mathsf{receive}:\phi \parallel \{\mathsf{c}_1:?\phi\};\{\mathsf{c}_1\}}{\emptyset \vdash \mathsf{e};\mathsf{c}_1.\mathsf{receive}:\phi \parallel \{\mathsf{c}_1:?\phi\};\{\mathsf{c}_1\}}}}{\emptyset \vdash \mathsf{connect}\ \mathsf{c}_1\ \mathsf{begin}.?\mathsf{int}.\mathsf{end}\{\mathsf{e};\mathsf{c}_1.\mathsf{receive}\}:\mathsf{int} \parallel \emptyset;\emptyset}$$

where $\mathsf{e} = \mathsf{connect}\ \mathsf{c}_2\ \mathsf{begin}.?(!\mathsf{int}.\mathsf{end}).\mathsf{end}\{\mathsf{c}_2.\mathsf{receiveS}(\mathsf{x})\{\mathsf{x}.\mathsf{send}(5)\}\}$

Fig. 10. An example of type inference

As an example we show the inference for the thread P_1 of Example 4.2 in Fig. 10.

We could also easily modify the inference rules in such a way such that the session types in the connect expressions are inferred. It is enough to modify the inference rule for connect avoiding to use the inference substitution for obtaining the required session types. More precisely the new inference rule is:

$$\textbf{ConnI}' \quad \frac{\Gamma \vdash \mathsf{e}:\mathsf{t} \parallel \Sigma;\mathcal{S} \quad \Sigma(\!(\mathsf{u})\!) = \eta \quad \mathsf{u} \notin \mathsf{dom}(\Gamma) \quad \Gamma' = \Gamma \text{ if } \mathsf{u} \text{ is a name else } \Gamma, \mathsf{u}:\mathsf{s}}{\Gamma' \vdash \mathsf{connect}\ \mathsf{u}\ \mathsf{begin}.\eta\downarrow\ \{\mathsf{e}\}:\mathsf{t} \parallel \Sigma \setminus \mathsf{u};\emptyset}$$

Thanks to this rule, users do not have to declare the session type explicitly in connect; for example, the user can write only: $\mathsf{connect}\ \mathsf{c}\ \{\mathsf{c}.\mathsf{send}(\mathsf{true});\mathsf{c}.\mathsf{send}(\mathsf{false})\}$ instead

of connect c begin.!bool.!bool.end $\{\mathsf{c}.\mathsf{send}(\mathsf{true});\mathsf{c}.\mathsf{send}(\mathsf{false})\}$. However we think the explicit declaration is sensible as an aid for design and documentation, as well as for determining the types of the data exchanged in the protocol communications.

7 Type Safety and Communication Safety

7.1 Subject Reduction

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 $\Gamma \vdash h : \mathtt{ok}$. A convenient notation is $\Gamma;\Sigma;\mathcal{S} \vdash \mathtt{e};h$, which is short for $\Gamma;\Sigma;\mathcal{S} \vdash \mathtt{e} : \mathtt{t}$ for some $\mathtt{t}$ and $\Gamma \vdash h : \mathtt{ok}$. Similarly $\Gamma;\Sigma \vdash P;h$ means $\Gamma;\Sigma \vdash P : \mathtt{thread}$ and $\Gamma \vdash h : \mathtt{ok}$. We first show that the type system of Section 6 satisfies the subject reduction property.

Theorem 7.1 (Subject Reduction).

- $\Gamma;\Sigma;\mathcal{S} \vdash \mathtt{e} : \mathtt{t}$, *and* $\Gamma;\Sigma;\mathcal{S} \vdash \mathtt{e};h$ *and* $\mathtt{e},h \longrightarrow \mathtt{e}',h'$ *imply* $\Gamma';\Sigma';\mathcal{S} \vdash \mathtt{e}';h'$ *and* $\Gamma';\Sigma';\mathcal{S} \vdash \mathtt{e}' : \mathtt{t}$ *with* $\Gamma \subseteq \Gamma'$ *and* $\Sigma \subseteq \Sigma'$.
- $\Gamma;\Sigma \vdash P;h$ *and* $P,h \longrightarrow P',h'$ *imply* $\Gamma';\Sigma' \vdash P';h'$ *with* $\Gamma \subseteq \Gamma'$ *and* $\Sigma \subseteq \Sigma'$.

The proof uses generation lemmas and substitution lemmas in a standard way. The novelty of this proof relies on a detailed analysis of the relations between session environments for typing expressions inside evaluation contexts and the filled contexts. More precisely we introduce a partial order on session environments in Definition 7.2. When proving type preservation for the case $E[\mathtt{e}],h \longrightarrow E[\mathtt{e}'],h'$, we apply Lemma 7.3 to extrapolate properties of the session environment used for typing $\mathtt{e}$ out of that used for typing $E[\mathtt{e}]$. Similarly for the case when two threads communicate by the communication rules in Fig. 6.

Definition 7.2 (Prefix Order on Session Environments).

1. $\eta \preceq \eta'$ is the smallest partial order such that $\pi \preceq \pi.\eta$;
2. $\Sigma \preceq \Sigma'$ if $\mathtt{u} : \eta \in \Sigma$ implies $\mathtt{u} : \eta' \in \Sigma'$ and $\eta \preceq \eta'$.

Notice that $\Sigma \preceq \Sigma'$ iff $\Sigma' = \Sigma.\Sigma''$ for some Σ''.

Lemma 7.3 (Subderivations).

If a derivation $\mathcal{D}$ proves $\Gamma;\Sigma;\mathcal{S} \vdash E[\mathtt{e}] : \mathtt{t}$ then $\mathcal{D}$ contains a subderivation whose conclusion is the typing of the showed occurrence of $\mathtt{e}$*:* $\Gamma;\Sigma';\mathcal{S}' \vdash \mathtt{e} : \mathtt{t}'$ *and* $\Sigma' \preceq \Sigma$.

The proof is by induction on evaluation contexts.

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.

These properties hold for a thread obtained by reducing a well-typed closed thread in which all expressions are user expressions. We write $\prod_{0\leq i<n} e_i$ for $e_0 \mid e_1 \mid \ldots \mid e_{n-1}$. We say a thread P is *initial* if $\emptyset;\emptyset \vdash P : \mathsf{thread}$ is derivable and $P \equiv \prod_{0\leq i<n} e_i$ where e_i is a user expression. Notice that this implies $\emptyset;\emptyset \vdash P;\emptyset$. For stating **P1**, we add a new constant $\mathsf{CommErr}$ (*communication error*) to the syntax and the following rule:

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

if e and e' are session expressions with the same subject and they are not dual of each other. We can now prove that we never reach a state containing such incompatible expressions.

Corollary 7.4 ($\mathsf{CommErr}$ **Freedom).** *Assume P_0 is initial and $P_0, \emptyset \longrightarrow\!\!\!\!\rightarrow P, h$. Then P does not contain* $\mathsf{CommErr}$*, i.e., there does not exist P' such that $P \equiv P' \mid \mathsf{CommErr}$.*

The proof of the above theorem is straightforward from the subject reduction theorem. Next we show that the progress property **P2** holds in our typing system.

Theorem 7.5 (Progress). *Assume P_0 is initial and $P_0, \emptyset \longrightarrow\!\!\!\!\rightarrow P, h$. Then one of the following holds.*

- *In P, all expressions are values, i.e., $P \equiv \prod_{0\leq i<n} v_i$;*
- *$P, h \longrightarrow P', h'$;*
- *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., $P \equiv E[\mathsf{connect}\ c\ s\{e\}] \mid Q$.*

The key in showing progress is the natural correspondence between irreducible session expressions and parts of session types formalised in the following definition.

Definition 7.6. *Define $\propto$ between irreducible session expressions and parts of session types as follows:*

$$c.\mathsf{receive} \propto ?t \quad c.\mathsf{send}(v) \propto !t \quad c.\mathsf{receiveS}(x)\{e\} \propto ?(\rho) \quad c.\mathsf{sendS}(c') \propto !(\rho)$$
$$c.\mathsf{receiveIf}\{e_1\}\{e_2\} \propto ?\langle \eta_1, \eta_2 \rangle \quad c.\mathsf{sendIf}(v)\{e_1\}\{e_2\} \propto !\langle \eta_1, \eta_2 \rangle$$
$$c.\mathsf{receiveWhile}\{e\} \propto ?\langle \pi \rangle^* \quad c.\mathsf{sendWhile}(v)\{e\} \propto !\langle \pi \rangle^*$$

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

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 7.7. *Let e be an irreducible session expression with subject c and $\Gamma;\Sigma \vdash E[e] : \mathsf{thread}$. Then $\Sigma(c) = \pi.\eta$ with $e \propto \pi$.*

The proof of Theorem 7.5 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 expression, e_2. Then by Lemma 7.7, $\mathsf{e}_1 \propto \pi$ and $\mathsf{e}_2 \propto \overline{\pi}$. Therefore e_1 and e_2 are dual of each other and can communicate.

Note that the Theorem 7.5 shows that *threads can always communicate at live channels*. From the above theorem, immediately we get:

Corollary 7.8 (Completion of Sessions). *Assume P_0 is initial and $P_0, \emptyset \longrightarrow\!\!\!\!\!\rightarrow P, h$. Suppose $P \equiv \prod_{0 \leq \mathsf{i} < \mathsf{n}} \mathsf{e}_\mathsf{i}$ and irreducible. Then either all e_i are values $(0 \leq \mathsf{i} < \mathsf{n})$ or there is some j $(0 \leq \mathsf{j} < \mathsf{n})$ such that $\mathsf{e}_\mathsf{j} \in \{\mathsf{NullExc}, E[\mathsf{connect}\ \mathsf{c}\ \mathsf{s}\{\mathsf{e}\}]\}$.*

Finally we state the main property (**P3**) of our typing system. For this purpose, we define the partial order $\sqsubseteq$ on live channel types as the smallest partial order such that:

- $\eta \sqsubseteq \pi.\eta$;
- $\pi_i.\eta \sqsubseteq \dagger\langle \pi_1, \pi_2 \rangle.\eta \quad (i \in \{1,2\})$;
- $\rho_i \sqsubseteq \dagger\langle \rho_1, \rho_2 \rangle \quad (i \in \{1,2\})$;
- $\dagger\langle \pi.\langle \pi \rangle^*, \varepsilon \rangle.\eta \sqsubseteq \langle \pi \rangle^*.\eta$.

This partial order takes into account reduction as formalised in the following theorem: for any configuration $E[\mathsf{e}_0] \,|\, Q, h$ reachable from the initial configuration and containing the irreducible session expression e_0, that if it proceeds, then either (1) it does so in the sub-thread Q, or (2) Q contains the dual expression e'_0, which (a) interacts with e_0, and (b) has a dual type at c (and therefore, through application of Lemma 7.7 the two expressions conform to the "shape" of their type *i.e.*, $\eta = \pi.\eta_0$ with $\mathsf{e}_0 \propto \pi$ and $\mathsf{e}'_0 \propto \overline{\pi}$), and (c) then the type of channel c "correctly shrinks" as $\eta' \sqsubseteq \eta$.

Theorem 7.9 (Communication-Order Preservation). *Let P_0 be initial. Assume that $P_0, \emptyset \longrightarrow\!\!\!\!\!\rightarrow E[\mathsf{e}_0] \,|\, Q, h \longrightarrow P', h'$ 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) *$\Gamma; \Sigma, \mathsf{c} : \eta \vdash E[\mathsf{e}_0] : \mathsf{thread}$ and $\Gamma; \Sigma', \mathsf{c} : \overline{\eta} \vdash E'[\mathsf{e}'_0] : \mathsf{thread}$; and*
   (c) *$\Gamma; \hat{\Sigma}, \mathsf{c} : \eta' \vdash \mathsf{e} : \mathsf{thread}$ and $\Gamma; \hat{\Sigma}', \mathsf{c} : \overline{\eta}' \vdash \mathsf{e}' : \mathsf{thread}$ with $\eta' \sqsubseteq \eta$.*

8 Related work

Linear typing systems Session types for the π-calculus originate from linear typing systems [17, 20], whose main aim is to guarantee that a channel is used exactly or at most once within a term. The objective of the session types in our paper is to check sequences of channel usages, rather than to verify that a variable or channel is used linearly (exactly once) within a program.

In the context of programming languages, the work [10] proposes a type system for checking protocols and resource usage in order to enforce linearity of variables in

the presence of aliasing; hence, their objectives are different. They implemented the typing system in Vault [7], a low level C-like language. The main issue that they had to address is that a shared component should not reference 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 fresh channels takes place when reducing connect, and therefore we do not need explicit constructs for this purpose. Finally, note that the system of [10] is not readily applicable in a concurrent setting, and hence in channel-based communication.

Programming languages and sessions In [27] the authors extend previous work [13], and define a concurrent functional language with session primitives. Their language supports the sending of channels and higher-order values, and incorporates branching and selection, along with recursive sessions. Moreover, it incorporates the multithreading primitive fork, whose operational semantics is similar to that of spawn, and also channel sharing. Finally, their system allows live channels as parameters to functions, and tracks aliasing of channels; as a result, their system is polymorphic.

In [25], 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; an object offers several methods via a branching construct, and a client selects the branch that corresponds to the desired method. In this context, sessions represent sequences of method invocations between two parties. Each branch that corresponds to a method invocation has two components, the arguments are encoded as being output and the return type as input. Labelled branches are also used to denote exceptions, when present, and their system incorporates recursive session types. However, run-time checks are considered in order to check protocol conformance; an attempt to provide static analysis is left as future work. Note that in their work there is no formalisation in terms of operational semantics and type system.

We developed our formalism building upon previous experience with $\mathcal{L}_{doos}$ [8], 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 [8], 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 we preferred to compromise such superficial type-checking — the essence of our system is in typing a session body against the session type.

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 [25]. 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 [8]. We discovered a flaw in the progress theorem in $\mathcal{L}_{doos}$ [8], and developed the new type system with hot sets in order to guard against the offending configurations.

Subject Reduction and Progress In all previously mentioned papers on session types, typability guarantees absence of run-time communication errors. However, not all of them have the subject reduction property: the problem emerges when sending a live channel to a thread which already uses this channel to communicate, as in our Example 4.2. This example can be translated into the calculi studied in [5, 12, 18, 27], and this issue has been discussed with some of their authors [1].

Although MOOSE has been inspired by the previously mentioned papers, we believe that MOOSE is the only calculus which guarantees absence of starvation on live channels. For example, we can encode the counterpart of Example 5.3 in the calculi in [5, 12, 18, 27]. More details on these two issues can be found in Appendix F.

Note that 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, by simply dropping the hot set. In this sense, our system is not only theoretically sound, but also modular.

9 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 its 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. In particular, the type inference on session environments (Proposition 6.1), and the progress property on live channels (Theorem 7.5) have never been proved in any work on session types including those in the π-calculus.

Advanced session types in the π-calculus and process calculi An issue that arises with the use of sessions is how to group and distinguish different behaviours within a program or protocol. In [18] and subsequently in [27] the authors utilise labelled *branching* and *selection*; the first enables a process to offer alternative session paths indexed by labels, and the second is used dually to choose a path by selecting one of the available labels. In [11, 15, 18, 26], branching and selection are considered as an effective way to simulate methods of objects. Several advancements have been made, ranging from simple session subtyping [11] to more complex bounded session polymorphism [15], which corresponds to parametric polymorphism within session types. Our conditional constructs are a simplification of branching and selection, therefore in essence the same behaviour realised by branching types can also be expressed using our types.

As another study on the enrichment of basic session types, in [5] the authors integrate the *correspondence assertions* of [14] 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.

Exceptions, timeout and implementation Another 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 [25]. 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 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 if we wish to preserve the progress property on live channels: to handle more complex synchronisation incurred by exception and timeout is an urgent further study in order to apply our framework to more practical communication protocols.

Finally, we are considering a prototype implementation using source to source translation from MOOSE to Java code. The most interesting parts are: firstly, the choice of a suitable runtime representation for both shared and linear channels; secondly, the ability to detect and control implicit multithreading; finally, the efficient implementation of higher-order sessions.

Acknowlegments We gratefully acknowledge (in alphabetic order) Eduardo Bonelli, Adriana Compagnoni, Kohei Honda, Simon Gay, Pablo Garralda, Elsa Gunter, Antonio Ravara and Vasco Vasconcelos, for discussions on subject reduction and progress for systems with sessions types which strongly influenced the present version of the section on related work. Vasco Vasconcelos gave also useful suggestions on a first draft of the present paper.

References

1. Personal communication by e-mails between the authors of [4, 5, 11, 13, 16, 18, 24, 25, 27]. Nov 2005–.
2. A. Ahern and N. Yoshida. Formalising java rmi with explicit code mobility. In *OOPSLA '05, the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 403–422. ACM Press, 2005.
3. G. Bierman, M. Parkinson, and A. Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical Report 563, University of Cambridge Computer Laboratory, April 2003.
4. E. Bonelli, A. Compagnoni, and E. Gunter. Typechecking safe process synchronization. In *FGUC 2004*, ENTCS, 2004.
5. E. Bonelli, A. Compagnoni, and E. Gunter. Correspondence assertions for process synchronization in concurrent communications. *JFP*, 15(2), 2005.
6. M. Carbone, K. Honda, and N. Yoshida. A theoretical basis of communication-centered concurrent programming. Web Services Choreography Working Group mailing list, to appear as a WSCDL working report.

7. R. DeLine and M. Fahndrich. Enforcing high-level protocols in low-level software. In *PLDI '01*, pages 59–69. ACM Press, 2001.
8. M. Dezani, N. Yoshida, A. Ahern, and S. Drossopoulou. A Distributed Object Oriented Language with Session Types. In *Proc. of TGC'05*, LNCS, 2005.
9. S. Drossopoulou. Advanced issues in object oriented languages course notes. http://www.doc.ic.ac.uk/˜scd/Teaching/AdvOO.html.
10. M. Fahndrich and R. DeLine. Adoption and focus: practical linear types for imperative programming. In *PLDI '02*, pages 13–24. ACM Press, 2002.
11. S. Gay and M. Hole. Types and subtypes for client-server interactions. In *ESOP'99*, volume 1576 of *LNCS*, pages 74–90. Springer-Verlag, 1999.
12. S. Gay and M. Hole. Subtyping for session types in the pi-calculus. *Acta Informatica*, 42(2/3):191–225, 2005.
13. S. Gay, V. T. Vasconcelos, and A. Ravara. Session types for inter-process communication. TR 2003–133, Department of Computing, University of Glasgow, Mar. 2003.
14. A. D. Gordon and A. Jeffrey. Typing correspondence assertions for communication protocols. In *mfps01*, volume 45 of *ENTCS*, pages 379–409. Elsevier, May 2001.
15. M. Hole and S. J. Gay. Bounded polymorphism in session types. Technical Report TR-2003-132, Department of Computing Science, University of Glasgow, March 2003.
16. K. Honda. Types for dyadic interaction. In *CONCUR'93*, volume 715 of *LNCS*, pages 509–523. Springer-Verlag, 1993.
17. K. Honda. Composing processes. In *POPL'96*, pages 344–357. ACM Press, 1996.
18. K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer-Verlag, 1998.
19. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.
20. N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the Pi-Calculus. *ACM Transactions on Programming Languages and Systems*, 21(5):914–947, Sept. 1999. Summary in POPL 1996.
21. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100(1), 1992.
22. D. Mostrous. Moose: a Minimal Object Oriented Language with Session Types. Master's thesis, Imperial College London, MSc Thesis, Sep, 2005.
23. B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
24. K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *PARLE'94*, volume 817 of *LNCS*, pages 398–413. Springer-Verlag, 1994.
25. A. Vallecillo, V. T. Vasconcelos, and A. Ravara. Typing the behavior of objects and components using session types. In *Foclasa 2002*, volume 68(3) of *ENTCS*. Elsevier, 2002.
26. V. Vasconcelos. Typed concurrent objects. In *Proc. ECOOP'94*, 1994.
27. V. T. Vasconcelos, A. Ravara, and S. Gay. Session types for functional multithreading. In *CONCUR'04*, volume 3170 of *LNCS*, pages 497–511. Springer-Verlag, 2004.
28. Web Services Choreography Working Group. Web Services Choreography Description Language. http://www.w3.org/2002/ws/chor/.

A Appendix: Lookup Functions

Fig. 11 defines auxiliary functions used in the operational semantics and typing rules.

Field lookup

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

Method lookup

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

Method type lookup

$$\frac{\mathsf{class}\ C\ \mathsf{extends}\ D\ \{\tilde{\mathsf{f}}\,\tilde{\mathsf{t}}\ \tilde{M}\} \in \mathtt{CT} \quad \mathsf{t}\,\mathsf{m}\ (\tilde{\tau}\tilde{\mathsf{x}})\ \{\mathsf{e}\} \in \tilde{M}}{\mathsf{mtype}(\mathsf{m}, C) = \tilde{\tau} \to \mathsf{t}} \qquad \frac{\mathsf{class}\ C\ \mathsf{extends}\ D\ \{\tilde{\mathsf{f}}\,\tilde{\mathsf{t}}\ \tilde{M}\} \in \mathtt{CT} \quad \mathsf{m} \notin \tilde{M}}{\mathsf{mtype}(\mathsf{m}, C) = \mathsf{mtype}(\mathsf{m}, D)}$$

Method body lookup

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

τ is either t or η.

Fig. 11. Lookup Functions

B Appendix: Typing System

We type expressions and threads with respect to a fixed class table $\mathtt{CT}$, and this is reflected in the rules of Fig. 13 which define well-formed types.

The subtyping $<:$ is formalised in Fig. 14.

Fig. 13 gives the formation rules for standard and session environments, where the domain of an environment is defined as usual.

The typing rules for expressions are given in Fig. 16, Fig. 17, Fig. 18 and Fig. 20. Notice that in rules **ReceiveIF** and **SendIF** either η_1 and η_2 are *both complete* or *both incomplete* parts of live channel types in order to agree with the syntax (see Fig. 4).

Fig. 21 defines well-formed class tables.

C Appendix: Type Inference Rules

We denote $\mathsf{fv}(P)$ and $\mathsf{fch}(P)$ the sets of free variables and channels in P, respectively.

Fig. 22 gives some auxiliary operators on sessions environments and hot sets. The inference rules occupy Fig. 23, Fig. 24, Fig. 25 and Fig. 26.

$$
\begin{array}{ll}
\overline{\mathsf{begin}.\rho} & = \mathsf{begin}.\overline{\rho} \\
\overline{\pi.\mathsf{end}} & = \overline{\pi}.\mathsf{end} \\
\overline{\dagger\langle \rho_1, \rho_2 \rangle} & = \overline{\dagger}\langle \overline{\rho_1}, \overline{\rho_2} \rangle \\
\overline{?} & = ! \\
\overline{!} & = ? \\
\overline{\dagger \mathsf{t}} & = \overline{\dagger}\mathsf{t} \\
\overline{\dagger(\rho)} & = \overline{\dagger}(\rho) \\
\overline{\dagger\langle \pi_1, \pi_2 \rangle} & = \overline{\dagger}\langle \overline{\pi_1}, \overline{\pi_2} \rangle \\
\overline{\dagger\langle \pi \rangle^*} & = \overline{\dagger}\langle \overline{\pi} \rangle^* \\
\overline{\pi_1.\pi_2} & = \overline{\pi_1}.\overline{\pi_2}
\end{array}
$$

Fig. 12. Duality of Session Types.

$$
\begin{array}{llll}
\textbf{Class} & \textbf{LSession} & \textbf{Wf-Session} & \textbf{Tuple} \qquad\qquad \textbf{Bool} \\
\dfrac{C \in \mathsf{dom}(\mathtt{CT})}{\vdash C : \mathtt{tp}} & \dfrac{}{\vdash \eta : \mathtt{tp}} & \dfrac{}{\vdash \mathsf{s} : \mathtt{tp}} & \dfrac{}{\vdash (\mathsf{s}, \overline{\mathsf{s}}) : \mathtt{tp}} \qquad \dfrac{}{\vdash \mathsf{bool} : \mathtt{tp}}
\end{array}
$$

Fig. 13. Well-formed Types

$$
\frac{}{(\mathsf{s}, \overline{\mathsf{s}}) <: \mathsf{s}} \qquad \frac{}{(\mathsf{s}, \overline{\mathsf{s}}) <: \overline{\mathsf{s}}} \qquad \frac{C \in dom(\mathtt{CT})}{C <: C} \qquad \frac{C <: D \qquad D <: E}{C <: E} \qquad \frac{\mathsf{class}\ C\ \mathsf{extends}\ D\ \{\tilde{\mathsf{f}}\ \tilde{\mathsf{t}}\ \tilde{M}\} \in \mathtt{CT}}{C <: D}
$$

Fig. 14. Subtyping

$$
\begin{array}{llll}
\textbf{Emp} & \textbf{EVar} & \textbf{EOid} & \textbf{Ethis} \\
\dfrac{}{\emptyset \vdash \mathsf{ok}} & \dfrac{\vdash \mathsf{t} : \mathtt{tp} \quad \mathsf{x} \notin \mathsf{dom}(\Gamma)}{\Gamma, \mathsf{x} : \mathsf{t} \vdash \mathsf{ok}} & \dfrac{\vdash C : \mathtt{tp} \quad \mathsf{o} \notin \mathsf{dom}(\Gamma)}{\Gamma, \mathsf{o} : C \vdash \mathsf{ok}} & \dfrac{\vdash C : \mathtt{tp} \quad \mathsf{this} \notin \mathsf{dom}(\Gamma)}{\Gamma, \mathsf{this} : C \vdash \mathsf{ok}}
\end{array}
$$

$$
\begin{array}{lll}
\textbf{SEmp} & \textbf{SERC} & \textbf{SECC} \\
\dfrac{}{\emptyset \vdash \mathsf{ok}} & \dfrac{\mathsf{u} \notin \mathsf{dom}(\Sigma)}{\Sigma, \mathsf{u} : \eta \vdash \mathsf{ok}} & \dfrac{\mathsf{u} \notin \mathsf{dom}(\Sigma)}{\Gamma, \mathsf{u} : \updownarrow \vdash \mathsf{ok}}
\end{array}
$$

Fig. 15. Well-formed environments

$$
\begin{array}{lllll}
\textbf{Null} & \textbf{Oid} & \textbf{True} & \textbf{False} & \textbf{Chan} \\
\dfrac{\Gamma \vdash \mathsf{ok} \quad \vdash \mathsf{t} : \mathtt{tp}}{\Gamma; \emptyset; \emptyset \vdash \mathsf{null} : \mathsf{t}} & \dfrac{\Gamma, \mathsf{o} : C \vdash \mathsf{ok}}{\Gamma, \mathsf{o} : C; \emptyset; \emptyset \vdash \mathsf{o} : C} & \dfrac{\Gamma \vdash \mathsf{ok}}{\Gamma; \emptyset; \emptyset \vdash \mathsf{true} : \mathsf{bool}} & \dfrac{\Gamma \vdash \mathsf{ok}}{\Gamma; \emptyset; \emptyset \vdash \mathsf{false} : \mathsf{bool}} & \dfrac{\Gamma \vdash \mathsf{ok}}{\Gamma; \emptyset; \emptyset \vdash \mathsf{c} : \mathsf{s}}
\end{array}
$$

Fig. 16. Typing Rules for Values

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

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

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

Seq
$$\frac{\Gamma ; \Sigma ; \mathcal{S} \vdash e : t \qquad \Gamma ; \Sigma' ; \mathcal{S} \vdash e' : t'}{\Gamma ; \Sigma . \Sigma' ; \mathcal{S} \vdash e ; e' : t'}$$

NewC
$$\frac{\Gamma \vdash \mathsf{ok} \quad \vdash C : \mathsf{tp}}{\Gamma ; \emptyset ; \emptyset \vdash \mathsf{new}\ C : C}$$

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

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

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

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

Meth
$$\frac{\Gamma ; \Sigma_0 ; \mathcal{S} \vdash e : C \qquad \Gamma ; \Sigma_i ; \mathcal{S} \vdash e_i : t_i \quad i \in \{1 \ldots n\}}{\Gamma ; \Sigma_0 . \Sigma_1 \ldots \Sigma_n ; \mathcal{S} \vdash e.m\,(e_1 \ldots e_n) : t} \quad \mathsf{mtype}(m, C) = t_1 \ldots t_n \to t$$

MethLin
$$\frac{\Gamma ; \Sigma_0 ; \{u\} \vdash e : C \qquad \Gamma ; \Sigma_i ; \{u\} \vdash e_i : t_i \quad i \in \{1 \ldots n\}}{\Gamma ; \Sigma_0 . \Sigma_1 \ldots \Sigma_n . \{u : \eta\} ; \{u\} \vdash e.m\,(e_1 \ldots e_n, u) : t} \quad \mathsf{mtype}(m, C) = t_1 \ldots t_n, \eta \to t$$

Fig. 17. Typing Rules for Standard Expressions

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

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

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

ReceiveS
$$\frac{\Gamma \setminus x\ ;\ \Sigma, x : \rho ; \{x\} \vdash e : t \quad \mathit{closed}(\Sigma)}{\Gamma ; \{u : ?(\rho)\} . \Sigma ; \{u\} \vdash u.\mathsf{receiveS}\,(x)\{e\} : \mathit{Object}}$$

SendS
$$\frac{\Gamma \vdash \mathsf{ok} \quad \vdash \rho : \mathsf{tp}}{\Gamma ; \{u' : \rho, u : !(\rho)\} ; \{u\} \vdash u.\mathsf{sendS}\,(u')\ :\ \mathit{Object}}$$

ReceiveIf
$$\frac{\Gamma ; \Sigma, u : \eta_i ; \{u\} \vdash e_i : t \quad i \in \{1,2\}}{\Gamma ; \Sigma, u : ?\langle \eta_1, \eta_2 \rangle ; \{u\} \vdash u.\mathsf{receiveIf}\,\{e_1\}\{e_2\} : t}$$

SendIf
$$\frac{\Gamma ; \Sigma_1 ; \{u\} \vdash e : \mathsf{bool} \qquad \Gamma ; \Sigma_2, u : \eta_i ; \{u\} \vdash e_i : t \quad i \in \{1,2\}}{\Gamma ; \Sigma_1 . \{\Sigma_2, u :! \langle \eta_1, \eta_2 \rangle\} ; \{u\} \vdash u.\mathsf{sendIf}\,(e)\{e_1\}\{e_2\} : t}$$

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

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

Fig. 18. Typing Rules for Communication Expressions

$$\textbf{Start}\ \frac{\Gamma;\Sigma;\mathcal{S}\vdash \mathtt{e}:\mathtt{t}}{\Gamma;\Sigma\vdash \mathtt{e}:\mathsf{thread}} \qquad \textbf{Par}\ \frac{\Gamma;\Sigma\vdash P:\mathsf{thread}\quad \Gamma;\Sigma'\vdash P':\mathsf{thread}}{\Gamma;\Sigma\|\Sigma'\vdash P\,|\,P':\mathsf{thread}}$$

Fig. 19. Typing Rules for Threads

$$\textbf{WeakS}\ \frac{\Gamma;\ \Sigma;\emptyset\vdash \mathtt{e}:\mathtt{t}}{\Gamma;\ \Sigma,\mathtt{u}:\varepsilon;\emptyset\vdash \mathtt{e}:\mathtt{t}} \qquad \textbf{Weak}\ \frac{\Gamma;\Sigma;\emptyset\vdash \mathtt{e}:\mathtt{t}\quad \mathtt{u}\in\mathsf{dom}(\Sigma)}{\Gamma;\Sigma;\{\mathtt{u}\}\vdash \mathtt{e}:\mathtt{t}}$$

$$\textbf{Sub}\ \frac{\Gamma;\Sigma;\mathcal{S}\vdash \mathtt{e}:\mathtt{t}}{\Gamma;\Sigma;\mathcal{S}\vdash \mathtt{e}:\mathtt{t}'}\ \mathtt{t}<:\mathtt{t}' \qquad \textbf{Close}\ \frac{\Gamma;\Sigma,\mathtt{u}:\pi;\mathcal{S}\vdash \mathtt{e}:\mathtt{t}}{\Gamma;\Sigma,\mathtt{u}:\pi.\mathsf{end};\mathcal{S}\vdash \mathtt{e}:\mathtt{t}}$$

Fig. 20. Non-structural Typing Rules for Expressions

$$\textbf{M-ok}\ \frac{\mathtt{this}:C,\tilde{\mathtt{x}}:\tilde{\mathtt{t}}\ ;\ \emptyset;\ \emptyset\vdash \mathtt{e}:\mathtt{t}}{\mathtt{t\ m}\ (\tilde{\mathtt{t}}\ \tilde{\mathtt{x}})\ \ \{\mathtt{e}\}:\mathtt{ok\ in}\ C} \qquad \textbf{MLin-ok}\ \frac{\mathtt{this}:C,\tilde{\mathtt{x}}:\tilde{\mathtt{t}}\ ;\ \mathtt{x}:\eta;\ \{\mathtt{x}\}\vdash \mathtt{e}:\mathtt{t}}{\mathtt{t\ m}\ (\tilde{\mathtt{t}}\ \tilde{\mathtt{x}},\eta\mathtt{x})\ \ \{\mathtt{e}\}:\mathtt{ok\ in}\ C}$$

$$\textbf{C-ok}\ \frac{\tilde{M}:\mathtt{ok\ in}\ C}{\mathsf{class}\ C\ \mathsf{extends}\ D\ \{\tilde{\mathtt{f}}\,\tilde{\mathtt{t}}\ \tilde{M}\}:\mathtt{ok}}\ \mathsf{class}\ C\ \mathsf{extends}\ D\ \{\tilde{\mathtt{f}}\,\tilde{\mathtt{t}}\ \tilde{M}\}\in\mathtt{CT}$$

$$\textbf{CT-ok}\ \frac{\mathsf{class}\ C\ \mathsf{extends}\ D\ \{\tilde{\mathtt{f}}\,\tilde{\mathtt{t}}\ \tilde{M}\}:\mathtt{ok}\qquad \mathtt{CT}:\mathtt{ok}}{\mathtt{CT},\mathsf{class}\ C\ \mathsf{extends}\ D\ \{\tilde{\mathtt{f}}\,\tilde{\mathtt{t}}\ \tilde{M}\}:\mathtt{ok}}$$

Fig. 21. Well-formed Class Tables

Closure of linear channel types and of environment sessions

$$\eta\downarrow=\begin{cases}\eta, & \text{if } \eta=\rho,\\ \pi.\dagger\langle\eta_1\downarrow,\eta_2\downarrow\rangle, & \text{if } \eta=\pi.\dagger\langle\eta_1,\eta_2\rangle,\\ \eta.\mathsf{end} & \text{otherwise.}\end{cases}\qquad \Sigma\downarrow=\{\eta:\Sigma(\mathtt{u})\downarrow\ \ |\mathtt{u}\in\mathsf{dom}(\Sigma)\}$$

Union of hot sets

$$\mathcal{S}\uplus\mathcal{S}'=\begin{cases}\mathcal{S} & \text{if either } \mathcal{S}=\mathcal{S}' \text{ or } \mathcal{S}'=\emptyset,\\ \mathcal{S}' & \text{if } \mathcal{S}=\emptyset,\\ \text{undefi ned} & \text{otherwise.}\end{cases}$$

Extension of environment sessions

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

Fig. 22. Auxiliary Operators for Inference

NullI $\dfrac{\Gamma \vdash \mathsf{ok} \quad \vdash \mathtt{t} : \mathtt{tp}}{\Gamma \vdash \mathsf{null} : \mathtt{t} \parallel \emptyset;\emptyset}$ **OidI** $\dfrac{\Gamma, \mathtt{o} : C \vdash \mathsf{ok}}{\Gamma, \mathtt{o} : C \vdash \mathtt{o} : C \parallel \emptyset;\emptyset}$ **TrueI** $\dfrac{\Gamma \vdash \mathsf{ok}}{\Gamma \vdash \mathsf{true} : \mathsf{bool} \parallel \emptyset;\emptyset}$ **FalseI** $\dfrac{\Gamma \vdash \mathsf{ok}}{\Gamma \vdash \mathsf{false} : \mathsf{bool} \parallel \emptyset;\emptyset}$ **ChanI** $\dfrac{\Gamma \vdash \mathsf{ok}}{\Gamma \vdash \mathtt{c} : \mathtt{s} \parallel \emptyset;\emptyset}$

Fig. 23. Inference Rules for Values

VarI $\dfrac{\Gamma, \mathtt{x} : \mathtt{t} \vdash \mathsf{ok}}{\Gamma, \mathtt{x} : \mathtt{t} \vdash \mathtt{x} : \mathtt{t} \parallel \emptyset;\emptyset}$ **ThisI** $\dfrac{\Gamma, \mathsf{this} : C \vdash \mathsf{ok}}{\Gamma, \mathsf{this} : C \vdash \mathsf{this} : C \parallel \emptyset;\emptyset}$ **FldI** $\dfrac{\Gamma \vdash \mathtt{e} : C \parallel \Sigma;\mathcal{S}}{\Gamma \vdash \mathtt{e.f} : \mathtt{t} \parallel \Sigma;\mathcal{S}} \ \mathtt{f}\,\mathtt{t} \in \mathsf{fields}(C)$

SeqI $\dfrac{\Gamma \vdash \mathtt{e} : \mathtt{t} \parallel \Sigma;\mathcal{S} \qquad \Gamma \vdash \mathtt{e}' : \mathtt{t}' \parallel \Sigma';\mathcal{S}'}{\Gamma \vdash \mathtt{e};\mathtt{e}' : \mathtt{t}' \parallel \Sigma.\Sigma';\mathcal{S} \uplus \mathcal{S}'}$ **NewCI** $\dfrac{\Gamma \vdash \mathsf{ok} \quad \vdash C : \mathtt{tp}}{\Gamma \vdash \ \mathsf{new}\ C : C \parallel \emptyset;\emptyset}$

FldAssI $\dfrac{\Gamma \vdash \mathtt{e} : C \parallel \Sigma;\mathcal{S} \qquad \Gamma \vdash \mathtt{e}' : \mathtt{t} \parallel \Sigma';\mathcal{S}'}{\Gamma \vdash \mathtt{e.f} := \mathtt{e}' : \mathtt{t} \parallel \Sigma.\Sigma';\mathcal{S} \uplus \mathcal{S}'} \ \mathtt{f}\,\mathtt{t} \in \mathsf{fields}(C)$ **NewSI** $\dfrac{\Gamma \vdash \mathsf{ok}}{\Gamma \vdash \ \mathsf{new}\ (\mathtt{s}, \overline{\mathtt{s}}) : (\mathtt{s}, \overline{\mathtt{s}}) \parallel \emptyset;\emptyset}$

SpawnI $\dfrac{\Gamma \vdash \mathtt{e} : \mathtt{t} \parallel \Sigma;\mathcal{S}}{\Gamma \vdash \mathsf{spawn}\{\ \mathtt{e}\ \} : \mathit{Object} \parallel \Sigma\downarrow;\mathcal{S}}$ **NullPEI** $\dfrac{\Gamma \vdash \mathsf{ok} \quad \vdash \mathtt{t} : \mathtt{tp}}{\Gamma \vdash \mathsf{NullExc} : \mathtt{t} \parallel \emptyset;\emptyset}$

MethI $\dfrac{\Gamma \vdash \mathtt{e} : C \parallel \Sigma_0;\mathcal{S}_0 \qquad \Gamma \vdash \mathtt{e}_i : \mathtt{t}_i \parallel \Sigma_i;\mathcal{S}_i \quad i \in \{1 \ldots n\}}{\Gamma \vdash \mathtt{e.m}(\mathtt{e}_1 \ldots \mathtt{e}_n) : \mathtt{t} \parallel \Sigma_0.\Sigma_1 \ldots \Sigma_n;\mathcal{S}_0 \uplus \mathcal{S}_1 \uplus \ldots \uplus \mathcal{S}_n} \ \mathsf{mtype}(\mathtt{m}, C) = \mathtt{t}_1 \ldots \mathtt{t}_n \to \mathtt{t}$

MethLinI $\dfrac{\Gamma \vdash \mathtt{e} : C \parallel \Sigma_0;\mathcal{S}_0 \qquad \Gamma \vdash \mathtt{e}_i : \mathtt{t}_i \parallel \Sigma_i;\mathcal{S}_i \quad \mathcal{S}_i \subseteq \{\mathtt{u}\} \quad i \in \{1 \ldots n\}}{\Gamma \vdash \mathtt{e.m}(\mathtt{e}_1 \ldots \mathtt{e}_n, \mathtt{u}) : \mathtt{t} \parallel \Sigma_0.\Sigma_1 \ldots \Sigma_n.\{\mathtt{u}\};\{\mathtt{u}\}} \quad \mathsf{mtype}(\mathtt{m}, C) = \mathtt{t}_1 \ldots \mathtt{t}_n, \eta \to \mathtt{t}$

Fig. 24. Inference Rules for Standard Expressions

D Appendix: Proof of Subject Reduction

We show that structural equivalence preserves types using a lemma on the parallel composition of session environments.

Lemma D.1 (Commutativity of Parallel Composition of Session Environments). $\Sigma_1 \| (\Sigma_2 \| \Sigma_3)$ *defined* $\Longrightarrow (\Sigma_1 \| \Sigma_2) \| \Sigma_3$ *defined and* $\Sigma_1 \| (\Sigma_2 \| \Sigma_3) = (\Sigma_1 \| \Sigma_2) \| \Sigma_3$.

Proof. By induction on the size of Σ_1.

Lemma D.2 (Preservation of Typing under Structural Equivalence). *If* $\Gamma;\Sigma;\mathcal{S} \vdash P : \mathsf{thread}$ *and* $P \equiv P'$ *then* $\Gamma;\Sigma;\mathcal{S} \vdash P' : \mathsf{thread}$.

Proof. By induction on derivations.

The next goal is to prove that term substitution preserves types (Lemma D.8). We first show three auxiliary lemmas.

ConnI

$$\frac{\Gamma \vdash e : t \parallel \Sigma; \mathcal{S} \quad \Sigma(\!(u)\!) = \eta \quad s = \mathsf{begin}.\sigma(\eta\downarrow) \quad u \notin \mathsf{dom}(\Gamma) \quad \Gamma' = \Gamma \text{ if } u \text{ is a name else } \Gamma, u : s}{\Gamma' \vdash \mathsf{connect}\ u\ s\ \{e\} : \sigma(t) \parallel \sigma(\Sigma) \setminus u; \emptyset}$$

SendI

$$\frac{\Gamma \vdash e : t \parallel \Sigma; \mathcal{S} \quad \mathcal{S} \subseteq \{u\}}{\Gamma \vdash u.\mathsf{send}(\ e\ ) : \mathit{Object} \parallel \Sigma.\{u : !t\}; \{u\}}$$

ReceiveI

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

ReceiveSI

$$\frac{\Gamma \vdash e : t \parallel \Sigma; \mathcal{S} \quad x \notin \Gamma \quad \mathcal{S} \subseteq \{x\} \quad \Sigma(\!(x)\!) = \eta}{\Gamma \vdash u.\mathsf{receiveS}(x)\{e\} : \mathit{Object} \parallel \{u : ?(\eta\downarrow)\}.\Sigma\downarrow; \{u\}}$$

SendSI

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

ReceiveIfI

$$\frac{\Gamma \vdash e_i : t \parallel \Sigma_i; \mathcal{S}_i \quad \Sigma_i(\!(u)\!) = \eta_i \quad \forall u' \neq u.\Sigma_1(u') = \Sigma_2(u') \quad \mathcal{S}_i \subseteq \{u\} \quad i \in \{1,2\}}{\Gamma \vdash u.\mathsf{receiveIf}\{e_1\}\{e_2\} : t \parallel \Sigma_1 \setminus u, u : ?\langle \eta_1, \eta_2 \rangle; \{u\}}$$

SendIfI

$$\frac{\Gamma \vdash e : \mathsf{bool} \parallel \Sigma; {}_0\mathcal{S}_0 \quad \Gamma \vdash e_i : t \parallel \Sigma_i; \mathcal{S}_i \quad \Sigma_i(\!(u)\!) = \eta_i \quad \forall u' \neq u.\Sigma_1(u') = \Sigma_2(u') \quad \mathcal{S}_j \subseteq \{u\} \quad i \in \{1,2\} \quad j \in \{0,1,2\}}{\Gamma \vdash u.\mathsf{sendIf}(e)\{e_1\}\{e_2\} : t \parallel \Sigma_0.\{\Sigma_1 \setminus u, u : !\langle \eta_1, \eta_2 \rangle\}; \{u\}}$$

ReceiveWhileI

$$\frac{\Gamma \vdash e : t \parallel \Sigma; \mathcal{S} \quad \Sigma(\!(u)\!) = \pi \quad \Sigma \subseteq \{u : \pi\} \quad \mathcal{S} \subseteq \{u\}}{\Gamma \vdash u.\mathsf{receiveWhile}\{e\} : t \parallel \{u : ?\langle \pi \rangle^*\}; \{u\}}$$

SendWhileI

$$\frac{\Gamma \vdash e : \mathsf{bool} \parallel \Sigma; {}_0\mathcal{S}_0 \quad \Gamma \vdash e_1 : t \parallel \Sigma_1; \mathcal{S}_1 \quad \Sigma_i(\!(u)\!) = \pi_i \quad \Sigma_i \subseteq \{u : \pi_i\} \quad \mathcal{S}_i \subseteq \{u\} \quad i \in \{0,1\}}{\Gamma \vdash u.\mathsf{sendWhile}(e)\{e'\} : t \parallel \{u : \pi_0.!\langle \pi_1.\pi_0 \rangle^*\}; \{u\}}$$

Fig. 25. Inference Rules for Communication Expressions

Start

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

Par

$$\frac{\Gamma \vdash P : \mathsf{thread} \parallel \Sigma \quad \Gamma \vdash P : \mathsf{thread} \parallel {}'\Sigma'}{\Gamma \vdash P \mid P' : \mathsf{thread} \parallel \Sigma \| \Sigma'}$$

Fig. 26. Inference Rules for Threads

Lemma D.3 (Composition of Session Environments & Substitution). *If* $\tilde{u} \notin \mathsf{dom}(\Sigma_1.\Sigma_2)$ *then:*

1. $\Sigma_1.\Sigma_2$ *defined* $\Longrightarrow (\Sigma_1.\Sigma_2)[\tilde{u}'/\tilde{u}]$ *defined.*
2. $(\Sigma_1.\Sigma_2)[\tilde{u}'/\tilde{u}] = \Sigma_1[\tilde{u}'/\tilde{u}].\Sigma_2[\tilde{u}'/\tilde{u}]$

Proof. (1) and (2) follow both by induction on the size of Σ_1.

Definition D.4 (Channel Occurrences). *A free occurrence of the variable* x *in the expression* e *is* a channel occurrence *if either* x *occurs as subject of session expressions or* x *is the sent channel in sent channel expressions.*

Lemma D.5 (Channel Occurrence Property). *If* $\Gamma;\Sigma;\mathcal{S} \vdash e : t$, $x \in \mathsf{fv}(e)$, *and* $x \notin \mathsf{dom}(\Gamma)$ *then all occurrences of* x *in* e *are channel occurrences.*

Proof. By analysis of the typing rules.

Lemma D.6 (Narrowing of Standard Environments). *If* $\Gamma, x : t\,;\Sigma;\mathcal{S} \vdash e : t$ *and* x *only has channel occurrences in* e, *then* $\Gamma;\Sigma;\mathcal{S} \vdash e : t$.

Proof. By induction on derivations.

Lemma D.7 (Weakening of Standard Environments). *If* $\Gamma;\Sigma;\mathcal{S} \vdash e : t$ *then* $\Gamma, x : t\,;\Sigma;\mathcal{S} \vdash e : t$.

Proof. By induction on derivations.

Lemma D.8 (Preservation of Typing under Substitution).

1. *If* $\Gamma, \tilde{x}\,\tilde{t}\,;\Sigma;\mathcal{S} \vdash e : t$ *and* $\tilde{x}{:}\tilde{t} \in \Gamma$ *and* $\tilde{x} \cap \mathsf{dom}(\Sigma) = \emptyset$ *and* $\tilde{x} \cap \mathsf{fv}(e) = \emptyset$ *then* $\Gamma;\Sigma[\tilde{x}'/\tilde{x}];\mathcal{S} \vdash e[\tilde{x}'/\tilde{x}] : t$;
2. *If* $\Gamma;\Sigma;\mathcal{S} \vdash e : t$ *and* c *is fresh then* $\Gamma;\Sigma[c/u];\mathcal{S}[c/u] \vdash e[c/u] : t$.
3. *If* $\Gamma, \mathsf{this} : C;\Sigma;\mathcal{S} \vdash e : t$ *and* $\Gamma;\emptyset;\emptyset \vdash o : C$ *then* $\Gamma;\Sigma;\mathcal{S} \vdash e[o/\mathsf{this}] : t$.

Proof. (1), (2) and (3) are proved by induction on derivations.

As usual generation lemmas are handy for subject reduction proofs.

Lemma D.9 (Generation Lemma for Standard Expressions).

1. $\Gamma;\Sigma;\mathcal{S} \vdash x : t$ *implies* $\Sigma = \mathcal{S} = \emptyset$ *and* $x : t' \in \Gamma$ *for some* $t' <: t$.
2. $\Gamma;\Sigma;\mathcal{S} \vdash c : t$ *implies* $\Sigma = \mathcal{S} = \emptyset$ *and* $t = s$.
3. $\Gamma;\Sigma;\mathcal{S} \vdash \mathsf{null} : t$ *implies* $\Sigma = \mathcal{S} = \emptyset$.
4. $\Gamma;\Sigma;\mathcal{S} \vdash v : t$ *with* $v \in \{\mathsf{true}, \mathsf{false}\}$ *implies* $\Sigma = \mathcal{S} = \emptyset$ *and* $t = \mathsf{bool}$.
5. $\Gamma;\Sigma;\mathcal{S} \vdash o : t$ *implies* $\Sigma = \mathcal{S} = \emptyset$ *and* $t = C$.
6. $\Gamma;\Sigma;\mathcal{S} \vdash \mathsf{NullExc} : t$ *implies* $\Sigma = \mathcal{S} = \emptyset$.
7. $\Gamma;\Sigma;\mathcal{S} \vdash \mathsf{this} : t$ *implies* $\Sigma = \mathcal{S} = \emptyset$ *and* $t = C$ *and* $\mathsf{this} : C' \in \Gamma$ *for some* $C' <: C$.
8. $\Gamma;\Sigma;\mathcal{S} \vdash e_1;e_2 : t$ *implies* $\Sigma = \Sigma_1.\Sigma_2$ *and* $\mathcal{S} = \mathcal{S}_1 = \mathcal{S}_2$ *and* $t = t_2$ *and* $\Gamma;\Sigma_i;\mathcal{S}_i \vdash e_i : t_i$ *(*$i \in \{1,2\}$*).*
9. $\Gamma;\Sigma;\mathcal{S} \vdash e.f := e' : t$ *implies* $\Sigma = \Sigma_1.\Sigma_2$ *and* $\mathcal{S} = \mathcal{S}_1 = \mathcal{S}_2$ *and* $\Gamma;\Sigma_1;\mathcal{S}_1 \vdash e : C$ *and* $\Gamma;\Sigma_2;\mathcal{S}_2 \vdash e' : t$ *and* $f\,t \in \mathsf{fields}(C)$.
10. $\Gamma;\Sigma;\mathcal{S} \vdash e.f : t$ *implies* $\Gamma;\Sigma;\mathcal{S} \vdash e : C$ *and* $f\,t \in \mathsf{fields}(C)$.
11. $\Gamma;\Sigma;\mathcal{S} \vdash e.m(e_1 \ldots e_n) : t$ *implies* $\Gamma;\Sigma_0 \vdash e : C$ *and* $\mathcal{S} = \mathcal{S}_i$ *and* $\Gamma;\Sigma_i;\mathcal{S}_i \vdash e_i : t_i$ *(*$1 \le i \le n$*) and*
    (a) *either* $\Sigma = \Sigma_0.\Sigma_1 \ldots \Sigma_n$ *and* $\mathsf{mtype}(m, C) = t_1 \ldots t_n \to t$;
    (b) *or* $\Sigma = \Sigma_0.\Sigma_1 \ldots \Sigma_{n-1}.\{u : \eta\}$ *and* $\mathcal{S} = \{u\}$ *and* $\mathsf{mtype}(m, C) = t_1 \ldots t_{n-1}, \eta \to t$.
12. $\Gamma;\Sigma;\mathcal{S} \vdash \mathsf{new}\ C : t$ *implies* $\Sigma = \mathcal{S} = \emptyset$ *and* $t = C$.
13. $\Gamma;\Sigma;\mathcal{S} \vdash \mathsf{new}\ (s, \overline{s}) : t$ *implies* $\Sigma = \mathcal{S} = \emptyset$ *and* $t = (s, \overline{s})$.

14. $\Gamma;\Sigma;\mathcal{S} \vdash \mathsf{spawn}\{\ \mathsf{e}\ \}:\mathsf{t}$ *implies closed*(Σ) *and* $\mathsf{t} =$ *Object and* $\Gamma;\Sigma;\mathcal{S} \vdash \mathsf{e}:\mathsf{t}$.

Lemma D.10 (Generation Lemma for Communication Expressions).

1. $\Gamma;\Sigma;\mathcal{S} \vdash \mathsf{connect\ u\ s}\ \{\mathsf{e}\}:\mathsf{t}$ *implies* $\mathsf{s} = \mathsf{begin}.\rho$ *and* $\Gamma;\emptyset;\emptyset \vdash \mathsf{u}:\mathsf{begin}.\rho$ *and and* $\Gamma \setminus \mathsf{u};\Sigma,\mathsf{u}:\rho;\{\mathsf{u}\} \vdash \mathsf{e}:\mathsf{t}$.
2. $\Gamma;\Sigma;\mathcal{S} \vdash \mathsf{u}.\mathsf{receive}:\mathsf{t}$ *implies* $\Sigma = \mathsf{u}:?\mathsf{t}$ *and* $\mathcal{S} = \{\mathsf{u}\}$.
3. $\Gamma;\Sigma;\mathcal{S} \vdash \mathsf{u}.\mathsf{send}(\ \mathsf{e}\ ):\mathsf{t}$ *implies* $\Sigma = \Sigma'.\{\mathsf{u}:!\mathsf{t}\}$ *and* $\mathcal{S} = \{\mathsf{u}\}$ *and* $\mathsf{t} =$ *Object and* $\Gamma;\Sigma';\{\mathsf{u}\} \vdash \mathsf{e}:\mathsf{t}$.
4. $\Gamma;\ \Sigma;\mathcal{S} \vdash \mathsf{u}.\mathsf{receiveS}(\mathsf{x})\{\mathsf{e}\}:\mathsf{t}$ *implies* $\Sigma = \{\mathsf{u}:?(\rho)\}.\Sigma'$ *and* $\mathcal{S} = \{\mathsf{u}\}$ *and* $\mathsf{t} =$ *Object and closed*(Σ) *and* $\Gamma \setminus \mathsf{x}\ ;\ \Sigma',\mathsf{x}:\rho;\{\mathsf{x}\} \vdash \mathsf{e}:\mathsf{t}'$.
5. $\Gamma;\ \Sigma;\mathcal{S} \vdash \mathsf{u}.\mathsf{sendS}(\mathsf{u}'):\mathsf{t}$ *implies* $\Sigma = \{\mathsf{u}':\rho,\mathsf{u}:?(\rho)\}$ *and* $\mathcal{S} = \{\mathsf{u}\}$ *and* $\mathsf{t} =$ *Object*.
6. $\Gamma;\Sigma;\mathcal{S} \vdash \mathsf{u}.\mathsf{receiveIf}\ \{\mathsf{e}_1\}\{\mathsf{e}_2\}:\mathsf{t}$ *implies* $\Sigma = \Sigma',\mathsf{u}:?\langle\eta_1,\eta_2\rangle$ *and* $\mathcal{S} = \{\mathsf{u}\}$ *and* $\Gamma;\Sigma',\mathsf{u}:\eta_i;\{\mathsf{u}\} \vdash \mathsf{e}_i:\mathsf{t}$ *(*$i \in \{1,2\}$*)*.
7. $\Gamma;\Sigma;\mathcal{S} \vdash \mathsf{u}.\mathsf{sendIf}\ (\mathsf{e})\{\mathsf{e}_1\}\{\mathsf{e}_2\}:\mathsf{t}$ *implies* $\Sigma = \Sigma_1.\{\Sigma_2,\mathsf{u}:!\langle\eta_1,\eta_2\rangle\}$ *and* $\mathcal{S} = \{\mathsf{u}\}$ *and* $\Gamma;\Sigma_1;\{\mathsf{u}\} \vdash \mathsf{e}:\mathsf{bool}$ *and* $\Gamma;\Sigma_2,\mathsf{u}:\eta_i;\{\mathsf{u}\} \vdash \mathsf{e}_i:\mathsf{t}$ *(*$i \in \{1,2\}$*)*.
8. $\Gamma;\Sigma;\mathcal{S} \vdash \mathsf{u}.\mathsf{receiveWhile}\{\mathsf{e}\}:\mathsf{t}$ *implies* $\Sigma = \mathsf{u}:?\langle\pi\rangle^*$ *and* $\mathcal{S} = \{\mathsf{u}\}$ *and* $\Gamma;\mathsf{u}:\pi;\{\mathsf{u}\} \vdash \mathsf{e}:\mathsf{t}$.
9. $\Gamma;\Sigma;\mathcal{S} \vdash \mathsf{u}.\mathsf{sendWhile}(\mathsf{e})\{\mathsf{e}'\}:\mathsf{t}$ *implies* $\Sigma = \mathsf{u}:\pi.!\langle\pi'.\pi\rangle^*$ *and* $\mathcal{S} = \{\mathsf{u}\}$ *and* $\Gamma;\mathsf{u}:\pi;\{\mathsf{u}\} \vdash \mathsf{e}:\mathsf{bool}$ *and* $\Gamma;\mathsf{u}:\pi';\{\mathsf{u}\} \vdash \mathsf{e}':\mathsf{t}$.

Lemma D.11 (Generation Lemma for Threads).

1. $\Gamma;\Sigma \vdash \mathsf{e}:\mathsf{thread}$ *implies* $\Gamma;\Sigma;\mathcal{S} \vdash \mathsf{e}:\mathsf{t}$.
2. $\Gamma;\Sigma \vdash P_1 \mid P_2:\mathsf{thread}$ *implies* $\Sigma = \Sigma_1 \| \Sigma_2$ *and* $\Gamma;\Sigma_i \vdash P_i:\mathsf{thread}$ *(*$i \in \{1,2\}$*)*.

The last lemma shows sufficient conditions for replacing expressions by expressions inside evaluation contexts extending to session environments the partial order which takes into account reduction introduced in Section 7.

Definition D.12 (Suffix Order on Session Environments). $\Sigma \sqsubseteq \Sigma'$ if $\mathsf{u}:\eta \in \Sigma$ implies $\mathsf{u}:\eta' \in \Sigma'$ and $\eta \sqsubseteq \eta'$.

Lemma D.13 (Context Substitution).

If $\Gamma;\Sigma'.\Sigma;\mathcal{S} \vdash E[\mathsf{e}]:\mathsf{t}$ *and* $\Gamma;\Sigma';\mathcal{S}' \vdash \mathsf{e}:\mathsf{t}'$ *and* $\Gamma,\Gamma';\Sigma'';\mathcal{S}' \vdash \mathsf{e}':\mathsf{t}''$ *are such that:*

1. *all subjects in* Σ_1 *and* Γ' *are fresh;*
2. $\Sigma_2 = \Sigma_2\downarrow$*;*
3. $\Sigma_3 \sqsubseteq \Sigma_4$*;*

where $\Sigma_1 = \Sigma'' - \Sigma'$, $\Sigma_2 = \Sigma' - \Sigma''$, $\Sigma_3 = \Sigma'' - \Sigma_1$ *and* $\Sigma_4 = \Sigma' - \Sigma_2$, *then* $\Gamma,\Gamma';\Sigma''.\Sigma;\mathcal{S} \vdash E[\mathsf{e}']:\mathsf{t}$.

Proof. By induction on E.

Theorem 7.1 (Subject Reduction).

- $\Gamma;\Sigma;\mathcal{S} \vdash \mathsf{e}:\mathsf{t}$, *and* $\Gamma;\Sigma;\mathcal{S} \vdash \mathsf{e};h$ *and* $\mathsf{e},h \longrightarrow \mathsf{e}',h'$ *imply* $\Gamma';\Sigma';\mathcal{S} \vdash \mathsf{e}';h'$ *and* $\Gamma';\Sigma';\mathcal{S} \vdash \mathsf{e}':\mathsf{t}$ *with* $\Gamma \subseteq \Gamma'$ *and* $\Sigma \subseteq \Sigma'$.
- $\Gamma;\Sigma \vdash P;h$ *and* $P,h \longrightarrow P',h'$ *imply* $\Gamma';\Sigma' \vdash P;h'$ *with* $\Gamma \subseteq \Gamma'$ *and* $\Sigma \subseteq \Sigma'$.

Proof. By induction on reduction: we only consider the four more interesting cases.

Rule **Spawn**. $\Gamma;\Sigma;\mathcal{S} \vdash E[\mathsf{spawn}\{\ \mathsf{e}\ \}], h$ implies by definition $\Gamma;\Sigma;\mathcal{S} \vdash E[\mathsf{spawn}\{\ \mathsf{e}\ \}] : \mathsf{thread}$ and $\Gamma;\Sigma;\mathcal{S} \vdash h$.

$$\begin{array}{rl}
\Gamma;\Sigma;\mathcal{S} \vdash E[\mathsf{spawn}\{\ \mathsf{e}\ \}] : \mathsf{thread} \Rightarrow & \Gamma;\Sigma;\mathcal{S} \vdash E[\mathsf{spawn}\{\ \mathsf{e}\ \}] : \mathsf{t} \\
& \text{by Lemma D.11(1)} \\
\Rightarrow & \Gamma;\Sigma';\mathcal{S}' \vdash \mathsf{spawn}\{\ \mathsf{e}\ \} : \mathsf{t}' \ \&\ \Sigma' \preceq \Sigma \\
& \text{by Lemma 7.3} \\
\Rightarrow & \Sigma' = \Sigma'\downarrow \ \&\ \mathsf{t}' = \mathit{Object}\ \& \\
& \Gamma;\Sigma';\mathcal{S}' \vdash \mathsf{e} : \mathsf{t}'' \\
& \text{by Lemma D.9(14)}
\end{array}$$

$$\begin{array}{rll}
\Sigma' \preceq \Sigma \Rightarrow \Sigma = \Sigma'.\Sigma'' & & \text{by definition} \\
\Rightarrow \Gamma;\Sigma'';\mathcal{S} \vdash E[\mathsf{null}] : \mathsf{t} & & \text{by Lemma D.13 and rule } \textbf{Null} \\
\Rightarrow \Gamma;\Sigma'';\mathcal{S} \vdash E[\mathsf{null}] : \mathsf{thread} & & \text{by rule } \textbf{Start}
\end{array}$$

$$\Gamma;\Sigma';\mathcal{S}' \vdash \mathsf{e} : \mathsf{t}'' \Rightarrow \Gamma;\Sigma';\mathcal{S}' \vdash \mathsf{e} : \mathsf{thread} \text{ by rule } \textbf{Start}$$

From $\Gamma;\Sigma'' \vdash E[\mathsf{null}] : \mathsf{thread}$ and $\Gamma;\Sigma' \vdash \mathsf{e} : \mathsf{thread}$ we get $\Gamma;\Sigma''\|\Sigma' \vdash E[\mathsf{null}] \,|\, \mathsf{e} : \mathsf{thread}$ by rule **Par** and we conclude remarking that $\Sigma''\|\Sigma' = \Sigma' \cup \Sigma'' = \Sigma'.\Sigma'' = \Sigma$ since $\Sigma' = \Sigma'\downarrow$.

Rule **Connect**. $\Gamma;\Sigma;\mathcal{S} \vdash E_1[\mathsf{connect}\ \mathsf{u}_1\,\mathsf{s}\{\mathsf{e}_1\}] \,|\, E_2[\mathsf{connect}\ \mathsf{u}_2\,\overline{\mathsf{s}}\{\mathsf{e}_2\}], h$ implies by definition $\Gamma;\Sigma;\mathcal{S} \vdash E_1[\mathsf{connect}\ \mathsf{u}_1\,\mathsf{s}\{\mathsf{e}_1\}] \,|\, E_2[\mathsf{connect}\ \mathsf{u}_2\,\overline{\mathsf{s}}\{\mathsf{e}_2\}] : \mathsf{thread}$ and $\Gamma;\Sigma;\mathcal{S} \vdash h$. Let $\mathsf{s}_1 = \mathsf{s}$ and $\mathsf{s}_2 = \overline{\mathsf{s}}$.

$$\begin{array}{l}
\Gamma;\Sigma;\mathcal{S} \vdash E_1[\mathsf{connect}\ \mathsf{u}_1\,\mathsf{s}_1\{\mathsf{e}_1\}] \,|\, E_2[\mathsf{connect}\ \mathsf{u}_2\,\mathsf{s}_2\{\mathsf{e}_2\}] : \mathsf{thread} \quad \Rightarrow \\
\Gamma;\Sigma_i;\mathcal{S}_i \vdash E_i[\mathsf{connect}\ \mathsf{u}_i\,\mathsf{s}_i\{\mathsf{e}_i\}] : \mathsf{t}_i\ (i \in \{1,2\})\ \&\ \Sigma = \Sigma_1\|\Sigma_2 \\
\text{by Lemma D.11(2) and (1)} \quad \Rightarrow \\
\Gamma;\Sigma_i';\mathcal{S}_i \vdash \mathsf{connect}\ \mathsf{u}_i\,\mathsf{s}_i\{\mathsf{e}_i\}] : \mathsf{t}_i'\ (i \in \{1,2\})\ \&\ \Sigma_i' \preceq \Sigma_i \\
\text{by Lemma 7.3} \quad \Rightarrow \\
\Gamma;\emptyset \vdash \mathsf{u} : \mathsf{s}_i\ \&\ \mathsf{s}_i = \mathsf{begin}.\rho_i\ \&\ \Gamma;\Sigma_i', \mathsf{u} : \rho_i;\{\mathsf{u}_i\} \vdash \mathsf{e}_i : \mathsf{t}_i'\ (i \in \{1,2\}) \\
\text{by Lemma D.10(1)} \quad \Rightarrow \\
\Gamma;\Sigma_i', \mathsf{c} : \eta_i;\{\mathsf{c}\} \vdash \mathsf{e}_i[\mathsf{c}/\mathsf{u}_i] : \mathsf{t}_i'\ (i \in \{1,2\}) \\
\text{by Substitution Lemma} \quad \Rightarrow \\
\Gamma;\Sigma_i, \mathsf{c} : \eta_i;\{\mathsf{c}\} \vdash E_i[\mathsf{e}_i[\mathsf{c}/\mathsf{u}_i]] : \mathsf{t}_i\ (i \in \{1,2\}) \\
\text{by Lemma D.13} \quad \Rightarrow \\
\Gamma;\Sigma_i, \mathsf{c} : \eta_i \vdash E_i[\mathsf{e}_i[\mathsf{c}/\mathsf{u}_i]] : \mathsf{thread}\ \ (i \in \{1,2\}) \\
\text{by rule } \textbf{Start} \quad \Rightarrow \\
\Gamma;\Sigma, \mathsf{c} :\updownarrow \vdash E_1[\mathsf{e}_1[\mathsf{c}/\mathsf{u}_1]] \,|\, E_2[\mathsf{e}_2[\mathsf{c}/\mathsf{u}_2]] : \mathsf{thread} \\
\text{by rule } \textbf{Par}
\end{array}$$

We conclude remarking that $\Gamma;\Sigma, \mathsf{c} :\updownarrow \vdash h \cdot \mathsf{c}$.

Rule **ComS**. $\Gamma;\Sigma;\mathcal{S} \vdash E_1[\mathsf{c}.\mathsf{send}(\mathsf{v})] \,|\, E_2[\mathsf{c}.\mathsf{receive}], h$ implies by definition $\Gamma;\Sigma;\mathcal{S} \vdash E_1[\mathsf{c}.\mathsf{send}(\mathsf{v})] \,|\, E_2[\mathsf{c}.\mathsf{receive}] : \mathsf{thread}$ and $\Gamma;\Sigma;\mathcal{S} \vdash h$.

$$\begin{array}{l}
\Gamma;\Sigma;\mathcal{S} \vdash E_1[\mathsf{c}.\mathsf{send}(\mathsf{v})] \,|\, E_2[\mathsf{c}.\mathsf{receive}] : \mathsf{thread} \quad \Rightarrow \\
\Gamma;\Sigma_1;\mathcal{S}_1 \vdash E_1[\mathsf{c}.\mathsf{send}(\mathsf{v})] : \mathsf{t}_1\ \&\ \Gamma;\Sigma_2;\mathcal{S}_2 \vdash E_2[\mathsf{c}.\mathsf{receive}] : \mathsf{t}_2\ \&\ \Sigma = \Sigma_1\|\Sigma_2 \\
\text{by Lemma D.11(2) and (1)}
\end{array}$$

$\Gamma;\Sigma_1;\mathcal{S}_1 \vdash E_1[\mathsf{c}.\mathsf{send}(\mathsf{v})]:\mathsf{t}_1$ $\Rightarrow$
$\Gamma;\Sigma_1';\mathcal{S}_1' \vdash \mathsf{c}.\mathsf{send}(\mathsf{v}):\mathsf{t}_1' \;\&\; \Sigma_1' \preceq \Sigma_1$
by Lemma 7.3 $\Rightarrow$
$\Sigma_1' = \{\mathsf{c}:!\mathsf{t}_1''\}$ and $\mathcal{S}_1' = \{\mathsf{c}\}$ and $\mathsf{t}_1' = \mathit{Object}$ and $\;\&\; \Gamma;\emptyset \vdash \mathsf{v}:\mathsf{t}_1''$
by Lemmas D.10(3) and D.9 (2), (3), (4), (5) .

$\Gamma;\Sigma_2;\mathcal{S}_2 \vdash E_2[\mathsf{c}.\mathsf{receive}]:\mathsf{t}_2 \Rightarrow \Gamma;\Sigma_2';\mathcal{S}_2' \vdash \mathsf{c}.\mathsf{receive}:\mathsf{t}_2' \;\&\; \Sigma_2' \preceq \Sigma_2$ by Lemma 7.3.

$\Gamma;\Sigma_2';\mathcal{S}_2' \vdash \mathsf{c}.\mathsf{receive}:\mathsf{t}_2'$ implies $\Sigma_2' = \{\mathsf{c}:?\mathsf{t}_2'\}$ and $\mathcal{S}_2' = \{\mathsf{c}\}$ by Lemma D.10(2). Let $\mathsf{c}:!\mathsf{t}_2'.\eta \in \Sigma_1$ and $\mathsf{c}:?\mathsf{t}_2'.\overline{\eta} \in \Sigma_2$. By Lemma D.13 and rule **Null** $\Gamma;\Sigma_1'';\mathcal{S}_1 \vdash E_1[\mathsf{null}]:\mathsf{t}_1$ where $\Sigma_1'' = \Sigma_1 \setminus \mathsf{c} \cup \{\mathsf{c}:\eta\}$. From $\Gamma;\emptyset \vdash \mathsf{v}:\mathsf{t}_2'$ we get $\Gamma;\Sigma_2'';\mathcal{S}_2 \vdash E_2[\mathsf{v}]:\mathsf{t}_2$ by Lemma D.13 where $\Sigma_2'' = \Sigma_2 \setminus \mathsf{c} \cup \{\mathsf{c}:\overline{\eta}\}$. We conclude using rules **Par** and **Start**.

Rule **ComSS**. $\Gamma;\Sigma \vdash E_1[\mathsf{c}.\mathsf{receiveS}(\mathsf{x})\{\mathsf{e}\}] \mid E_2[\mathsf{c}.\mathsf{sendS}(\mathsf{c}')]:\mathsf{thread}$ implies by Lemma D.11(2) and (1) $\Gamma;\Sigma_1;\mathcal{S}_1 \vdash E_1[\mathsf{c}.\mathsf{receiveS}(\mathsf{x})\{\mathsf{e}\}]:\mathsf{t}_1$ and $\Gamma;\Sigma_2;\mathcal{S}_2 \vdash E_2[\mathsf{c}.\mathsf{sendS}(\mathsf{c}')]:\mathsf{t}_2$ with $\Sigma = \Sigma_1 \| \Sigma_2$.
From $\Gamma;\Sigma_1;\mathcal{S}_1 \vdash E_1[\mathsf{c}.\mathsf{receiveS}(\mathsf{x})\{\mathsf{e}\}]:\mathsf{t}_1$ using Lemma 7.3 and Lemma D.10(4) we obtain $\Gamma;\Sigma_1';\mathcal{S}_1' \vdash \mathsf{c}.\mathsf{receiveS}(\mathsf{x})\{\mathsf{e}\}:\mathsf{t}_1'$ with $\Sigma_1' \preceq \Sigma_1 \;\&\; \Sigma_1' = \{\mathsf{c}:?(\rho)\}.\Sigma_1'' \;\&\; \Sigma_1 = \Sigma_1'.\Sigma_1''' \;\&\; \mathcal{S}_1' = \{\mathsf{c}\} \;\&\; \mathsf{t}_1' = \mathit{Object}$.
From $\Gamma;\Sigma_2;\mathcal{S}_2 \vdash E_2[\mathsf{c}.\mathsf{sendS}(\mathsf{c}')]:\mathsf{t}_2$ using Lemma 7.3 and Lemma D.10(5) we obtain $\Gamma;\Sigma_2';\mathcal{S}_2' \vdash \mathsf{c}.\mathsf{sendS}(\mathsf{c}'):\mathsf{t}_2'$ with $\Sigma_2' \preceq \Sigma_2 \;\&\; \Sigma_2' = \{\mathsf{c}':\rho',\mathsf{c}:!(\rho')\} \;\&\; \Sigma_2 = \Sigma_2'.\Sigma_2'' \;\&\; \mathcal{S}_2' = \{\mathsf{c}\} \;\&\; \mathsf{t}_2' = \mathit{Object}$. Since $\Sigma_1 \| \Sigma_2$ is defined we also have $\rho = \rho'$.

$$\begin{array}{l}\Gamma;\Sigma_1';\{\mathsf{c}\} \vdash \mathsf{c}.\mathsf{receiveS}(\mathsf{x})\{\mathsf{e}\}:\mathsf{t}_1' \Rightarrow \Gamma\setminus\mathsf{x};\Sigma_1'',\mathsf{x}:\rho;\{\mathsf{x}\} \vdash \mathsf{spawn}\{\ \mathsf{e}\ \}:\mathsf{t}_3' \\ \qquad\qquad \text{by Lemma D.10(4)} \\ \qquad\qquad \Rightarrow \Gamma;\Sigma_1'',\mathsf{c}':\rho;\{\mathsf{c}'\} \vdash \mathsf{e}[\mathsf{c}'/\mathsf{x}]:\mathsf{t}_3' \\ \qquad\qquad \text{by Lemma D.8(2)} \\ \qquad\qquad \Rightarrow \Gamma;\Sigma_1'',\mathsf{c}':\rho \vdash \mathsf{e}[\mathsf{c}'/\mathsf{x}]:\mathsf{thread} \\ \qquad\qquad \text{by rule } \textbf{Start}\end{array}$$

From $\Gamma;\Sigma_1;\mathcal{S}_1 \vdash E_1[\mathsf{c}.\mathsf{receiveS}(\mathsf{x})\{\mathsf{e}\}]:\mathsf{t}_1$ and $\Gamma;\Sigma_1';\mathcal{S}_1' \vdash \mathsf{c}.\mathsf{receiveS}(\mathsf{x})\{\mathsf{e}\}:\mathsf{t}_1'$ we get $\Gamma;\Sigma_1''' \vdash E_1[\mathsf{null}]:\mathsf{thread}$ by Lemma D.13 and rules **Null** and **Start**.
From $\Gamma;\Sigma_2;\mathcal{S}_2 \vdash E_2[\mathsf{c}.\mathsf{sendS}(\mathsf{c}')]:\mathsf{t}_2$ and $\Gamma;\Sigma_2';\mathcal{S}_2' \vdash \mathsf{c}.\mathsf{sendS}(\mathsf{c}'):\mathsf{t}_2'$ we get $\Gamma;\Sigma_2'' \vdash E_2[\mathsf{null}]:\mathsf{thread}$ using Lemma D.13 and rules **Null** and **Start**.
Applying rule **Par** to $\Gamma;\Sigma_1'',\mathsf{c}':\rho \vdash \mathsf{e}[\mathsf{c}'/\mathsf{x}]:\mathsf{thread}$, $\Gamma;\Sigma_1''' \vdash E_1[\mathsf{null}]:\mathsf{thread}$ and $\Gamma;\Sigma_2'' \vdash E_2[\mathsf{null}]:\mathsf{thread}$ we derive $\Gamma;\Sigma_1'',\mathsf{c}':\rho\|\Sigma_1'''\|\Sigma_2'' \vdash \mathsf{e}[\mathsf{c}'/\mathsf{x}] \mid E_1[\mathsf{null}] \mid E_2[\mathsf{null}]:\mathsf{thread}$. By case analysis we can show that $\Sigma_1'',\mathsf{c}':\rho\|\Sigma_1'''\|\Sigma_2'' = \Sigma$ and this concludes the proof.

E Appendix: Proof of Progress

Lemma E.1. *If* $\Gamma;\Sigma,\mathsf{u}:\eta \vdash P:\mathsf{thread}$ *then* $P \equiv \mathsf{e} \mid Q$, $\Sigma = \Sigma_1,\mathsf{u}:\eta\|\Sigma_2$ *and* $\Gamma;\Sigma_1,\mathsf{u}:\eta \vdash \mathsf{e}:\mathsf{thread}$.

Proof. By Lemma D.11(2) and the definition of $\|$.

Lemma E.2. *Assume* P_0 *is initial and* $P_0,\emptyset \longrightarrow\!\!\!\!\!\rightarrow P,h$. *Then* $\Gamma;\Sigma \vdash P;h$ *for some* Γ, Σ, *such that all predicates in* Σ *are* $\updownarrow$.

Proof. 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 add premises to the session environments. Moreover the added premise is $\mathsf{c} : \updownarrow$ where c is the fresh created channel.

Lemma 7.7 *Let* e *be an irreducible session expression with subject* c *and* $\Gamma;\Sigma \vdash E[\mathsf{e}] : \mathsf{thread}$. *Then* $\Sigma(\mathsf{c}) = \pi.\eta$ *with* $\mathsf{e} \propto \pi$.

Proof. By Lemmas D.11(1) and 7.3 we get $\Gamma;\Sigma';\mathcal{S} \vdash \mathsf{e} : \mathsf{t}'$ for some $\Sigma' \preceq \Sigma$ and t'. Observing that the session environments in the typing of values are always empty from Lemma D.10(2), (3), (4), (5), (6), (7), (8), (9), we get $\Sigma(\mathsf{c}) = \pi.\eta$ with $\mathsf{e} \propto \pi$.

Lemma E.3. *If* $\mathsf{connect\ u\ s}\{\mathsf{e}\}$ *is a well-typed expression and* $\mathsf{e} = C[\mathsf{e}']$ *where* $C[\,]$ *is an arbitrary context and* e' *is a session expression with subject* u' *then one of the following conditions holds:*

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

Proof. By induction on $C[\,]$.

We say that e *directly originates* e' in the reduction $P,h \longrightarrow\!\!\!\!\!\rightarrow P',h'$ if one of the following conditions holds:

- $P,h \longrightarrow\!\!\!\!\!\rightarrow \mathsf{e} \,|\, Q,h_1 \longrightarrow \mathsf{e}' \,|\, Q,h_2 \longrightarrow\!\!\!\!\!\rightarrow P',h'$;
- $\mathsf{e} = E[\mathsf{spawn}\{\ \mathsf{e}_1\ \}]$, and $\mathsf{e}' = E[\mathsf{null}]$ and
  $P,h \longrightarrow\!\!\!\!\!\rightarrow E[\mathsf{spawn}\{\ \mathsf{e}_1\ \}] \,|\, \mathsf{null} \,|\, Q,h_1 \longrightarrow E[\mathsf{null}] \,|\, \mathsf{e}_1 \,|\, Q,h_2 \longrightarrow\!\!\!\!\!\rightarrow P',h'$;
- $\mathsf{e} = E[\mathsf{spawn}\{\ \mathsf{e}_1\ \}]$, and $\mathsf{e}' = \mathsf{e}_1$ and
  $P,h \longrightarrow\!\!\!\!\!\rightarrow E[\mathsf{spawn}\{\ \mathsf{e}_1\ \}] \,|\, \mathsf{null} \,|\, Q,h_1 \longrightarrow E[\mathsf{null}] \,|\, \mathsf{e}_1 \,|\, Q,h_2 \longrightarrow\!\!\!\!\!\rightarrow P',h'$;
- $\mathsf{e} = E_1[\mathsf{connect\ c\ s}\{\mathsf{e}_1\}]$, and $\mathsf{e}' = E_1[\mathsf{e}_1[\mathsf{c}'/\mathsf{c}]]$ and
  $P,h \longrightarrow\!\!\!\!\!\rightarrow E_1[\mathsf{connect\ c\ s}\{\mathsf{e}_1\}] \,|\, E_2[\mathsf{connect\ c}\ \overline{\mathsf{s}}\{\mathsf{e}_2\}] \,|\, Q,h_1 \longrightarrow$
  $E_1[\mathsf{e}_1[\mathsf{c}'/\mathsf{c}]] \,|\, E_2[\mathsf{e}_2[\mathsf{c}'/\mathsf{c}]] \,|\, Q,h_2 \longrightarrow\!\!\!\!\!\rightarrow P',h'$;
- $\mathsf{e} = E_2[\mathsf{connect\ c}\ \overline{\mathsf{s}}\{\mathsf{e}_2\}]$, and $\mathsf{e}' = E_2[\mathsf{e}_2[\mathsf{c}'/\mathsf{c}]]$ and
  $P,h \longrightarrow\!\!\!\!\!\rightarrow E_1[\mathsf{connect\ c\ s}\{\mathsf{e}_1\}] \,|\, E_2[\mathsf{connect\ c}\ \overline{\mathsf{s}}\{\mathsf{e}_2\}] \,|\, Q,h_1 \longrightarrow$
  $E_1[\mathsf{e}_1[\mathsf{c}'/\mathsf{c}]] \,|\, E_2[\mathsf{e}_2[\mathsf{c}'/\mathsf{c}]] \,|\, Q,h_2 \longrightarrow\!\!\!\!\!\rightarrow P',h'$;
- $\mathsf{e} = E_1[\mathsf{c.send}(\mathsf{v})]$, and $\mathsf{e}' = E_1[\mathsf{null}]$ and
  $P,h \longrightarrow\!\!\!\!\!\rightarrow E_1[\mathsf{c.send}(\mathsf{v})] \,|\, E_2[\mathsf{c.receive}] \,|\, Q,h_1 \longrightarrow E_1[\mathsf{null}] \,|\, E_2[\mathsf{v}] \,|\, Q,h_2 \longrightarrow\!\!\!\!\!\rightarrow P',h'$;
- $\mathsf{e} = E_2[\mathsf{c.receive}]$, and $\mathsf{e}' = E_2[\mathsf{v}]$ and
  $P,h \longrightarrow\!\!\!\!\!\rightarrow E_1[\mathsf{c.send}(\mathsf{v})] \,|\, E_2[\mathsf{c.receive}] \,|\, Q,h_1 \longrightarrow E_1[\mathsf{null}] \,|\, E_2[\mathsf{v}] \,|\, Q,h_2 \longrightarrow\!\!\!\!\!\rightarrow P',h'$;
- $\mathsf{e} = E_1[\mathsf{c.receiveS}(\mathsf{x})\{\mathsf{e}\}]$, and $\mathsf{e}' = E_1[\mathsf{null}]$ and
  $P,h \longrightarrow\!\!\!\!\!\rightarrow E_1[\mathsf{c.receiveS}(\mathsf{x})\{\mathsf{e}\}] \,|\, E_2[\mathsf{c.sendS}(\mathsf{c}')] \,|\, Q,h_1 \longrightarrow$
  $E_1[\mathsf{null}] \,|\, \mathsf{e}[\mathsf{c}'/\mathsf{x}] \,|\, E_2[\mathsf{null}] \,|\, Q,h_2 \longrightarrow\!\!\!\!\!\rightarrow P',h'$;
- $\mathsf{e} = E_1[\mathsf{c.receiveS}(\mathsf{x})\{\mathsf{e}\}]$, and $\mathsf{e}' = \mathsf{e}[\mathsf{c}'/\mathsf{x}]$ and
  $P,h \longrightarrow\!\!\!\!\!\rightarrow E_1[\mathsf{c.receiveS}(\mathsf{x})\{\mathsf{e}\}] \,|\, E_2[\mathsf{c.sendS}(\mathsf{c}')] \,|\, Q,h_1 \longrightarrow$
  $E_1[\mathsf{null}] \,|\, \mathsf{e}[\mathsf{c}'/\mathsf{x}] \,|\, E_2[\mathsf{null}] \,|\, Q,h_2 \longrightarrow\!\!\!\!\!\rightarrow P',h'$;
- $\mathsf{e} = E_2[\mathsf{c.sendS}(\mathsf{c}')]$, and $\mathsf{e}' = E_2[\mathsf{null}]$ and
  $P,h \longrightarrow\!\!\!\!\!\rightarrow E_1[\mathsf{c.receiveS}(\mathsf{x})\{\mathsf{e}\}] \,|\, E_2[\mathsf{c.sendS}(\mathsf{c}')] \,|\, Q,h_1 \longrightarrow$
  $E_1[\mathsf{null}] \,|\, \mathsf{e}[\mathsf{c}'/\mathsf{x}] \,|\, E_2[\mathsf{null}] \,|\, Q,h_2 \longrightarrow\!\!\!\!\!\rightarrow P',h'$;

- $\mathsf{e} = E_1[\mathsf{c}.\mathsf{sendIf}(\mathsf{true})\{\mathsf{e}_1\}\{\mathsf{e}_2\}]$, and $\mathsf{e}' = E_1[\mathsf{e}_1]$ and $P,h \longrightarrow\!\!\!\!\rightarrow E_1[\mathsf{c}.\mathsf{sendIf}(\mathsf{true})\{\mathsf{e}_1\}\{\mathsf{e}_2\}] \,|\, E_2[\mathsf{c}.\mathsf{receiveIf}\{\mathsf{e}_3\}\{\mathsf{e}_4\}] \,|\, Q,h_1 \longrightarrow E_1[\mathsf{e}_1] \,|\, E_2[\mathsf{e}_3] \,|\, Q,h_2 \longrightarrow\!\!\!\!\rightarrow P',h'$;
- $\mathsf{e} = E_2[\mathsf{c}.\mathsf{receiveIf}\{\mathsf{e}_3\}\{\mathsf{e}_4\}]$, and $\mathsf{e}' = E_2[\mathsf{e}_3]$ and $P,h \longrightarrow\!\!\!\!\rightarrow E_1[\mathsf{c}.\mathsf{sendIf}(\mathsf{true})\{\mathsf{e}_1\}\{\mathsf{e}_2\}] \,|\, E_2[\mathsf{c}.\mathsf{receiveIf}\{\mathsf{e}_3\}\{\mathsf{e}_4\}] \,|\, Q,h_1 \longrightarrow E_1[\mathsf{e}_1] \,|\, E_2[\mathsf{e}_3] \,|\, Q,h_2 \longrightarrow\!\!\!\!\rightarrow P',h'$;
- $\mathsf{e} = E_1[\mathsf{c}.\mathsf{sendIf}(\mathsf{false})\{\mathsf{e}_1\}\{\mathsf{e}_2\}]$, and $\mathsf{e}' = E_1[\mathsf{e}_2]$ and $P,h \longrightarrow\!\!\!\!\rightarrow E_1[\mathsf{c}.\mathsf{sendIf}(\mathsf{false})\{\mathsf{e}_1\}\{\mathsf{e}_2\}] \,|\, E_2[\mathsf{c}.\mathsf{receiveIf}\{\mathsf{e}_3\}\{\mathsf{e}_4\}] \,|\, Q,h_1 \longrightarrow E_1[\mathsf{e}_2] \,|\, E_2[\mathsf{e}_4] \,|\, Q,h_2 \longrightarrow\!\!\!\!\rightarrow P',h'$;
- $\mathsf{e} = E_2[\mathsf{c}.\mathsf{receiveIf}\{\mathsf{e}_3\}\{\mathsf{e}_4\}]$, and $\mathsf{e}' = E_2[\mathsf{e}_4]$ and $P,h \longrightarrow\!\!\!\!\rightarrow E_1[\mathsf{c}.\mathsf{sendIf}(\mathsf{false})\{\mathsf{e}_1\}\{\mathsf{e}_2\}] \,|\, E_2[\mathsf{c}.\mathsf{receiveIf}\{\mathsf{e}_3\}\{\mathsf{e}_4\}] \,|\, Q,h_1 \longrightarrow E_1[\mathsf{e}_2] \,|\, E_2[\mathsf{e}_4] \,|\, Q,h_2 \longrightarrow\!\!\!\!\rightarrow P',h'$;
- $\mathsf{e} = E_1[\mathsf{c}.\mathsf{sendWhile}(\mathsf{e})\{\mathsf{e}_1\}]$, and $\mathsf{e}' = E_1[\mathsf{c}.\mathsf{sendIf}(\mathsf{e})\{\mathsf{e}_1;\mathsf{c}.\mathsf{sendWhile}(\mathsf{e})\{\mathsf{e}_1\}\}\{\mathsf{null}\}]$ and $P,h \longrightarrow\!\!\!\!\rightarrow E_1[\mathsf{c}.\mathsf{sendWhile}(\mathsf{e})\{\mathsf{e}_1\}] \,|\, E_2[\mathsf{c}.\mathsf{receiveWhile}\{\mathsf{e}_2\}] \,|\, Q,h_1 \longrightarrow E_1[\mathsf{c}.\mathsf{sendIf}(\mathsf{e})\{\mathsf{e}_1;\mathsf{c}.\mathsf{sendWhile}(\mathsf{e})\{\mathsf{e}_1\}\}\{\mathsf{null}\}] \,|\, E_2[\mathsf{c}.\mathsf{receiveIf}\{\mathsf{e}_2;\mathsf{c}.\mathsf{receiveWhile}\{\mathsf{e}_2\}\}\{\mathsf{null}\}] \,|\, Q,h_2 \longrightarrow\!\!\!\!\rightarrow P',h'$;
- $\mathsf{e} = E_2[\mathsf{c}.\mathsf{receiveWhile}\{\mathsf{e}_2\}]$, and $\mathsf{e}' = E_2[\mathsf{c}.\mathsf{receiveIf}\{\mathsf{e}_2;\mathsf{c}.\mathsf{receiveWhile}\{\mathsf{e}_2\}\}\{\mathsf{null}\}]$ and $P,h \longrightarrow\!\!\!\!\rightarrow E_1[\mathsf{c}.\mathsf{sendWhile}(\mathsf{e})\{\mathsf{e}_1\}] \,|\, E_2[\mathsf{c}.\mathsf{receiveWhile}\{\mathsf{e}_2\}] \,|\, Q,h_1 \longrightarrow E_1[\mathsf{c}.\mathsf{sendIf}(\mathsf{e})\{\mathsf{e}_1;\mathsf{c}.\mathsf{sendWhile}(\mathsf{e})\{\mathsf{e}_1\}\}\{\mathsf{null}\}] \,|\, E_2[\mathsf{c}.\mathsf{receiveIf}\{\mathsf{e}_2;\mathsf{c}.\mathsf{receiveWhile}\{\mathsf{e}_2\}\}\{\mathsf{null}\}] \,|\, Q,h_2 \longrightarrow\!\!\!\!\rightarrow P',h'$;

The relation e *originates* e' in the reduction $P,h \longrightarrow\!\!\!\!\rightarrow P',h'$ is the reflexive and transitive closure of e *directly originates* e' in the reduction $P,h \longrightarrow\!\!\!\!\rightarrow P',h'$.

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 c_0, $\mathsf{c}_1,\ldots$. This means that if $P,h \longrightarrow\!\!\!\!\rightarrow Q,h' \longrightarrow\!\!\!\!\rightarrow R,h''$ and c_i is a channel created in the reduction $P,h \longrightarrow\!\!\!\!\rightarrow Q,h'$, and c_j is a channel created in the reduction $Q,h' \longrightarrow\!\!\!\!\rightarrow R,h''$, then $i < j$. This allows us to define the *active channel* of a session environment as the channel with the maximum index which is the subject of a premise whose predicate is not the empty live channel type (i.e. ε or end).

Lemma E.4. *Assume P_0 is initial and $P_0,\emptyset \longrightarrow\!\!\!\!\rightarrow P,h$ and $\Gamma;\Sigma \vdash P;h$, and $P \equiv E[\mathsf{e}] \,|\, Q$ where e is an irreducible session expression, and $\Sigma = \Sigma_1 \| \Sigma_2$ and $\Gamma;\Sigma_1 \vdash E[\mathsf{e}] : \mathsf{thread}$. If c is the active channel of Σ_1 then c is the subject of e.*

Proof. Assume by contradiction that the subject of e is c_j with $j \neq i$ and $\mathsf{c}_j \in \mathsf{dom}(\Sigma_1)$; by construction $j < i$. Let e' be an expression which originates $E[\mathsf{e}]$ in the reduction $P_0,\emptyset \longrightarrow\!\!\!\!\rightarrow P,h$ and which does not contain c_i or c_j. We need to examine some possible cases:

- e' has open first a session on c_j and then a session on c_i: in this case we cannot use c_j as subject by Lemma E.3;
- e' has open first a session on c_i and then a session on c_j: this case is impossible, since $j < i$;
- e' has first received c_j and then has open a session on c_i: in this case we cannot use c_j as subject by Lemma E.3;
- e' has first received c_i and then has open a session on c_j: this case is impossible, since $j < i$;

– e' has received both channels c_i and c_j: this is impossible too, since after reception of a new channel a new thread is always generated, so two received channels cannot belong to the session environment of the same expression.

Theorem 7.5 (Progress). *Assume P_0 is initial and $P_0, \emptyset \rightarrow\!\!\!\!\rightarrow P, h$. Then one of the following holds.*

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

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

The only interesting case is $P \equiv V \,|\, Q$, where V is a parallel of values and Q is a parallel of evaluation contexts containing irreducible session expressions. Let $Q \equiv \prod_{0 \leq i < n} E_i[\mathsf{e}_i]$. By Subject Reduction we have $\Gamma; \Sigma \vdash P; h$. This implies $\Sigma = \Sigma_0 \| \ldots \| \Sigma_{n-1}$ and $\Gamma; \Sigma_i \vdash E_i[\mathsf{e}_i] : \mathsf{thread}$ by Lemma D.11(2). Each Σ_i has an active channel: let c be the active channel with the maximum index between all these. By Lemma E.2 $\Sigma(\mathsf{c}) = \updownarrow$ and then by definition of $\|$ each channel occurs exactly in two session environments between $\Sigma_1, \ldots, \Sigma_n$ with dual live channel types. Let c occurs in Σ_1 and Σ_2 with types $\pi.\eta$ and $\overline{\pi.\eta}$ respectively. Then by Lemma E.4 c is the subject of e_1 and e_2. Lemma 7.7 gives $\mathsf{e}_1 \propto \pi$ and $\mathsf{e}_2 \propto \overline{\pi}$. Therefore e_1 and e_2 are dual of each other and can communicate.

Theorem 7.9 (Communication-Order Preserving). *Let P_0 be initial. Assume that $P_0, \emptyset \rightarrow\!\!\!\!\rightarrow E[\mathsf{e}_0] \,|\, Q, h \longrightarrow P', h'$ 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) *$\Gamma; \Sigma, \mathsf{c} : \eta \vdash E[\mathsf{e}_0] : \mathsf{thread}$ and $\Gamma; \Sigma', \mathsf{c} : \overline{\eta} \vdash E'[\mathsf{e}'_0] : \mathsf{thread}$; and*
   (c) *$\Gamma; \hat{\Sigma}, \mathsf{c} : \eta' \vdash \mathsf{e} : \mathsf{thread}$ and $\Gamma; \hat{\Sigma}', \mathsf{c} : \overline{\eta}' \vdash \mathsf{e}' : \mathsf{thread}$ with $\eta' \sqsubseteq \eta$.*

Proof. By the proof of the Progress Theorem (Theorem 7.5) if the reduction step

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

does not reduce Q alone, then $Q \equiv E'[\mathsf{e}'_0] \,|\, R$. By the Subject Reduction Theorem (Theorem 7.1) $\Gamma; \check{\Sigma} \vdash E[\mathsf{e}_0] \,|\, E'[\mathsf{e}'_0] \,|\, R : \mathsf{thread}$, which implies by Lemma D.11(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 channel c is the active channel of Σ_1 and Σ_2 with dual live channel types. Let $\Sigma_1 = \Sigma, \mathsf{c} : \pi.\eta$ and $\Sigma_2 = \Sigma', \mathsf{c} : \overline{\pi.\eta}$. Lemma 7.7 gives $\mathsf{e}_0 \propto \pi$ and $\mathsf{e}'_0 \propto \overline{\pi}$. Therefore e_0 and e'_0 are session expressions dual of each other and then they can communicate. Thus we have:

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

We consider only the case $\mathsf{e}_0 \equiv \mathsf{c}.\mathsf{receive}$ and $\mathsf{e}'_0 \equiv \mathsf{c}.\mathsf{send}(\mathsf{v})$ and $\pi = ?\mathsf{t}$, the proofs in all cases being similar. 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**. Lastly from the proof the Subject Reduction Theorem we get $\Gamma; \Sigma, \mathsf{c} : \eta \vdash \mathsf{e} : \mathsf{thread}$ and $\Gamma; \Sigma', \mathsf{c} : \overline{\eta} \vdash \mathsf{e}' : \mathsf{thread}$.

F Appendix: Subject Reduction and Progress in Related Work

Subject Reduction In all previously mentioned papers typability guarantees absence of run-time communication errors. However not all of them have the subject reduction property. The problem comes from sending a live channel to a thread which already uses this channel to communicate, as in our Example 4.2. In the calculi of [5, 12, 18], this example translates as follows:

```
accept x(u) in
 accept y(w) in
   throw w[u];
```

```
request x(u) in
 request y(w) in
  catch w(z) in
   z![5];
   u?(j);
```

These two processes in parallel are well typed in [18]: they reduce to $(\nu u)(u![5];u?(j))$ which is stuck and not typable, since a channel restriction can be typed only if in its body the channel type is $\perp$.

The type system of [5] rejects the parallel of the initial processes since a channel name is required to be fresh in order to be catch.

Using channel polarities and requiring a "balancing" condition for channel types only at top level $(\nu u)(u![5];u?(j))$ is typable also in the system of [12].

The same example can be rewritten in the language of [27] as follows, and there again it breaks subject reduction.

```
// fun1 x y =
let u = request x in
  let w = request y in
    let z = receive w in
      send 5 on z;
      let j = receive u in
        close u; close w
```

```
// fun2 x y =
let u = accept x in
  let w = accept y in
    send u on w;
    close w;
```

Progress Our present work has been influenced by the analysis done in all previously mentioned papers, however our language is the only guaranteeing absence of deadlocks. For example in the language of [27] we can type the parallel of the following processes (obtained by translating the threads of Example 5.3):

```
// fun1 x y =
let u = request x in
  let w = request y in
    let i = receive u in
      let j = receive w in
        close u; close w;
```

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

Note that in the above two nested 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 [5, 12, 18], where previous example is written as follows:

```
1  request x(u) in
2    request y(w) in
3      u?(i);
4          w?(j );
```

```
1  accept x(u)
2    accept y(w)
3      w![5];
4        u![6];
```