Bounded Session Types for Object Oriented Languages*

Mariangiola Dezani-Ciancaglini[1], Elena Giachino[1], Sophia Drossopoulou[2], and Nobuko Yoshida[2]

[1] Dipartimento di Informatica, Università di Torino dezani,giachino@di.unito.it
[2] Department of Computing, Imperial College London scd,yoshida@doc.ic.ac.uk

Abstract. Earlier work explored the introduction of session types into object oriented languages. Following the session types literature, two parties would start communicating, provided the types attached to that communication, *i.e.* the corresponding session types, were dual of each other. Then, the type system was able to ensure soundness, in the sense that two communicating partners were guaranteed to receive/send sequences of values following the order specified by their session types.
In the current paper we improve upon our earlier work in two ways: we extend the type system to support bounded polymorphism, and we make the selection more object-oriented, so that control structures determine how to continue evaluation, depending on the class of the object being sent/received.
Interestingly, although our notion of selection is more powerful than that in earlier work, the ensuing system turned out not to be more complex, except for the notion of duality, which needed to be extended, to correctly deal with bounded polymorphism, and to capture the new notion of selection.
The paper contains an example, informal explanations, a formal description of the operational semantics and of type system, and a proof of subject reduction.

1 Introduction

In earlier work [12], some of the authors of this paper explored the incorporation of session types [19] into object oriented languages through Moose, a minimal object oriented language extended with the notion of a *session*. A session is established when two parties *connect*, using *session types* which are dual to each other. Session types express sequences and iterations of types of values

* This work was partly funded by FP6-2004-510996 Coordination Action TYPES, by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project and by EPSRC GR/T03208, GR/S55538, GR/T04724 and GR/S68071. This paper reflects only the authors' views and the Community is not liable for any use that may be made of the information contained therein.

being received/sent. The *dual* of a session type is a session type where receipt is replaced by sending with a smaller type, and vice versa.

After such a session is established, the two parties can communicate by sending to, or receiving from, each other, objects of the types prescribed in their session type. Through the use of such session types, we could ensure soundness, in the sense that two communicating partners were guaranteed to receive/send values as expected in their session type.

In the current paper we describe the language MOOSE$_{<:}$ which extends MOOSE in the following two ways. Firstly, following ideas from [16, 18] we support bounded polymorphism in session types. Thus, the following fragment of a session type

```
?(X<:Image).!X
```

expresses that an object of a subclass of `Image` will be received, and then an object of the same class will be sent. This clearly agrees with the standard use of bounded polymorphism for λ-calculus [26].

Secondly, we have made the notion of selection more object-oriented, so that it is possible to determine branch selection and iteration based on the class of the object being sent/received. For example, the following fragment of a session type

```
?(X<:Image).?((((Y<:ArrayList).!X)*,(Z<:Image).ε)).?Address
```

expresses that first an object of class `X`, a subclass of `Image`, will be received, then a value will be received. If that value is an `ArrayList`, then an `X` will be sent, and this will be repeated until a value of class different from `ArrayList`, *i.e.* `Image`, is received. After that, an `Address` will be received.

A significant part of the design effort for MOOSE$_{<:}$was devoted to the design of the type system. In order to fully exploit the expressive power of bounded polymorphism we developed a sophisticated notion of duality between session types. The choice based on the class of exchanged objects has required particular care in determining the typing rules.

The type system of MOOSE can also achieve *progress* by taking into account the current channel used to communicate [12]. Because this analysis is orthogonal to the extensions from MOOSE to MOOSE$_{<:}$, and because progress for MOOSE$_{<:}$ could be obtained exactly as for MOOSE, in this work we do not explore this issue any further.

Related Papers Session types have been proposed first in [19] for the π-calculus and then they have been studied for several different settings, *i.e.* for π-calculus-based formalisms [1, 16, 20, 24, 28], for CORBA [29], for functional languages [11, 17, 30], for object-oriented languages [10, 12, 13], for boxed ambients [15], and recently, for CDL, a W3C standard description language for web services [5, 6, 21, 27, 32]. In this paper we essentially extend the language of [12].

Bounded quantification for object-oriented programming was introduced in [7] as a means of typing functions that operate uniformly over all subtypes of a given type. In order to deal with recursively defined types, bounded quantification was generalised to F-bounded quantification [4]. Pizza [25] is a strict superset of Java that incorporates F-bounded polymorphism. The recently-released version

1.5 of Java adds bounded polymorphism to the language. It is based on a proposal known as GJ (Generic Java) [2]. C# also supports bounded polymorphism. The language PolyTOIL [3] has match-bounded polymorphism that provides a very flexible yet safe type discipline for object-oriented programming. In fact the matching relation is more general than subtyping on object types. We only consider here bounded quantification.

[18] is the first study of bounded polymorphism in the π-calculus, and the first study of any form of polymorphism in relation to session types. In that paper polymorphism is associated with the labels in the branching types. We instead allow to bound all received values in session types.

The selection on the basis of the exchanged object class is reminiscent of the semantic subtyping approach [8, 9, 14].

Paper structure We express our ideas through the language MOOSE$_{<:}$. In Section 2 we give an example, in Sections 3, 4, 5 we give the formal system; in 5.3 we outline soundness, and in the Appendix we give the full proof.

2 Example: Collaborative Card Design

In this section, we describe MOOSE$_{<:}$ through an example, which expresses a typical collaboration pattern, *c.f.* [5, 6, 32], and which uses our new primitives of bounded polymorphism in session types, and branch selection according to the dynamic type of objects.

This simple protocol contains essential features which demonstrate the expressivity of the new features of MOOSE$_{<:}$, *i.e.* bounded polymorphism in session types for object oriented languages and branch selection according to the dynamic types of objects.

A card producer and a card customer collaborate for the design of a card that would please the customer. The design is based on two original photos, and

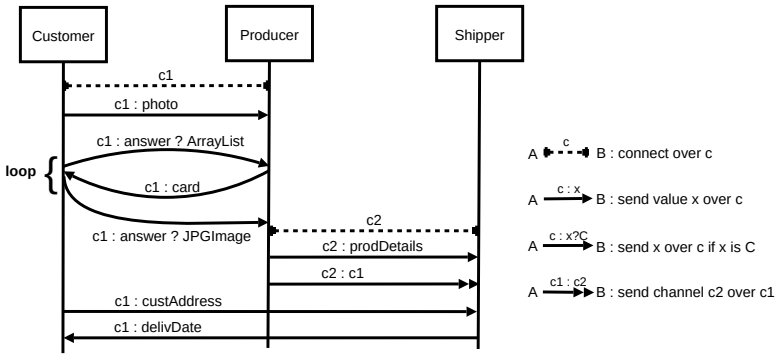

Fig. 1. Collaborative Card Design.

some card samples sent by the customer. The producer creates a new card based on the customer's originals and samples, and the customer examines the created card, and sends new samples. This process repeats itself (iterates), until the customer is satisfied. The customer expresses his satisfaction (and consequently the end of the iteration) by sending a single image, rather than further card samples. Thus, branch selection in control structures (here iteration) is based on the dynamic type of an object sent.

```
1  session AcceptCards =
2    begin.?(X1<:Image).?(X2<:X1).?((((Y<:ArrayList).!X1)*, (Z<:Image).ε)).
3    ?Address.!DeliveryDetails.end
4  session RequestJPGCards =
5    begin.!JPGImage.!JPGImage.!((ArrayList.?(X<:JPGImage))*, Image.ε)).
6    !Address.?DeliveryDetails.end
7  session RequestDelivery =
8    begin.!CardSet.!(?Address.!DeliveryDetails.end).end
9  session AcceptDelivery =
10   begin.?CardSet.?(?Address.!DeliveryDetails.end).end
```

Fig. 2. Session Types for the Card Producer-Customer-Shipper Example.

Furthermore, the card producer needs to be capable to collaborate with customers who use different image formats. If the customer sends two `JPG` originals, then the card producer should create `JPG` images too. If the customer sends two `GIF` originals, then the card producer should create `GIF` images too. The costumer is not allowed to sent the two images in different formats, for example one `JPG` and one `GIF` images. We express all this through bounded polymorphism.

In Fig. 1 we show a sequence diagram for the example we described above. The `CardProducer` and `JPGCardCustomer` participants initiate interaction over channel `c1`. The `JPGCardCustomer` sends to the `CardProducer` two photos in `JPG` format followed by his decision, which is either a list of cards (which represent samples and hints for the production of a new card), or just a single, chosen, card. The `CardProducer` examines the decision; if it is a list of cards, then he designs a new card and sends it to the `JPGCardCustomer`, and the process of card design iterates. If the `JPGCardCustomer`'s decision is a single, chosen card, then the iteration terminates, and the `CardProducer` prints the required number of copies of the chosen card. Then the `CardProducer` connects with the `Shipper` over channel `c2` and sends him the printed cards. He then delegates his part of the remaining activity with the `JPGCardCustomer` to the `Shipper`; the latter is realised by sending `c1` over `c2`. Now the `Shipper` will await `JPGCardCustomer`'s address, before responding with the delivery date.

In Fig. 2 we declare the necessary session types, and in Fig. 3, Fig. 4 and Fig. 5 we encode the given scenario in MOOSE$_{<:}$, using one class per protocol participant.

The session types `RequestJPGCards` and `AcceptCards` describe the communication pattern between the `CardProducer` and the `JPGCardCustomer`. The session type `AcceptCards` describes accepting a first image in *any* format, a second image in the same format of the first one, followed by an iteration. The iteration is repeated as long as the received object is an `ArrayList`, in which case an image, of the same class as the two received is sent; the iteration stops if an `Image` is received[3]. After the iteration, an address is received and the delivery details are sent. The session type `RequestJPGCards` models the sending of two images in `JPG` format, followed by an iteration. The iteration is repeated as long as an `ArrayList` is sent, in which case an `Image` is received; the iteration stops if an `Image` is sent[4]; afterwards, an address is sent and the delivery details are received. The interesting observation is that (under the assumption that `JPGImage` is a subclass of `Image`) the types `RequestJPGCards` and `AcceptCards` represent *dual* behaviours associated with the same session, in which the sending of a value in one end corresponds to its reception at the other.

Thus, the implementor of `AcceptCards` could also collaborate with a thread which required `GIF` instead of `JPG` images. Such a thread would have a type like:

```
1 session RequestGIFCards =
2   begin.!GIFImage.!GIFImage.!⟪(ArrayList.?(X<:GIFImage))*, Image.ε⟫.
3   !Address.?DeliveryDetails.end
```

In our terminology, `AcceptCards` is a dual of `RequestJPGCards` as well as of `RequestGIFCards`. Thus, in MOOSE$_{<:}$ more than one session type may be dual of another; therefore, our notion of duality is not standard.

Equally important is the assurance that each value received will belong to the type expected by the receiving party, where the latter can depend on the types of previously exchanged objects. For example, according to `AcceptCards`, the object sent in the communication within the cycle will belong to a subclass of that of the object received in the first communication.

The session type `RequestDelivery` describes sending the printed cards, followed by a *live* session channel of remaining type `?Address.!DeliveryDetails.end`. The session type `AcceptDelivery` is simply `RequestDelivery` with all the external ! and ? exchanged.

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 `sellCards` of class `CardProducer` contains a `connect` statement that implements the session type `AcceptCards`, while the method `buyCards` of class `JPGCardCustomer` contains a `connect` statement over the same channel and with the session type `RequestJPGCards`. When a `CardProducer` and a `JPGCardCustomer` are executing concurrently the method

[3] The iteration also stops if an object is received which is neither an `ArrayList` nor an `Image`. More in the next sections.

[4] As before, the iteration also stops if an object is sent which is neither an `ArrayList` nor an `Image`.

```
1   class CardProducer {
2
3       (X <: Image) createCard(X x1, Image x2, ArrayList y) {...} //impl.
            omitted
4
5       void sellCards() {
6          connect c1 AcceptCards {
7             c1.receive(x1) {
8             c1.receive(x2) {
9                c1.receiveWhile(y) {
10                  ArrayList ▷ c1.send(createCard(x1, x2, y));
11               }{ Image ▷ Image card := y;}
12            } }
13            CardSet cardSet := cardPrint(card); //impl. omitted
14            // print the required number of copies of the chosen card
15            spawn { connect c2 RequestDelivery {
16                      c2.send(cardSet); c2.sendS(c1);} }
17         } /* End connect */
18      } /* End method sellCards */
19
20  }
```

Fig. 3. Code for the CardProducer.

`sellCards` and `buyCards` respectively, they can engage in a session, which will result in a fresh channel being replaced for occurrences of the shared channel `c1` within both session bodies; freshness guarantees that the new channel only occurs in these two threads, therefore the objects can proceed to perform their interactions without the possibility of external interference.

The type of method `createCard` of the class `CardProducer` is *parameterized*, in that its return type is the class of its first argument, and it is a subclass of the class `Image`. We simplified Java syntax for polymorphic methods in an obvious way, and following the notation from [31].

After starting a session in the body of method `sellCards()`, two photos are received using `c1.receive(x1)`, `c1.receive(x2)` and replace `x1, x2`. Then an object is received by the iterative expression `c1.receiveWhile(y)`: while the received object is an array list, a new card – designed out of the photo and of the list of cards – is sent using `c1.send(createCard(x1, x2, y))`. If the received object is an image, the iteration stops. Then, the required copies of this card are printed and a new thread is spawned. The body of the spawn expression has a nested `connect`, via which printed cards are sent to the `Shipper`. Then the actual runtime channel, *i.e.* the channel which substituted `c1` when the outer `connect` took place, is sent through the construct `c2.sendS(c1)`. The latter is an example of *higher-order session communication*.

The method `buyCards` uses the field `cardList` to keep track of the cards proposed by the `CardProducer`. Once the session has started, the two `photos`

```
class JPGCardCustomer {

     JPGImage photo1;
     JPGImage photo2;
     ArrayList cardList;
     Address addr;
     DeliveryDetails dDetails;

     void buyCards(JPGImage photo) {
        connect c1 RequestJPGCards {
           c1.send(photo1);
           c1.send(photo2);
           Object answer := examine(cardList); //impl. omitted
           c1.sendWhile(answer) {
              Arraylist ▷ c1.receive(x) {
                 cardList.add(x);
                 answer := examine(cardList);}
              }{ Image ▷ null; }
           c1.send(addr);
           c1.receive(z) { dDetails := z; };
        } /* End connect */
     } /* End method buyCards */

}
```

Fig. 4. Code for the JPGCardCustomer.

```
class Shipper {

    void delivery() {
       connect c2 AcceptDelivery {
          c2.receive(x) { CardSet cardSet := x };
          c2.receiveS(x) {
             x.receive(y) { Address custAddress := y };
             DeliveryDetails delivDetails := new DeliveryDetails();
             //... set state of delivDetails
             x.send(delivDetails);
          }
       } /* End connect */
    } /* End method delivery */

}
```

Fig. 5. Code for the Shipper.

are sent using `c1.send(photo1)`, `c1.send(photo2)`, and a card is received using `c1.receive(x)`; this card replaces `x`. The method `examine` takes `cardList` and it returns an `answer`, which is sent using `c1.sendWhile(answer)`. If the answer is a list of cards (this is meant to happen if the `JPGCardCostumer` does not like any of the proposed cards, and then the answer is meant to contain suggestions of changes), then a new card is received through `c1.receive(x)` and added to the card list through `cardList.add(x)`, a new `answer` is produced through the call of method `examine`, and the iteration continues. If the answer is an image (this is meant to contain the card chosen by the `JPGCardCostumer`), then the iteration stops. Then, the customer's address, `addr`, is sent and an instance of `DeliveryDetails` is received.

Notice that in order to get an arbitrary number of repetitions, it is crucial to allow objects of different classes to be sent in the different iterations of `sendWhile`.

3 Syntax

In Fig. 6 we describe the syntax of MOOSE$_{<:}$, which extends the language MOOSE [12] to support bounded polymorphism and choice of session communication depending on object classes. We distinguish *user syntax*, *i.e.* source level code, and *runtime syntax*, which includes null pointer exceptions, threads and heaps.

Channels We distinguish *shared channels* and *live channels*. Shared channels have not yet been connected; they are used to decide if two threads can communicate, in which case they are replaced by fresh live channels. After a connection has been created the channel is live; data may be transmitted through such active channels only.

(type)	t	$::= \mathrm{X} \mid C \mid \mathsf{s} \mid (\mathsf{s},\mathsf{s})$
(class)	*class*	$::= \mathsf{class}\ C\ \mathsf{extends}\ C\ \{\ \tilde{\mathsf{f}}\tilde{\mathsf{t}}\ \ \tilde{meth}\ \}$
(method)	*meth*	$::= \mathsf{t}\,\mathsf{m}\ (\tilde{\mathsf{t}}\ \tilde{x},\tilde{\rho}\,\tilde{y})\ \{\mathsf{e}\} \mid (X <: \mathsf{t})\,\mathsf{m}\ (\tilde{\mathsf{t}}\ \tilde{x},\tilde{\rho}\,\tilde{y})\ \{\mathsf{e}\}$
(expression)	e	$::= \mathsf{x} \mid \mathsf{v} \mid \mathsf{this} \mid \mathsf{e};\mathsf{e} \mid \mathsf{e.f} := \mathsf{e} \mid \mathsf{e.f} \mid \mathsf{e.m}(\tilde{\mathsf{e}}) \mid \mathsf{new}\ C$
		$\mid \mathsf{new}\ (\mathsf{s},\mathsf{s}) \mid \mathsf{NullExc} \mid \mathsf{spawn}\{\ \mathsf{e}\ \} \mid \mathsf{connect\ u\ s}\{\mathsf{e}\}$
		$\mid \mathsf{u.send}(\mathsf{e}) \mid \mathsf{u.receive}(\mathsf{x})\{\mathsf{e}\} \mid \mathsf{u.sendS}(\mathsf{u}) \mid \mathsf{u.receiveS}(\mathsf{x})\{\mathsf{e}\}$
		$\mid \mathsf{u.sendCase}(\mathsf{e})\{\tilde{C} \triangleright \tilde{\mathsf{e}}\} \mid \mathsf{u.receiveCase}(\mathsf{x})\{\tilde{C} \triangleright \tilde{\mathsf{e}}\}$
		$\mid \mathsf{u.sendWhile}(\mathsf{e})\{\tilde{C} \triangleright \tilde{\mathsf{e}}\}\{\tilde{C} \triangleright \tilde{\mathsf{e}}\}$
		$\mid \mathsf{u.receiveWhile}(\mathsf{x})\{\tilde{C} \triangleright \tilde{\mathsf{e}}\}\{\tilde{C} \triangleright \tilde{\mathsf{e}}\}$
(channel)	u	$::= \mathsf{c} \mid \mathsf{x}$
(value)	v	$::= \mathsf{c} \mid \mathsf{null} \mid \mathsf{o}$
(thread)	P	$::= \mathsf{e} \mid P \mid P$

Fig. 6. Syntax, where syntax occurring only at runtime appears shaded.

User syntax The metavariable t ranges over types for expressions, ρ ranges over running session types, C ranges over class names and s ranges over shared session types. We introduce the full syntax of types in § 5.

Class declarations are as expected, except for the restriction that object fields cannot contain live channels. Without this restriction session would not behave as required, as shown in Example 5.1 of [12].

The method declaration $\mathsf{t}\,\mathsf{m}\ (\tilde{\mathsf{t}}\,\tilde{\mathsf{x}}, \tilde{\rho}\,\tilde{\mathsf{y}})\ \{\mathsf{e}\}$ introduces a standard method, while the method declaration $(X <: \mathsf{t})\,\mathsf{m}\ (\tilde{\mathsf{t}}\,\tilde{\mathsf{x}}, \tilde{\rho}\,\tilde{\mathsf{y}})\ \{\mathsf{e}\}$ introduces a parameterized method whose result has the type of the first of its parameters and this type is bound by t [5].

The syntax of user expressions e is standard except for the channel constructor $\mathsf{new}\ (\mathsf{s}, \mathsf{s}')$, which builds a fresh shared channel used to establish a private session, 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 describes the syntax for parameters, values, the self identifier this, sequence of expressions, assignment to fields, field access, method call, and object creation. The values are channels and null. Threads may be created through $\mathsf{spawn}\{\ \mathsf{e}\ \}$, 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 $\{\mathsf{e}\}$ in session communications that agree with the 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.

The expressions $\mathsf{u}.\mathsf{send}(\mathsf{e})$ and $\mathsf{u}.\mathsf{receive}(\mathsf{x})\{\mathsf{e}\}$ exchange values (which can be shared channels): the former evaluates e and sends the result over u, while the latter receives a value via u that will be bound to x within e. The expressions $\mathsf{u}.\mathsf{sendS}(\mathsf{u}')$ and $\mathsf{u}.\mathsf{receiveS}(\mathsf{x})\{\mathsf{e}\}$ exchange live channels: 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 next four primitives are extended from those in [12]; they allow choice of communication on the basis of the class of an object sent/received.

The expression $\mathsf{u}.\mathsf{sendCase}(\mathsf{e})\{C_1 \triangleright \mathsf{e}_1; \cdots; C_n \triangleright \mathsf{e}_n\}$ first evaluates the expression e to an object o, and sends o over u. It continues with e_i, where i is the smallest index in $\{1, ..., n\}$ such that o is C_i, if such an i exists. Otherwise, it returns null. The expression $\mathsf{c}.\mathsf{receiveCase}(\mathsf{x})\{C_1 \triangleright \mathsf{e}_1; \cdots; C_n \triangleright \mathsf{e}_n\}$ receives an object o via channel u and binds it to x. It continues with $\mathsf{e}_i[\mathsf{o}/\mathsf{x}]$, where i is the smallest index in $\{1, ..., n\}$ such that o is C_i, if such an i exists. Otherwise, it returns null.

The expressions $\mathsf{u}.\mathsf{sendWhile}(\mathsf{e})\{C_1 \triangleright \mathsf{e}_1; \cdots; C_n \triangleright \mathsf{e}_n\}\{D_1 \triangleright \mathsf{d}_1; \cdots; D_m \triangleright \mathsf{d}_m\}$ and $\mathsf{u}.\mathsf{receiveWhile}(\mathsf{x})\{C_1 \triangleright \mathsf{e}_1; \cdots; C_n \triangleright \mathsf{e}_n\}\{D_1 \triangleright \mathsf{d}_1; \cdots; D_m \triangleright \mathsf{d}_m\}$ express *iterative* communication. The expression $\mathsf{u}.\mathsf{sendWhile}(\mathsf{e})\{C_1 \triangleright \mathsf{e}_1; \cdots; C_n \triangleright \mathsf{e}_n\}\{D_1 \triangleright \mathsf{d}_1; \cdots; D_m \triangleright \mathsf{d}_m\}$ evaluates e to an object o and sends it over u. Then it continues with e_i and iterates, where i is the smallest index in $\{1, ..., n\}$ such that

[5] We do not expect any technical difficulties in allowing standard parameterized methods. We restricted the result type in this way in order to focus on the features of the object oriented paradigm which interact with sessions.

o is C_i, if such i exists. If no such i exists, then it continues with d_j, where j is the smallest index in $\{1, ..., m\}$ such that o is D_j, if such a j exists. Otherwise, it returns null. The meaning of the expression $\mathsf{u}.\mathsf{receiveWhile}(\mathsf{x})\{C_1 \triangleright \mathsf{e}_1; \cdots; C_n \triangleright \mathsf{e}_n\}\{D_1 \triangleright \mathsf{d}_1; \cdots; D_m \triangleright \mathsf{d}_m\}$ is analogous.

Runtime syntax The runtime syntax (shown shaded in Fig. 6) 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.

4 Operational Semantics

This section presents the operational semantics of MOOSE$_{<:}$, which is mainly inspired by the language MOOSE of [12]. We only discuss the more interesting rules. First we list the evaluation contexts.

$$\begin{array}{rl} E ::= & [\,] \mid E.\mathsf{f} \mid E; \mathsf{e} \mid E.\mathsf{f} := \mathsf{e} \mid \mathsf{o}.\mathsf{f} := E \mid E.\mathsf{m}(\tilde{\mathsf{e}}) \mid \mathsf{o}.\mathsf{m}(\tilde{\mathsf{v}}, E, \tilde{\mathsf{e}}) \\ \mid & \mathsf{c}.\mathsf{send}(E) \mid \mathsf{c}.\mathsf{sendCase}(E)\{C_1 \triangleright \mathsf{e}_1; \cdots; C_n \triangleright \mathsf{e}_n\} \end{array}$$

Fig. 7 defines auxiliary functions used in the operational semantics and typing rules. We assume a fixed, global class table $\mathtt{CT}$, which contains *Object* as top-most class.

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

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

Heaps, ranged over h, are built inductively using the heap composition operator "::", and contain mappings of object identifiers to instances of classes, and channels. In particular, a heap will contain the set of objects and *fresh* channels, both shared and live, that have been created since the beginning of execution. The heap produced by composing $h :: [\mathsf{o} \mapsto (C, \tilde{\mathsf{f}} : \tilde{\mathsf{v}})]$ 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 $h :: \mathsf{c}$ will contain the fresh channel c. Heap membership for object identifiers and channels is checked using standard set notation, we therefore write it as $\mathsf{o} \in h$ and $\mathsf{c} \in h$, respectively. Heap update for objects is written $h[\mathsf{o} \mapsto (C, \tilde{\mathsf{f}} : \tilde{\mathsf{v}})]$, and field update is written $(C, \tilde{\mathsf{f}} : \tilde{\mathsf{v}})[\mathsf{f} \mapsto \mathsf{v}]$.

Expressions Fig. 8 shows the rules for execution of expressions which correspond to the sequential part of the language. These are identical to the rules of [12] except for the addition of a fresh shared channel to the heap (rule **NewS-R**). In this rule we assume the two session types s and s' to be *dual*. The duality relation $\bowtie$ will be defined in Fig. 15. In rule **NewC-R** the auxiliary function $\mathsf{fields}(C)$ examines the class table and returns the field declarations for C. The

Field lookup

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

Method lookup

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

Method type lookup

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

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

Method body lookup

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

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

σ is either $\mathtt{t}$ or ρ
T is either $\mathtt{t}$ or $(X <: \mathtt{t})$.

Fig. 7. Lookup Functions.

method invocation rule is **Meth-R**; 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 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. 9, are given modulo the standard structural equivalence rules of the π-calculus [23], written $\equiv$. We define *multi-step* reduction as: $\twoheadrightarrow \stackrel{\mathsf{def}}{=} (\longrightarrow \cup \equiv)^*$. All rules are essentially taken from [12], except for the last three rules which support branching of the communications depending on the class of the object send/received.

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.

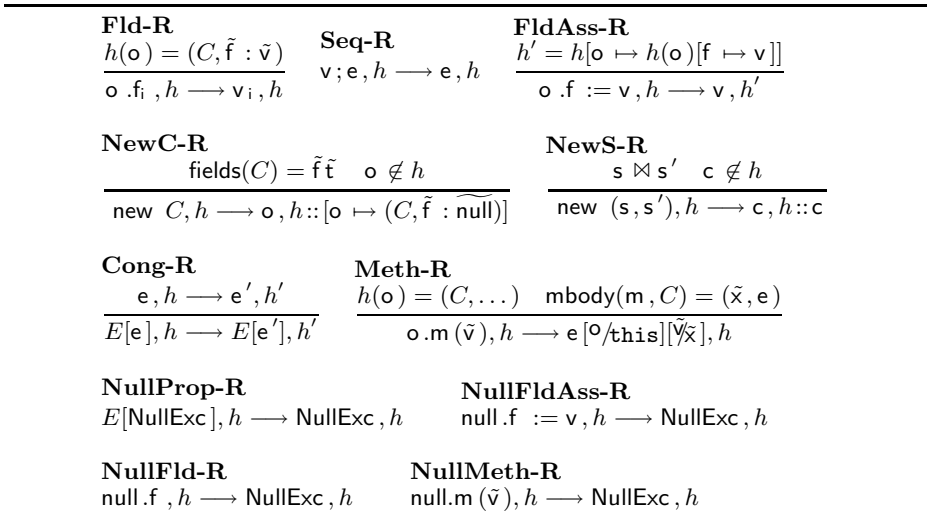

Fig. 8. Expression Reduction.

Rule **Connect-R** 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-R** gives simple session communication: value v is sent by one thread to another and it is bound to the variable x within the expression e at the receiver side.

Rule **ComSS-R** describes session delegation. One thread is ready to receive a live channel, which will be bound to the variable x within the expression e; the other 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, as shown in example 4.2 of [12].

To understand rule **ComSCaseSuccess-R** it is important to notice that in a well-typed process for all $i \in \{1, ..., n\}$ there is $k \in \{1, ..., m\}$ such that $C_i <: C'_k$ and vice versa. The sender thread checks if the exchanged object o belongs to some class of its own list of classes: if i is the smallest index in $\{1, ..., n\}$ such that o is C_i, then the sender thread continues with e_i. Similarly, if k is the smallest index in $\{1, ..., m\}$ such that o is C'_k, then the receiver thread continues with $\mathsf{e}'_k[\mathsf{o}/\mathsf{x}]$. If instead the object o does not belong to any of the classes C_i, C'_k for $i \in \{1, ..., n\}$ and $k \in \{1, ..., m\}$, then rule **ComSCaseFailure-R** is applied: both expressions return null. Notice that this choice is made at run time and

Struct

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

Spawn-R

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

Par-R

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

Str-R

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

Connect-R

$$\frac{\mathsf{c}' \notin h \qquad \mathsf{s} \bowtie \mathsf{s}'}{E_1[\mathsf{connect\ c\ s}\{\mathsf{e}_1\}] \mid E_2[\mathsf{connect\ c\ s}'\{\mathsf{e}_2\}], h \longrightarrow E_1[\mathsf{e}_1[\mathsf{c}'/\mathsf{c}]] \mid E_2[\mathsf{e}_2[\mathsf{c}'/\mathsf{c}]], h :: \mathsf{c}'}$$

ComS-R

$$E_1[\mathsf{u}\ .\mathsf{send}\ (\mathsf{v})] \mid E_2[\mathsf{u}\ .\mathsf{receive}\ (\mathsf{x})\{\mathsf{e}\}], h \longrightarrow E_1[\mathsf{null}] \mid E_2[\mathsf{e}\,[\mathsf{v}/\mathsf{x}]], h$$

ComSS-R

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

ComSCaseSuccess-R

$$\frac{\begin{array}{c} h(\mathsf{o}) = (C, \dots) \quad C <: C_i \quad \forall j < i (C \not<: C_j) \quad i \in \{1, \dots, n\} \\ C <: C_k' \quad \forall l < k (C \not<: C_l') \quad k \in \{1, \dots, m\} \end{array}}{\begin{array}{l} E_1[\mathsf{c}\ .\mathsf{sendCase}\ (\mathsf{o})\{C_1 \triangleright \mathsf{e}_1; \cdots; C_n \triangleright \mathsf{e}_n\}] \mid E_2[\mathsf{c}\ .\mathsf{receiveCase}\ (\mathsf{x})\{C_1' \triangleright \mathsf{e}_1'; \cdots; C_m' \triangleright \mathsf{e}_m'\}], h \\ \qquad \longrightarrow \quad E_1[\mathsf{e}_i] \mid E_2[\mathsf{e}_k'[\mathsf{o}/\mathsf{x}]], h \end{array}}$$

ComSCaseFailure-R

$$\frac{h(\mathsf{o}) = (C, \dots) \quad \forall i \in \{1, \dots, n\} (C \not<: C_i) \quad \forall k \in \{1, \dots, m\} (C \not<: C_k')}{\begin{array}{l} E_1[\mathsf{c}\ .\mathsf{sendCase}\ (\mathsf{o})\{C_1 \triangleright \mathsf{e}_1; \cdots; C_n \triangleright \mathsf{e}_n\}] \mid E_2[\mathsf{c}\ .\mathsf{receiveCase}\ (\mathsf{x})\{C_1' \triangleright \mathsf{e}_1'; \cdots; C_m' \triangleright \mathsf{e}_m'\}], h \\ \qquad \longrightarrow E_1[\mathsf{null}] \mid E_2[\mathsf{null}], h \end{array}}$$

ComSWhile-R

$$\begin{array}{l} E_1[\mathsf{c}\ .\mathsf{sendWhile}\ (\mathsf{e})\{C_1 \triangleright \mathsf{e}_1; \cdots; C_n \triangleright \mathsf{e}_n\}\{D_1 \triangleright \mathsf{d}_1; \cdots; D_m \triangleright \mathsf{d}_m\}] \mid \\ E_2[\mathsf{c}\ .\mathsf{receiveWhile}\ (\mathsf{x})\{C_1' \triangleright \mathsf{e}_1'; \cdots; C_{n'}' \triangleright \mathsf{e}_{n'}'\}\{D_1' \triangleright \mathsf{d}_1'; \cdots; D_{m'}' \triangleright \mathsf{d}_{m'}'\}], h \longrightarrow \\ E_1[\mathsf{c}\ .\mathsf{sendCase}\ (\mathsf{e})\{C_1 \triangleright \mathsf{e}_1''; \cdots; C_n \triangleright \mathsf{e}_n'', D_1 \triangleright \mathsf{d}_1; \cdots; D_m \triangleright \mathsf{d}_m\}] \mid \\ E_2[\mathsf{c}\ .\mathsf{receiveCase}\ (\mathsf{x})\{C_1' \triangleright \mathsf{e}_1'''; \cdots; C_{n'}' \triangleright \mathsf{e}_{n'}''', D_1' \triangleright \mathsf{d}_1'; \cdots; D_{m'}' \triangleright \mathsf{d}_{m'}'\}], h \\ \text{where } \mathsf{e}_i'' = \mathsf{e}_i; \mathsf{c}\ .\mathsf{sendWhile}\ (\mathsf{e})\{C_1 \triangleright \mathsf{e}_1; \cdots; C_n \triangleright \mathsf{e}_n\}\{D_1 \triangleright \mathsf{d}_1; \cdots; D_m \triangleright \mathsf{d}_m\} \\ \qquad \mathsf{e}_j''' = \mathsf{e}_j'; \mathsf{c}\ .\mathsf{receiveWhile}\ (\mathsf{x})\{C_1' \triangleright \mathsf{e}_1'; \cdots; C_{n'}' \triangleright \mathsf{e}_{n'}'\}\{D_1' \triangleright \mathsf{d}_1'; \cdots; D_{m'}' \triangleright \mathsf{d}_{m'}'\} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (1 \le i \le n)(1 \le j \le n') \end{array}$$

Fig. 9. Thread Reduction.

that it depends on the class of the exchanged object, which it is not known at compile time.

Rule **ComSWhile-R** describes iteration by means of case expressions. The iteration loops on the alternatives in the first pair of lists of classes and expressions, while the second pair of lists represents *default* expressions which can be possibly evaluated only once after the loop end. The typing rules assure that for all $i \in \{1,...,n\}$ there is $k \in \{1,...,n'\}$ such that $C_i <: C'_k$ and vice versa. Moreover, for all $j \in \{1,...,m\}$ there is $l \in \{1,...,m'\}$ such that $D_j <: D'_l$ and vice versa. The test is on the class of the exchanged object o: if there exists an i which is the smallest index in $\{1,...,n\}$ such that o is C_i, then the sender thread continues with e_i, and iterates. Otherwise, if there exists a j which is the smallest index in $\{1,...,m\}$ such that o is D_j, then the sender thread continues with d_j. Last, if the object o does not belong to any of the classes C_i, D_j for $i \in \{1,...,n\}$ and $j \in \{1,...,m\}$, then the sender thread returns null. Dually, if there exists a k which is the smallest index in $\{1,...,n'\}$ such that o is C'_k, then the receiver thread continues with $\mathsf{e}'_k[\mathsf{o}/\mathsf{x}]$, and iterate. Otherwise if there exists an l which is the smallest index in $\{1,...,m'\}$ such that o is D'_l, then the receiver thread continues with $\mathsf{d}'_l[\mathsf{o}/\mathsf{x}]$. Last, if the object o does not belong to any of the classes C'_k, D'_l for $k \in \{1,...,n'\}$ and $l \in \{1,...,m'\}$, then the receiver thread returns null.

5 The Type Assignment System and its Properties

5.1 Types

The full syntax of types is given in Fig. 10. It extends the syntax of [12] by allowing bounded polymorphism.

Partial session types, ranged over by π, represent sequences of communications, where ε is the empty communication, and $\pi_1.\pi_2$ consists of the communications in π_1 followed by those in π_2. The partial session types $?(X <: \mathsf{t}_1)$ and $!\mathsf{t}_2$ express respectively the reception of a value whose type is bound by type

π	$::= \varepsilon \mid \pi.\pi \mid ?(X <: \mathsf{t}) \mid !\mathsf{t} \mid ?(X <: \eta) \mid !(\eta) \mid$ $!\langle\!\langle \tilde{C}.\tilde{\pi} \rangle\!\rangle \mid ?\langle\!\langle (\tilde{X} <: \tilde{C}).\tilde{\pi} \rangle\!\rangle \mid$ $!\langle\!\langle (\tilde{C}.\tilde{\pi})^*, \tilde{C}.\tilde{\pi} \rangle\!\rangle \mid ?\langle\!\langle ((\tilde{X} <: \tilde{C}).\tilde{\pi})^*, (\tilde{X} <: \tilde{C}).\tilde{\pi} \rangle\!\rangle$	partial session type
η	$::= X \mid \pi.\mathsf{end} \mid \pi.\eta \mid !\langle\!\langle \tilde{C}.\tilde{\eta} \rangle\!\rangle \mid ?\langle\!\langle (\tilde{X} <: \tilde{C}).\tilde{\eta} \rangle\!\rangle$	ended session type
ρ	$::= \pi \mid \eta$	running session type
τ	$::= \rho \mid \updownarrow$	live session type
s	$::= \mathsf{begin}.\eta$	shared session type
θ	$::= \tau \mid \mathsf{s}$	session type
t	$::= X \mid C \mid \mathsf{s} \mid (\mathsf{s},\mathsf{s})$	standard type

Fig. 10. Syntax of Types.

$$\textbf{Class}\ \frac{C \in \mathcal{D}(\mathtt{CT})}{\vdash C : \mathtt{tp}} \qquad \textbf{Wf-Session}\ \frac{}{\vdash \mathtt{s} : \mathtt{tp}} \qquad \textbf{Pair}\ \frac{\mathtt{s} \bowtie \mathtt{s}'}{\vdash (\mathtt{s}, \mathtt{s}') : \mathtt{tp}}$$

Fig. 11. Well-formed Standard Types.

$\mathtt{t}_1$ and the sending of a value of type $\mathtt{t}_2$. Analogously, the partial session types $?(X <: \eta_1)$ and $!(\eta_2)$ represent the exchange of a live channel, and therefore of an active session, with remaining communications bound by η_1 and determined by η_2, respectively. Note that we allow bounds to be type variables; this supports more expressive session types as shown in the example of Section 2.

The *case* types $!(\!|\tilde{C}.\tilde{\pi}|\!)$ and $?(\!|(\tilde{X} <: \tilde{C}).\tilde{\pi}|\!)$ reflect in an obvious way the structure of the case expressions. The same observation applies to the *iterative* types $!(\!|(\tilde{C}.\tilde{\pi})^*, \tilde{C}.\tilde{\pi}|\!)$ and $?(\!|((\tilde{X} <: \tilde{C}).\tilde{\pi})^*, (\tilde{X} <: \tilde{C}).\tilde{\pi}|\!)$ and the while expressions.

An *ended session type*, η, is either a type variable or a partial session type concatenated either with `end` or with a case type whose branches in turn are all ended session types. It expresses a sequence of communications with its termination, *i.e.* no further communications on that channel are allowed at the end. Ended session types guarantee that a channel is consumed, *i.e.* it cannot be further used. This is essential to guarantee the uniqueness of communications in sessions, as shown in Example 5.2 of [12].

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

A *live session type* τ is either a running session type or $\updownarrow$. We use $\updownarrow$ when typing threads, to indicate the type of a channel which is being used by two threads in complementary ways.

A *shared session type*, $\mathtt{s}$, 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 describes the communication protocol. Shared session types are only used to type shared channels which behave as standard values in that they can be sent using `send` and can be stored in object fields. Live channels instead can only be sent using `sendS` and cannot be stored in object fields.

A *session type* θ is either a live session type or a shared session type.

Standard types, $\mathtt{t}$, are either type variables (X), class identifiers (C), shared session types, or pairs of shared session types which are duals (*i.e.* $(\mathtt{s}, \mathtt{s}')$). Fig. 11 defines well-formed standard types. Note that $\mathcal{D}(\mathtt{CT})$ denotes the domain of the class table `CT`, *i.e.* the set of classes declared in `CT`.

The *subtyping* judgements use subtyping environments which take into account the bounds of type variables. Subtyping environments (ranged over by Δ) are defined in Fig. 12, where $\mathcal{D}(\Delta)$ is the set of the left hand sides in the subtyping judgements of Δ.

$$\Delta := \emptyset \mid \Delta, X <: \mathtt{t} \mid \Delta, X <: \rho$$

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

SuAdd1
$$\frac{\Delta \vdash \mathsf{ok} \quad X \notin \mathcal{D}(\Delta)}{\Delta, X <: \mathtt{t} \vdash \mathsf{ok}}$$

SuAdd2
$$\frac{\Delta \vdash \mathsf{ok} \quad X \notin \mathcal{D}(\Delta)}{\Delta, X <: \rho \vdash \mathsf{ok}}$$

Fig. 12. Subtyping Environments.

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

$$\frac{\vdash (\mathtt{s}, \mathtt{s}') : \mathtt{tp}}{\Delta \ \vdash \ (\mathtt{s}, \mathtt{s}') <: \mathtt{s}} \qquad \frac{\vdash (\mathtt{s}, \mathtt{s}') : \mathtt{tp}}{\Delta \ \vdash \ (\mathtt{s}, \mathtt{s}') <: \mathtt{s}'}$$

Fig. 13. Subtyping for Standard Types.

$$\frac{\Delta \ \vdash \ \mathtt{t} <: \mathtt{t}'}{\Delta \ \vdash \ !\mathtt{t} <: !\mathtt{t}'} \qquad \frac{\Delta \ \vdash \ \mathtt{t} <: \mathtt{t}'}{\Delta \ \vdash \ ?(X <: \mathtt{t}') <: ?(X <: \mathtt{t})}$$

$$\frac{\Delta \ \vdash \ \eta <: \eta'}{\Delta \ \vdash \ !(\eta) <: !(\eta')} \qquad \frac{\Delta \ \vdash \ \eta <: \eta'}{\Delta \ \vdash \ ?(X <: \eta') <: ?(X <: \eta)}$$

$$\frac{}{\Delta, X <: \eta \ \vdash \ X <: \eta} \qquad \frac{\Delta \ \vdash \ \rho_i <: \rho_i' \quad i \in \{1, ..., n\}}{\Delta \ \vdash \ !(\!|C_1.\rho_1, \ldots, C_n.\rho_n|\!) <: !(\!|C_1.\rho_1', \ldots, C_n.\rho_n'|\!)}$$

$$\frac{\Delta, X_i <: C_i \ \vdash \ \rho_i <: \rho_i' \quad i \in \{1, ..., n\}}{\Delta \ \vdash \ ?(\!|(X_1 <: C_1).\rho_1, \ldots, (X_n <: C_n).\rho_n|\!) <: ?(\!|(X_1 <: C_1).\rho_1', \ldots, (X_n <: C_n).\rho_n'|\!)}$$

$$\frac{\Delta \ \vdash \ \pi_i <: \pi_i' \quad i \in \{1, ..., n+m\}}{\begin{array}{l}\Delta \ \vdash \ !(\!|(C_1.\pi_1, \ldots, C_n.\pi_n)^*, D_1.\pi_{n+1}, \ldots, D_m.\pi_{n+m}|\!) \\ \quad <: !(\!|(C_1.\pi_1', \ldots, C_n.\pi_n')^*, D_1.\pi_{n+1}', \ldots, D_m.\pi_{n+m}'|\!)\end{array}}$$

$$\frac{\Delta, X_i <: C_i \ \vdash \ \pi_i <: \pi_i' \quad i \in \{1, ..., n\} \quad \Delta, Y_j <: D_j \ \vdash \ \pi_{n+j} <: \pi_{n+j}' \quad j \in \{1, ..., m\}}{\begin{array}{l}\Delta \ \vdash \ ?(\!|((X_1 <: C_1).\pi_1, \ldots, (X_n <: C_n).\pi_n)^*, (Y_1 <: D_1).\pi_{n+1}, \ldots, (Y_m <: D_m).\pi_{n+m}|\!) \\ \quad <: ?(\!|((X_1 <: C_1).\pi_1', \ldots, (X_n <: C_n).\pi_n')^*, (Y_1 <: D_1).\pi_{n+1}', \ldots, (Y_m <: D_m).\pi_{n+m}'|\!)\end{array}}$$

$$\frac{\Delta \ \vdash \ \pi <: \pi'}{\Delta \ \vdash \ \pi.\mathsf{end} <: \pi'.\mathsf{end}} \qquad \frac{\Delta \ \vdash \ \pi <: \pi' \qquad \Delta \ \vdash \ \rho <: \rho'}{\Delta \ \vdash \ \pi.\rho <: \pi'.\rho'}$$

Fig. 14. Subtyping for Running Session Types.

$$\frac{\rho \bowtie \rho'}{\mathsf{begin}.\rho \bowtie \mathsf{begin}.\rho'} \qquad \frac{}{\mathsf{end} \bowtie \mathsf{end}} \qquad \frac{\rho \bowtie \rho'}{\rho' \bowtie \rho} \qquad \frac{\rho \bowtie \rho'[\eta/X]}{!(\eta).\rho \bowtie ?(X <: \eta').\rho'} \qquad \frac{\rho \bowtie \rho'[\mathtt{t}/X]}{!\mathtt{t}\,.\rho \bowtie ?(X <: \mathtt{t}\,').\rho'}$$

$$\frac{\begin{array}{c}\&_{i\in\{1,\ldots,n\},j\in\{1,\ldots,m\}}\ (\rho_i \bowtie \rho'_j[C_i \curlyvee C'_j/X_j]) \\ \forall i \in \{1,...,n\}\exists j \in \{1,...,m\}.\ C_i <: C'_j \qquad \forall j \in \{1,...,m\}\exists i \in \{1,...,n\}.\ C'_j <: C_i\end{array}}{!(\!|C_1.\rho_1, \ldots, C_n.\rho_n|\!) \bowtie ?(\!|(X_1 <: C'_1).\rho'_1, \ldots, (X_m <: C'_m).\rho'_m|\!)}$$

$$\frac{\begin{array}{c}\&_{i\in\{1,\ldots,n\},k\in\{1,\ldots,n'\}}\ (\pi_i \bowtie \pi'_k[C_i \curlyvee C'_k/X_k])\& \\ \&_{j\in\{1,\ldots,m\},l\in\{1,\ldots,m'\}}\ (\pi_{n+j} \bowtie \pi'_{n'+l}[D_j \curlyvee D'_l/Y_l]) \\ \forall i \in \{1,...,n\}\exists k \in \{1,...,n'\}.\ C_i <: C'_k \qquad \forall k \in \{1,...,n'\}\exists i \in \{1,...,n\}.\ C'_k <: C_i \\ \forall j \in \{1,...,m\}\exists l \in \{1,...,m'\}.\ D_j <: D'_l \qquad \forall l \in \{1,...,m'\}\exists j \in \{1,...,m\}.\ D'_l <: D_j\end{array}}{\begin{array}{c}!(\!|(C_1.\pi_1, \ldots, C_n.\pi_n)^*, D_1.\pi_{n+1}, \ldots, D_m.\pi_{n+m}|\!) \bowtie \\ ?(\!|((X_1 <: C'_1).\pi'_1, \ldots, (X_{n'} <: C'_{n'}).\pi'_{n'})^*(Y_1 <: D'_1).\pi'_{n'+1}, \ldots, (Y_{m'} <: D'_{m'}).\pi'_{n'+m'}|\!)\end{array}}$$

Fig. 15. Duality Relation.

The subtyping judgement for standard types has the shape:

$$\Delta \ \vdash \ \mathtt{t} <: \mathtt{t}'$$

and it holds if it can be derived from the axioms of Fig. 13 plus the reflexive and transitive rules. As in [26], we assume that the subclassing is acyclic.

The subtyping judgement for running session types has the shape:

$$\Delta \ \vdash \ \rho <: \rho'$$

and it holds if it can be derived from the axioms of Fig. 14 plus the reflexive and transitive rules. It is worthwhile to notice that, in contrast with [16], our session subtyping is covariant for outputs and contravariant for inputs with respect to the standard subtyping of the communicated values. The motivation for this is that we expect all channels whose type is a subtype of $!\mathtt{t}$ to be able to communicate with channels whose type is $?(X <: \mathtt{t}\,)$ and similarly for the other types. This requires the given rules for output. Similar reasons justify the rules for inputs.

In order to guarantee protocol soundness it is crucial to introduce a duality relation between shared session types which ensures that two sessions agree with each other with respect to the order in which data are communicated and with respect to the types of the communicated data. The introduction of bounded polymorphism allows to relate more than one session type to another session type, in contrast to the systems of [1, 12, 16, 20, 30], where each session type has a unique dual. The definition of this relation (denoted by $\bowtie$) is given in Fig. 15, where we use $\mathtt{t} <: \mathtt{t}'$ as shorthand for $\emptyset \ \vdash \ \mathtt{t} <: \mathtt{t}'$ and similarly for $\rho <: \rho'$.

Two case types $!(\!|C_1.\rho_1, \ldots, C_n.\rho_n|\!)$ and $?(\!|(X_1 <: C'_1).\rho'_1, \ldots, (X_m <: C'_m).\rho'_m|\!)$ are dual if for any arbitrary class C either $C <: C_i$ and $C <: C'_j$ for some

$i \in \{1,\ldots,n\}$ and $j \in \{1,\ldots,m\}$, or $C \not<: C_i$ and $C \not<: C'_j$ for all $i \in \{1,\ldots,n\}$ and $j \in \{1,\ldots,m\}$. This condition is necessary in order to ensure applicability of one of the two reduction rules **ComSCaseSuccess-R** and **ComSCaseFailure-R**. We ensure this condition by requiring that for all $i \in \{1,...,n\}$ there exists a $j \in \{1,...,m\}$ such that $C_i <: C'_j$ and for all $j \in \{1,...,m\}$ there exists a $i \in \{1,...,n\}$ such that $C'_j <: C_i$. Moreover, if there is a class which is a subclass of both C_i and C'_j (i.e. C_i and C'_j are comparable), then we need to guarantee that the two threads will continue the communication in a coherent way.[6] This means that ρ_i and ρ'_j must agree as specified below. Instead, if C_i and C'_j are incomparable, then ρ_i and ρ'_j can be unrelated. In order to express the above conditions we define the "minimum" of two classes as follows:

$$C \curlyvee C' = \begin{cases} C & \text{if } C <: C', \\ C' & \text{if } C' <: C, \\ \bot & \text{otherwise.} \end{cases}$$

and we require $\rho_i \bowtie \rho'_j[{}^{C_i \curlyvee C'_j}/_{X_j}]$ for all $i \in \{1,...,n\}$ and all $j \in \{1,...,m\}$, with the convention $\rho_i \bowtie \rho'_j[{}^{\bot}/_{X_j}] = \mathsf{true}$.

The duality of iterative types can be explained similarly, taking into account that we need to ensure that either both threads choose to iterate, or both threads choose default expressions, or both threads choose null.

It is easy to verify by induction on the definition of $\bowtie$ that subtyping preserves duality, *i.e.*:

Lemma 5.1. *If $\rho \bowtie \rho'$ and $\rho'' <: \rho'$, then $\rho \bowtie \rho''$.*

5.2 Typing Rules

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

$$\Delta; \Gamma; \Sigma \vdash \mathsf{e} : \mathsf{t} \qquad\qquad \Delta; \Gamma; \Sigma \vdash P : \mathsf{thread}$$

where the *standard environment* Γ associates standard types to this, parameters, objects, and shared channels, while the *session environment* Σ contains only judgements for channel names and variables. Fig. 16 defines well-formedness of standard and session environments, where the domain of an environment is defined as usual and denoted by $\mathcal{D}()$.

The main differences with the typing rules of [12] are the addition of bounded polymorphism and the deletion of hot sets, which in [12] were used to guarantee progress, a property we will not consider here. As we already discussed in the introduction, we could add hot sets to get progress without problems.

In Fig. 17, Fig. 18 and Fig. 20 we give the typing rules for expressions and threads using the lookup functions defined in Fig. 7. In the typing rules for

[6] Taking into account the order in which the classes appear we could avoid to check some pairs of i, j. For simplicity we do not consider this refinement.

Standard Environments, and Well-formed Standard Environments

$$\Gamma \;::=\; \emptyset \mid \Gamma, \mathtt{x} : \mathtt{t} \mid \Gamma, \mathtt{this} : C \mid \Gamma, \mathtt{o} : C \mid \Gamma, \mathtt{c} : \mathtt{s} \mid \Gamma, \mathtt{c} : (\mathtt{s}, \mathtt{s}')$$

Emp

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

EVar

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

EOid

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

Ethis

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

ECha1

$$\frac{\Gamma \vdash \mathsf{ok} \quad \vdash \mathtt{s} : \mathtt{tp} \quad \mathtt{c} \notin \mathcal{D}(\Gamma)}{\Gamma, \mathtt{c} : \mathtt{s} \vdash \mathsf{ok}}$$

ECha2

$$\frac{\Gamma \vdash \mathsf{ok} \quad \vdash (\mathtt{s}, \mathtt{s}') : \mathtt{tp} \quad \mathtt{c} \notin \mathcal{D}(\Gamma)}{\Gamma, \mathtt{c} : (\mathtt{s}, \mathtt{s}') \vdash \mathsf{ok}}$$

Session Environments, and Well-formed Session Environments

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

SEmp

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

SeAdd

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

Fig. 16. Standard and Session Environments.

expressions the session environments of the conclusions are obtained from those of the premises and possibly other session environments using the *concatenation* operator, $\circ$, defined below. We consider different cases for the concatenation of session types since we want to avoid to have redundant ε. As usual, $\perp$ stands for undefined.

$$- \;\; \rho \circ \rho' \;=\; \begin{cases} \rho & \text{if } \; \rho' = \varepsilon \\ \rho' & \text{if } \; \rho = \varepsilon \\ \rho.\mathsf{end} & \text{if } \; \rho' = \varepsilon.\mathsf{end} \text{ and } \rho \text{ is a partial session type} \\ \rho.\rho' & \text{if } \; \rho \text{ is a partial session type and} \\ & \rho' \text{ is a running session type} \\ \perp & \text{otherwise.} \end{cases}$$

$$- \;\; \Sigma \setminus \Sigma' \;=\; \{\mathtt{u} : \Sigma(\mathtt{u}) \mid \mathtt{u} \in \mathcal{D}(\Sigma) \setminus \mathcal{D}(\Sigma')\}$$

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

The concatenation of two channel types ρ and ρ' is the unique channel type (if it exists) which prescribes all the communications of ρ followed by all those of ρ'. The concatenation only exists if ρ is a partial session type, and ρ' is a running session type. The extension to session environments is straightforward. The typing rules concatenate the session environments to take into account the

Typing Rules for Values

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

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

Chan
$$\frac{\Gamma, \mathsf{c} : \mathsf{t} \vdash \mathsf{ok}}{\Delta; \Gamma, \mathsf{c} : \mathsf{t}\,; \emptyset \vdash \mathsf{c} : \mathsf{t}}$$

Typing Rules for Standard Expressions

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

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

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

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

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

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

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

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

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

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

MethB
$$\frac{\begin{array}{c}\Delta; \Gamma; \Sigma_0 \vdash \mathsf{e} : C \quad \Delta; \Gamma; \Sigma_i \vdash \mathsf{e}_i : \mathsf{t}_i \quad i \in \{1 \ldots n\} \\ \mathsf{mtype}(\mathsf{m}, C) = X, \mathsf{t}_2, \ldots, \mathsf{t}_n, \rho_1, \ldots, \rho_m \to X <: \mathsf{t} \qquad \Delta \vdash \mathsf{t}_1 <: \mathsf{t}\end{array}}{\Delta; \Gamma; \Sigma_0 \circ \Sigma_1 \ldots \circ \Sigma_n \circ \{\mathsf{u}_1 : \rho_1, \ldots, \mathsf{u}_m : \rho_m\} \vdash \mathsf{e}.\mathsf{m}\,(\mathsf{e}_1, \ldots, \mathsf{e}_n, \mathsf{u}_1, \ldots, \mathsf{u}_m) : \mathsf{t}_1}$$

Fig. 17. Typing Rules for Expressions I.

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

Rule **Spawn** requires that all sessions used by the spawned thread are finally consumed, *i.e.* they are all ended session types. This is necessary in order to avoid configurations in which more than two threads are ready to communicate on the same live channel. To guarantee the consumption we define:

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

Typing Rules for Communication Expressions

Conn

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

Send

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

Receive

$$\frac{\Delta, X <: \mathsf{t}\,; \Gamma, \mathsf{x} : X; \Sigma \vdash \mathsf{e} : \mathsf{t}' \qquad X \notin \Gamma \cup \Delta \cup \Sigma \setminus \mathsf{u}}{\Delta; \Gamma; \{\mathsf{u} : ?(X <: \mathsf{t})\} \circ \Sigma \vdash \mathsf{u}.\mathsf{receive}(\mathsf{x})\{\mathsf{e}\} : \mathsf{t}'}$$

SendS

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

ReceiveS

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

SendCase

$$\frac{\Delta; \Gamma; \Sigma_0 \vdash \mathsf{e} : C \quad \Delta; \Gamma; \Sigma, \mathsf{u} : \rho_i \vdash \mathsf{e}_i : \mathsf{t} \quad C_i <: C \quad i \in \{1, \ldots, n\}}{\Delta; \Gamma; \Sigma_0 \circ \Sigma, \mathsf{u} : !(\!|C_1.\rho_1, \ldots, C_n.\rho_n|\!) \vdash \mathsf{u}.\mathsf{sendCase}(\mathsf{e})\{C_1 \triangleright \mathsf{e}_1; \cdots ; C_n \triangleright \mathsf{e}_n\} : \mathsf{t}}$$

ReceiveCase

$$\frac{\Delta, X_i <: C_i; \Gamma, \mathsf{x} : X_i; \Sigma, \mathsf{u} : \rho_i \vdash \mathsf{e}_i : \mathsf{t} \quad i \in \{1, \ldots, n\}}{\Delta; \Gamma; \Sigma, \mathsf{u} : ?(\!|(X_1 <: C_1).\rho_1, \ldots, (X_n <: C_n).\rho_n|\!) \vdash \mathsf{u}.\mathsf{receiveCase}(\mathsf{x})\{C_1 \triangleright \mathsf{e}_1; \cdots ; C_n \triangleright \mathsf{e}_n\} : \mathsf{t}}$$

SendWhile

$$\frac{\begin{array}{c}\Delta; \Gamma; \emptyset \vdash \mathsf{e} : C \quad \Delta; \Gamma; \{\mathsf{u} : \pi_i\} \vdash \mathsf{e}_i : \mathsf{t} \quad C_i <: C \quad i \in \{1, \ldots, n\} \\ \Delta; \Gamma; \{\mathsf{u} : \pi_{n+j}\} \vdash \mathsf{d}_j : \mathsf{t} \quad D_j <: C \quad j \in \{1, \ldots, m\}\end{array}}{\begin{array}{l}\Delta; \Gamma; \{\mathsf{u} : !(\!|(C_1.\pi_1, \ldots, C_n.\pi_n)^*, D_1.\pi_{n+1}, \ldots, D_m.\pi_{n+m}|\!)\} \\ \vdash \mathsf{u}.\mathsf{sendWhile}(\mathsf{e})\{C_1 \triangleright \mathsf{e}_1; \cdots ; C_n \triangleright \mathsf{e}_n\}\{D_1 \triangleright \mathsf{d}_1; \cdots ; D_m \triangleright \mathsf{d}_m\} : \mathsf{t}\end{array}}$$

ReceiveWhile

$$\frac{\begin{array}{c}\Delta, X_i <: C_i; \Gamma, \mathsf{x} : X_i; \{\mathsf{u} : \pi_i\} \vdash \mathsf{e}_i : \mathsf{t} \quad i \in \{1, \ldots, n\} \\ \Delta, Y_j <: D_j; \Gamma, \mathsf{x} : Y_j; \{\mathsf{u} : \pi_{n+j}\} \vdash \mathsf{d}_j : \mathsf{t} \quad j \in \{1, \ldots, m\}\end{array}}{\begin{array}{l}\Delta; \Gamma; \{\mathsf{u} : ?(\!|((X_1 <: C_1).\pi_1, \ldots, (X_n <: C_n).\pi_n)^*, (Y_1 <: D_1).\pi_{n+1}, \ldots, (Y_m <: D_m).\pi_{n+m}|\!)\} \\ \vdash \mathsf{u}.\mathsf{receiveWhile}(\mathsf{x})\{C_1 \triangleright \mathsf{e}_1; \cdots ; C_n \triangleright \mathsf{e}_n\}\{D_1 \triangleright \mathsf{d}_1; \cdots ; D_m \triangleright \mathsf{d}_m\} : \mathsf{t}\end{array}}$$

Non-structural Typing Rules for Expressions

Sub

$$\frac{\Delta; \Gamma; \Sigma \vdash \mathsf{e} : \mathsf{t} \quad \Delta \vdash \mathsf{t} <: \mathsf{t}'}{\Delta; \Gamma; \Sigma \vdash \mathsf{e} : \mathsf{t}'}$$

WeakES

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

WeakE

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

Fig. 18. Typing Rules for Expressions II.

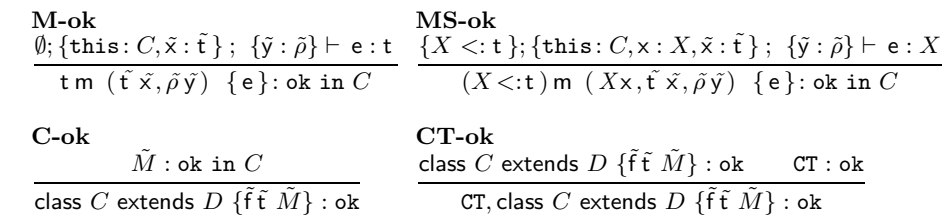

Fig. 19. Well-formed Class Tables.

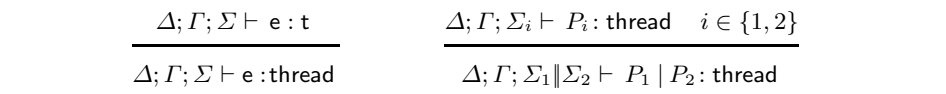

Fig. 20. Typing Rules for Threads.

Rules **Meth** and **MethB** retrieve the type of the method $\texttt{m}$ from the class table using the auxiliary function $\texttt{mtype}(\texttt{m}, C)$ defined in Fig. 7. The session environments of the premises are concatenated with $\{\texttt{u}_1 : \rho_1, \ldots, \texttt{u}_m : \rho_m\}$, which represents the communication protocols of the channels $\texttt{u}_1, \ldots, \texttt{u}_m$ during the execution of the method body. The difference between the two rules is the type of the return value, which is fixed in rule **Meth** and instead parametrized on the type of the first argument in rule **MethB**.

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

Rules **SendCase** and **ReceiveCase** put together the types of the different alternatives in the expected way. Notice that, in a specific case expression, all ρ_i for $i \in \{1, ..., n\}$ are either partial session types or ended session types – this is guaranteed by the syntax of case session types. Similarly for rules **SendWhile** and **ReceiveWhile**.

Rule **WeakES**, where **ES** stands for empty session, is necessary to type a branch of a case expression where the channel which is the subject of the conditional is not used. Rule **WeakE**, where **E** stands for $\texttt{end}$, allows us to obtain ended session types as predicates of session environments in order to apply rules **Conn**, **Spawn** and **ReceiveS**.

Fig. 19 defines well-formed class tables. Rules **M-ok** and **MS-ok** type-check the method bodies with respect to a class C taking as environments the asso-

ciation between formal parameters and their types and the association between this and C. These rules differ in the return type of the method.

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

$$\theta \| \theta' \;=\; \begin{cases} \updownarrow & \text{if} \quad \theta \bowtie \theta' \\ \bot & \text{otherwise.} \end{cases}$$

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

Note that $\updownarrow \| \theta = \theta \| \updownarrow = \bot$.
Using the operator $\|$ the typing rules for processes are straightforward (see Fig. 20). Rule **Start** promotes an expression to the thread level; and rule **Par** types a composition of threads if the composition of their session environments is defined.

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

The typing rules of MOOSE$_{<:}$ are not syntax directed, because of the non structural typing rules and also because of the use of the "$\circ$" and "$\|$" operators in composing session environments. Nevertheless, we can design a type inference algorithm for MOOSE$_{<:}$ as we did for MOOSE [12].

5.3 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 $\Gamma; \Sigma \vdash h$. The judgement is defined in Fig. 21. The judgement $\Gamma; h \vdash \mathsf{v} : \mathsf{t}$ guarantees that the value v has type t. The judgement $\Gamma; h \vdash \mathsf{o}$ guarantees that the object o is well-formed, *i.e.* that its fields contain values according to the declared field types in C, the class of that object. The judgement $\Gamma; \Sigma \vdash h$ guarantees that the heap is well-formed for Γ and Σ, *i.e.* that all objects are well-formed, all o in the domain of Γ denote in the h objects of the class given to them in Γ, all channels in the domain of Σ are channels in h, and no channel occurs both in Γ and Σ.
We define $\Delta; \Gamma; \Sigma \vdash P; h$ as a shorthand for $\Delta; \Gamma; \Sigma \vdash P : \mathsf{thread}$ and $\Gamma; \Sigma \vdash h$.

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

Standard ingredients of Subject Reduction proofs are Generation Lemmas. The Generation Lemmas in this work are somewhat unusual, because, due to the non-structural rules, when an expression is typed, we only can deduce *some* information about the session environment used in the typing. For example,

HNull
$$\frac{C \in \mathcal{D}(\mathtt{CT})}{\Gamma; h \vdash \mathsf{null} : C}$$

HObj
$$\frac{h(\mathsf{o}) = (\Gamma(\mathsf{o}), ...) \quad \emptyset \vdash \Gamma(\mathsf{o}) <: C}{\Gamma; h \vdash \mathsf{o} : C}$$

HCha
$$\frac{\emptyset \vdash \Gamma(\mathsf{c}) <: \mathsf{t}}{\Gamma; h \vdash \mathsf{c} : \mathsf{t}}$$

WfObj
$$\frac{h(\mathsf{o}) = (C, \tilde{\mathsf{f}} : \tilde{\mathsf{v}}) \qquad \mathsf{fields}(C) = \tilde{\mathsf{f}}\,\tilde{\mathsf{t}} \quad \Gamma; h \vdash \mathsf{v}_i : \mathsf{t}_i}{\Gamma; h \vdash \mathsf{o}}$$

WfHeap
$$\frac{\forall \mathsf{o} \in \mathcal{D}(h) : \ \Gamma; h \vdash \mathsf{o} \qquad \forall \mathsf{o} \in \mathcal{D}(\Gamma) : \ \Gamma; h \vdash \mathsf{o} : \Gamma(\mathsf{o}) \qquad \forall \mathsf{c} \in \mathcal{D}(\Sigma) : \ \mathsf{c} \in h \qquad \mathcal{D}(\Gamma) \cap \mathcal{D}(\Sigma) = \emptyset}{\Gamma; \Sigma \vdash h}$$

Fig. 21. Types of Runtime Entities, and Well-formed Heaps

$\Gamma; \Sigma \vdash \mathsf{x} : \mathsf{t}$ does *not* imply that $\Sigma = \emptyset$; instead, it implies that $\mathcal{R}(\Sigma) \subseteq \{\varepsilon, \varepsilon.\mathsf{end}\}$, where $\mathcal{R}(\Sigma)$ is the range of Σ.

In order to express the Generation Lemmas, we define the partial order $\preceq$ among session environments, which basically reflects the differences introduced through the application of non-structural rules.

Definition 5.2 (Weakening Order $\preceq$). *$\Sigma \preceq \Sigma'$ is the smallest partial order such that:*

- $\Sigma \preceq \Sigma, \mathsf{u} : \varepsilon$ *if* $\mathsf{u} \notin \mathcal{D}(\Sigma)$,
- $\Sigma, \mathsf{u} : \pi \preceq \Sigma, \mathsf{u} : \pi.\mathsf{end}$,
- $\Sigma, \mathsf{u} : \varepsilon.\mathsf{end} \preceq \Sigma, \mathsf{u} : \updownarrow$.

The following lemma states that the ordering relation $\preceq$ preserves the types of expressions and threads, and its proof is easy using the non-structural typing rules and Generation Lemmas.

Lemma 5.3. *1. If $\Sigma \preceq \Sigma'$ and $\Delta; \Gamma; \Sigma \vdash \mathsf{e} : \mathsf{t}$, then $\Delta; \Gamma; \Sigma' \vdash \mathsf{e} : \mathsf{t}$.*
2. If $\Sigma \preceq \Sigma'$ and $\Delta; \Gamma; \Sigma \vdash P : \mathsf{thread}$, then $\Delta; \Gamma; \Sigma' \vdash P : \mathsf{thread}$.

Using the above lemma and the Generation Lemmas one can show that the structural equivalence preserves typing.

Lemma 5.4 (Preservation of Typing under Structural Equivalence). *If $\Delta; \Gamma; \Sigma \vdash P : \mathsf{thread}$ and $P \equiv P'$, then $\Delta; \Gamma; \Sigma \vdash P' : \mathsf{thread}$.*

Lemma 5.5 states that the typing of $E[\mathsf{e}]$ can be broken down into the typing of e, and the typing of $E[\mathsf{x}]$. Furthermore, Σ, the environment used to type $E[\mathsf{x}]$, can be broken down into two environments, $\Sigma = \Sigma_1 \circ \Sigma_2$, where Σ_1 is used to type e, and Σ_2 is used to type $E[\mathsf{x}]$.

Lemma 5.5 (Subderivations). *If* $\Delta; \Gamma; \Sigma \vdash E[\mathtt{e}] : \mathtt{t}$ *, then there exist* Σ_1, Σ_2, $\mathtt{t}'$, $\mathtt{x}$ *fresh in* E,Γ*, such that* $\Sigma = \Sigma_1 \circ \Sigma_2$*, and* $\Delta; \Gamma; \Sigma_1 \vdash \mathtt{e} : \mathtt{t}'$*, and* $\Delta; \Gamma, \mathtt{x} : \mathtt{t}'; \Sigma_2 \vdash E[\mathtt{x}] : \mathtt{t}$ *.*

On the other hand, Lemma 5.6 allows the combination of the typing of $E[\mathtt{x}]$ and the typing of $\mathtt{e}$, provided that the contexts Σ_1 and Σ_2 used for the two typing can be composed through $\circ$, and that the type of $\mathtt{e}$ is the same as the one of $\mathtt{x}$ in the first typing.

Lemma 5.6 (Context Substitution). *If* $\Delta; \Gamma; \Sigma_1 \vdash \mathtt{e} : \mathtt{t}'$*, and* $\Delta; \Gamma, \mathtt{x} : \mathtt{t}'; \Sigma_2 \vdash E[\mathtt{x}] : \mathtt{t}$ *, and* $\Sigma_1 \circ \Sigma_2$ *is defined, then* $\Delta; \Gamma; \Sigma_1 \circ \Sigma_2 \vdash E[\mathtt{e}] : \mathtt{t}$ *.*

We can now state the Subject Reduction theorem:

Theorem 5.7 (Subject Reduction).

1. $\Delta; \Gamma; \Sigma \vdash \mathtt{e} : \mathtt{t}$ *, and* $\Gamma; \Sigma \vdash h$*, and* $\mathtt{e}, h \longrightarrow \mathtt{e}', h'$ *imply* $\Delta; \Gamma'; \Sigma \vdash \mathtt{e}' : \mathtt{t}$ *, and* $\Gamma'; \Sigma \vdash h'$*, with* $\Gamma \subseteq \Gamma'$*.*
2. $\Delta; \Gamma; \Sigma \vdash P; h$ *and* $P, h \longrightarrow P', h'$ *imply* $\Delta; \Gamma'; \Sigma' \vdash P; h'$ *with* $\Gamma \subseteq \Gamma'$ *and* $\Sigma \subseteq \Sigma'$*.*

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

6 Conclusion and Further Work

In this paper we presented the language MOOSE$_{<:}$, through which we studied the addition of bounded polymorphism to an object-oriented language with session types, and the selection of communication based on the class of objects being sent/received. Through a case study we demonstrated how these new features add flexibility and expressibility. Because of covariance/contravariance of output/input, and the need to match cases, MOOSE$_{<:}$ subtype and duality relations are more interesting than in most work on session types.

In terms of the type systems, there are several directions we plan to explore: We want to extend MOOSE$_{<:}$ to incorporate generic classes in the style of GJ [22], and allow method generic parameters to appear within class instantiations in the argument and result types. We would also like to add union types [26] so as to require the class of objects sent in case expressions to be one of the expected classes and thus guarantee applicability of rule **ComSCaseSuccess-R**. Finally, more refined notions of polymorphism, such as F-bounded polymorphism [4] and match-bounded polymorphism [3], deserve investigations in the framework of session types for object-oriented languages.

In terms of the language design, we believe that selection of communication based on the class of objects being sent/received, rather than on the basis of a

label or a boolean, is the right design choice in the context of an object oriented language. However, so far, we have only considered what is a natural *extension* of an object oriented language. A more interesting question to tackle in the future, is an *amalgamation* of object orientation with session primitives.

References

1. E. Bonelli, A. Compagnoni, and E. Gunter. Correspondence Assertions for Process Synchronization in Concurrent Communications. *Journal of Functional Programming*, 15(2):219–248, 2005.
2. G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In C. Chambers, editor, *OOPSLA '98*, pages 183–200. ACM Press, 1998.
3. K. B. Bruce, A. Schuett, R. van Gent, and A. Fiech. PolyTOIL: A Type-safe Polymorphic Object-oriented Language. *ACM TOPLAS*, 25(2):225–290, 2003.
4. P. Canning, W. Cook, W. Hill, W. Olthoff, and J. C. Mitchell. F-bounded Polymorphism for Object-Oriented Programming. In *FPCA'89*, pages 273–280. ACM Press, 1989.
5. M. Carbone, K. Honda, and N. Yoshida. A Calculus of Global Interaction Based on Session Types. In *DCM'06*, ENTCS. Elsevier, 2007. To appear.
6. M. Carbone, K. Honda, and N. Yoshida. Structured Communication-Centred Programming for Web Services. In R. De Nicola, editor, *ESOP'07*, volume 4421 of *LNCS*, pages 2–17. Springer-Verlag, 2007.
7. L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys*, 17(4):471–522, 1985.
8. G. Castagna, R. De Nicola, and D. Varacca. Semantic Subtyping for the π-calculus. In P. Panangaden, editor, *LICS '05*, pages 92–101. IEEE Computer Society Press, 2005.
9. G. Castagna and A. Frisch. A Gentle Introduction to Semantic Subtyping. In P. Barahona and A. Felty, editors, *PPDP'05*, pages 198–208. ACM Press, 2005. Joint ICALP-PPDP keynote talk.
10. M. Coppo, M. Dezani-Ciancaglini, and N. Yoshida. Asynchronous Session Types and Progress for Object-Oriented Languages. In *FMOODS'07*, LNCS. Springer-Verlag, 2007. To appear.
11. R. Corin, P.-M. Denielou, C. Fournet, K. Bhargavan, and J. Leifer. Secure Implementations for Typed Session Abstractions. In *CFS'07*. IEEE-CS Press, 2007. to appear.
12. M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopoulou. Session Types for Object-Oriented Languages. In D. Thomas, editor, *ECOOP'06*, volume 4067 of *LNCS*, pages 328–352. Springer-Verlag, 2006.
13. M. Dezani-Ciancaglini, N. Yoshida, A. Ahern, and S. Drossopoulou. A Distributed Object Oriented Language with Session Types. In R. D. Nicola and D. Sangiorgi, editors, *TGC'05*, volume 3705 of *LNCS*, pages 299–318. Springer-Verlag, 2005.
14. A. Frisch, G. Castagna, and V. Benzaken. Semantic Subtyping. In G. Plotkin, editor, *LICS'02*, pages 137–146. IEEE Computer Society Press, 2002.
15. P. Garralda, A. Compagnoni, and M. Dezani-Ciancaglini. BASS: Boxed Ambients with Safe Sessions. In M. Maher, editor, *PPDP'06*, pages 61–72. ACM Press, 2006.
16. S. Gay and M. Hole. Subtyping for Session Types in the Pi-Calculus. *Acta Informatica*, 42(2/3):191–225, 2005.

17. S. Gay, V. T. Vasconcelos, and A. Ravara. Session Types for Inter-Process Communication. TR 2003–133, Department of Computing, University of Glasgow, 2003.
18. S. J. Gay. Bounded Polymorphism in Session Types. *Mathematical Structures in Computer Science*, 2007. To appear.
19. K. Honda. Types for Dyadic Interaction. In E. Best, editor, *CONCUR'93*, volume 715 of *LNCS*, pages 509–523. Springer-Verlag, 1993.
20. K. Honda, V. T. Vasconcelos, and M. Kubo. Language Primitives and Type Disciplines for Structured Communication-based Programming. In C. Hankin, editor, *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer-Verlag, 1998.
21. K. Honda, N. Yoshida, and M. Carbone. Web Services, Mobile Processes and Types. *EATCS Bulletin*, 2:160–185, 2007.
22. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a Minimal Core Calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.
23. R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, Parts I and II. *Information and Computation*, 100(1), 1992.
24. D. Mostrous and N. Yoshida. Two session typing systems for higher-order mobile processes. In *TLCA'07*, LNCS. Springer-Verlag, 2007.
25. M. Odersky and P. Wadler. Pizza into Java: Translating Theory into Practice. In M. Felleisen, editor, *POPL'97*, pages 146–159. ACM Press, 1997.
26. B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
27. S. Sparkes. Conversation with Steve Ross-Talbot. *ACM Queue*, 4(2):14–23, 2006.
28. K. Takeuchi, K. Honda, and M. Kubo. An Interaction-based Language and its Typing System. In C. Halatsis, D. Maritsas, G. Philokyprou, and S. Theodoridis, editors, *PARLE'94*, volume 817 of *LNCS*, pages 398–413. Springer-Verlag, 1994.
29. A. Vallecillo, V. T. Vasconcelos, and A. Ravara. Typing the Behavior of Objects and Components using Session Types. In A. Brogi and J.-M. Jacquet, editors, *FOCLASA'02*, volume 68(3) of *ENTCS*, pages 439–456. Elsevier, 2002.
30. V. T. Vasconcelos, S. Gay, and A. Ravara. Typechecking a Multithreaded Functional Language with Session Types. *Theorical Computer Science*, 368(1-2):64–87, 2006.
31. D. Walker. Substructural Type Systems. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, pages 3–44. MIT Press, 2005.
32. Web Services Choreography Working Group. Web Services Choreography Description Language. http://www.w3.org/2002/ws/chor/.

A Proof of Subject Reduction

A.1 Generation Lemmas

Lemma A.1 (Generation for Standard Expressions).

1. $\Delta;\Gamma;\Sigma \vdash \mathsf{x} : \mathsf{t}$ *implies* $\emptyset \preceq \Sigma$ *and* $\mathsf{x} : \mathsf{t}' \in \Gamma$ *for some* t' *such that* $\Delta \vdash \mathsf{t}' <: \mathsf{t}$.
2. $\Delta;\Gamma;\Sigma \vdash \mathsf{c} : \mathsf{t}$ *implies* $\emptyset \preceq \Sigma$ *and* $\mathsf{t} = \mathsf{s}$.
3. $\Delta;\Gamma;\Sigma \vdash \mathsf{null} : \mathsf{t}$ *implies* $\emptyset \preceq \Sigma$.
4. $\Delta;\Gamma;\Sigma \vdash \mathsf{o} : \mathsf{t}$ *implies* $\emptyset \preceq \Sigma$ *and* $\mathsf{o} : C \in \Gamma$ *for some* C *such that* $\Delta \vdash C <: \mathsf{t}$.
5. $\Delta;\Gamma;\Sigma \vdash \mathsf{NullExc} : \mathsf{t}$ *implies* $\emptyset \preceq \Sigma$.
6. $\Delta;\Gamma;\Sigma \vdash \mathsf{this} : \mathsf{t}$ *implies* $\emptyset \preceq \Sigma$ *and* $\mathsf{this} : C \in \Gamma$ *for some* C *such that* $\Delta \vdash C <: \mathsf{t}$.
7. $\Delta;\Gamma;\Sigma \vdash \mathsf{e}_1;\mathsf{e}_2 : \mathsf{t}$ *implies* $\Sigma = \Sigma_1 \circ \Sigma_2$, *and* $\mathsf{t} = \mathsf{t}_2$ *and* $\Gamma;\Sigma_i \vdash \mathsf{e}_i : \mathsf{t}_i$ *for some* Σ_i, t_i $(i \in \{1,2\})$.
8. $\Delta;\Gamma;\Sigma \vdash \mathsf{e}.\mathsf{f} := \mathsf{e}' : \mathsf{t}$ *implies* $\Sigma = \Sigma_1 \circ \Sigma_2$, *and* $\Gamma;\Sigma_1 \vdash \mathsf{e} : C$ *and* $\Gamma;\Sigma_2 \vdash \mathsf{e}' : \mathsf{t}$ *with* $\mathsf{f}\,\mathsf{t} \in \mathsf{fields}(C)$ *for some* Σ_1, Σ_2, C.
9. $\Delta;\Gamma;\Sigma \vdash \mathsf{e}.\mathsf{f} : \mathsf{t}$ *implies* $\Gamma;\Sigma \vdash \mathsf{e} : C$ *and* $\mathsf{f}\,\mathsf{t} \in \mathsf{fields}(C)$ *for some* C.
10. $\Delta;\Gamma;\Sigma \vdash \mathsf{e}.\mathsf{m}(\mathsf{e}_1,\ldots,\mathsf{e}_n) : \mathsf{t}$ *implies* $\Delta;\Gamma;\Sigma_0 \vdash \mathsf{e} : C$, *and* $\Delta;\Gamma;\Sigma_i \vdash \mathsf{e}_i : \mathsf{t}_i$ *for* $1 \le i \le n-m$, *and* $\mathsf{e}_{n-m+j} = \mathsf{u}_j$ *for* $1 \le j \le m$, *and* $\Sigma_0 \circ \Sigma_1 \ldots \circ \Sigma_{n-m} \circ \{\mathsf{u}_1 : \rho_1, \ldots, \mathsf{u}_m : \rho_m\} \preceq \Sigma$ *and* $\mathsf{mtype}(\mathsf{m}, C) = \mathsf{t}_1, \ldots, \mathsf{t}_{n-m}, \rho_1, \ldots, \rho_m \to \mathsf{t}$, *for some* m $(0 \le m \le n)$, $\Sigma_i, \mathsf{t}_i, \mathsf{u}_j, \rho_j, C$ $(1 \le i \le n-m,\ 1 \le j \le m)$.
11. $\Delta;\Gamma;\Sigma \vdash \mathsf{new}\ C : \mathsf{t}$ *implies* $\emptyset \preceq \Sigma$ *and* $\Delta \vdash C <: \mathsf{t}$.
12. $\Delta;\Gamma;\Sigma \vdash \mathsf{new}\ (\mathsf{s}, \overline{\mathsf{s}}) : \mathsf{t}$ *implies* $\emptyset \preceq \Sigma$ *and* $\Delta \vdash (\mathsf{s}, \overline{\mathsf{s}}) <: \mathsf{t}$.
13. $\Delta;\Gamma;\Sigma \vdash \mathsf{spawn}\{\ \mathsf{e}\ \} : \mathsf{t}$ *implies* $\Sigma' \preceq \Sigma$, *and ended*(Σ') *and* $\mathsf{t} = \mathit{Object}$ *and* $\Delta;\Gamma;\Sigma' \vdash \mathsf{e} : \mathsf{t}'$ *for some* Σ', t'.

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

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

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

By induction hypothesis we get $\Delta;\Gamma;\Sigma_0 \vdash \mathsf{e} : C$, and $\Delta;\Gamma;\Sigma_i \vdash \mathsf{e}_i : \mathsf{t}_i$ for $1 \le i \le n-m$, and $\mathsf{e}_{n-m+j} = \mathsf{u}_j$ for $1 \le j \le m$, and

$$\Sigma_0 \circ \Sigma_1 \ldots \circ \Sigma_{n-m} \circ \{\mathsf{u}_1 : \rho_1, \ldots, \mathsf{u}_m : \rho_m\} \preceq \Sigma, \mathsf{u} : \varepsilon.\mathsf{end}$$

and $\mathsf{mtype}(\mathsf{m}, C) = \mathsf{t}_1, \ldots, \mathsf{t}_{n-m}, \rho_1, \ldots, \rho_m \to \mathsf{t}$, for some m $(0 \le m \le n)$, $\Sigma_i, \mathsf{t}_i, \mathsf{u}_j, \rho_j, C$ $(1 \le i \le n-m, 1 \le j \le m)$. By definition we also have that $\Sigma, \mathsf{u} : \varepsilon.\mathsf{end} \preceq \Sigma, \mathsf{u} :\updownarrow$, and from transitivity of $\preceq$ we obtain that $\Sigma_0 \circ \Sigma_1 \ldots \circ \Sigma_{n-m} \circ \{\mathsf{u}_1 : \rho_1, \ldots, \mathsf{u}_m : \rho_m\} \preceq \Sigma, \mathsf{u} :\updownarrow$.

Lemma A.2 (Generation for Communication Expressions).

1. $\Delta; \Gamma; \Sigma \vdash \mathsf{connect\ u\ s}\ \{\mathsf{e}\} : \mathsf{t}$ *implies* $\mathsf{s} = \mathsf{begin}.\eta$, *and* $\Delta; \Gamma; \emptyset \vdash \mathsf{u} : \mathsf{begin}.\eta$ *and* $\Delta; \Gamma \setminus \mathsf{u}\,; \Sigma, \mathsf{u} : \eta \vdash \mathsf{e} : \mathsf{t}$, *for some* η.
2. $\Delta; \Gamma; \Sigma \vdash \mathsf{u}.\mathsf{send}(\mathsf{e}) : \mathsf{t}$ *implies* $\mathsf{t} = \mathit{Object}$ *and* $\Delta; \Gamma; \Sigma' \vdash \mathsf{e} : \mathsf{t}'$ *and* $\Sigma' \circ \{\mathsf{u} : !\mathsf{t}\} \preceq \Sigma$ *for some* Σ', t'.
3. $\Delta; \Gamma; \Sigma \vdash \mathsf{u}.\mathsf{receive}(\mathsf{x})\{\mathsf{e}\} : \mathsf{t}$ *implies* $\Delta, X <: \mathsf{t}'; \Gamma, \mathsf{x} : X; \Sigma' \vdash \mathsf{e} : \mathsf{t}$ *and* $X \notin \Gamma \cup \Delta \cup \Sigma \setminus \mathsf{u}$ *and* $\{\mathsf{u} : ?(X <: \mathsf{t}')\} \circ \Sigma' = \Sigma$ *for some* X, t', Σ'.
4. $\Delta; \Gamma\,;\ \Sigma \vdash \mathsf{u}.\mathsf{sendS}(\mathsf{u}') : \mathsf{t}$ *implies* $\mathsf{t} = \mathit{Object}$ *and* $\{\mathsf{u}' : \eta, \mathsf{u} :!(\eta)\} \preceq \Sigma$ *for some* η.
5. $\Delta; \Gamma\,;\ \Sigma \vdash \mathsf{u}.\mathsf{receiveS}(\mathsf{x})\{\mathsf{e}\} : \mathsf{t}$ *implies* $\mathsf{t} = \mathit{Object}$ *and* $\Delta, \eta <: X; \Gamma \setminus \mathsf{x}\ ;\ \{\mathsf{x} : \eta\} \vdash \mathsf{e} : \mathsf{t}'$ *and* $\{\mathsf{u} : ?(X <: \eta)\} \preceq \Sigma$ *for some* X, η.
6. $\Delta; \Gamma; \Sigma \vdash \mathsf{u}.\mathsf{sendCase}(\mathsf{e})\{C_1 \triangleright \mathsf{e}_1; \cdots ; C_n \triangleright \mathsf{e}_n\} : \mathsf{t}$ *implies* $\Delta; \Gamma; \Sigma_0 \vdash \mathsf{e} : C$ *and* $\Delta, X_i <: C_i; \Gamma, \mathsf{x} : X_i; \Sigma', \mathsf{u} : \rho_i \vdash \mathsf{e}_i : \mathsf{t}\ \ \forall i \in \{1, ..., n\}$, *and* $\Sigma_0 \circ \Sigma', \mathsf{u} :!(\!|C_1.\rho_1, \ldots, C_n.\rho_n|\!) \preceq \Sigma$ *for some* $C, \Sigma_0, \Sigma', \rho_1, \ldots, \rho_n$.
7. $\Delta; \Gamma; \Sigma \vdash \mathsf{u}.\mathsf{receiveCase}(\mathsf{x})\{C_1 \triangleright \mathsf{e}_1; \cdots ; C_n \triangleright \mathsf{e}_n\} : \mathsf{t}$ *implies* $\Delta, X_i <: C_i; \Gamma, \mathsf{x} : X_i; \Sigma', \mathsf{u} : \rho_i \vdash \mathsf{e}_i : \mathsf{t}\ \ \forall i \in \{1, ..., n\}$, *and* $\Sigma', \mathsf{u} :?(\!|(X_1 <: C_1).\rho_1, \ldots, (X_n <: C_n).\rho_n|\!) \preceq \Sigma$ *for some* $\Sigma', \rho_1, \ldots, \rho_n$.
8. $\Delta; \Gamma; \Sigma \vdash \mathsf{u}.\mathsf{sendWhile}(\mathsf{e})\{C_1 \triangleright \mathsf{e}_1; \cdots ; C_n \triangleright \mathsf{e}_n\}\{D_1 \triangleright \mathsf{d}_1; \cdots ; D_m \triangleright \mathsf{d}_m\} : \mathsf{t}$ *implies* $\Delta; \Gamma; \emptyset \vdash \mathsf{e} : C$, *and* $\Delta; \Gamma; \{\mathsf{u} : \pi_i\} \vdash \mathsf{e}_i : \mathsf{t}$ *and* $C_i <: C$ $\forall i \in \{1, \ldots, n\}$, *and* $\Delta; \Gamma; \{\mathsf{u} : \pi_{n+j}\} \vdash \mathsf{d}_j : \mathsf{t}$ *and* $D_j <: C\ \forall j \in \{1, \ldots, m\}$, *and* $\{\mathsf{u} :!(\!|(C_1.\pi_1, \ldots, C_n.\pi_n)^*, D_1.\pi_{n+1}, \ldots, D_m.\pi_{n+m}|\!)\} \preceq \Sigma$ *for some* $C, \pi_1, \ldots, \pi_{n+m}$.
9. $\Delta; \Gamma; \Sigma \vdash \mathsf{u}.\mathsf{receiveWhile}(\mathsf{x})\{C_1 \triangleright \mathsf{e}_1; \cdots ; C_n \triangleright \mathsf{e}_n\}\{D_1 \triangleright \mathsf{d}_1; \cdots ; D_m \triangleright \mathsf{d}_m\} : \mathsf{t}$ *implies* $\Delta, X_i <: C_i; \Gamma, \mathsf{x} : X_i; \{\mathsf{u} : \pi_i\} \vdash \mathsf{e}_i : \mathsf{t}\ \ \forall i \in \{1, \ldots, n\}$, *and* $\Delta, Y_j <: D_j; \Gamma, \mathsf{x} : Y_j; \{\mathsf{u} : \pi_{n+j}\} \vdash \mathsf{d}_j : \mathsf{t}\ \ \forall j \in \{1, \ldots, m\}$, *and* $\{\mathsf{u} :?(\!|((X_1 <: C_1).\pi_1, \ldots, (X_n <: C_n).\pi_n)^*, (Y_1 <: D_1).\pi_{n+1}, \ldots, (Y_m <: D_m).\pi_{n+m}|\!)\} \preceq \Sigma$, *for some* $X_1, \ldots, X_n, Y_1, \ldots, Y_m, \pi_1, \ldots, \pi_{n+m}$.

Proof. Similar to the proof of Lemma A.1.

Lemma A.3 (Generation for Threads).

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

Proof. Similar to the proof of Lemma A.1.

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

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

In Lemma 5.4 we show that structural equivalence of terms preserves types. To prove this, we first prove in Lemma A.4 the neutrality of element $\emptyset$, and associativity and commutativity of parallel composition of session environments. Moreover we show in Lemma A.5 various properties of "$\preceq$", "$\|$", and "$\circ$" which easily follow from their definitions.

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

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

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

We define:

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

A running type is *atomic* if it is of one of the following shapes:

$$?(X <: \mathsf{t}), \ !\mathsf{t}, \ ?(X <: \eta), \ !(\eta), !(\!|\tilde{C}.\tilde{\rho}|\!), \ ?(\!|(\tilde{X} <: \tilde{C}).\tilde{\rho}|\!),$$
$$!(\!|(\tilde{C}.\tilde{\pi})^*, \tilde{C}.\tilde{\pi}|\!), \ ?(\!|((\tilde{X} <: \tilde{C}).\tilde{\pi})^*, (\tilde{X} <: \tilde{C}).\tilde{\pi}|\!).$$

Lemma A.5. *1.* $\emptyset \preceq \Sigma_1$, *and* $\Sigma_1 \| \Sigma_2$ *defined imply* $\Sigma_2 \preceq \Sigma_1 \| \Sigma_2$.
2. $\Sigma_1 \| \Sigma_2 \preceq \Sigma$, *implies that there are* Σ_1', Σ_2' *such that* $\Sigma_1 \preceq \Sigma_1'$ *and* $\Sigma_2 \preceq \Sigma_2'$ *and* $\Sigma_1' \| \Sigma_2' = \Sigma$.
3. $\Sigma_1 \preceq \Sigma_1'$, *and* $\Sigma_1' \circ \Sigma_2$ *defined, imply* $\Sigma_1 \circ \Sigma_2$ *defined, and* $\Sigma_1 \circ \Sigma_2 \preceq \Sigma_1' \circ \Sigma_2$.
4. *ended*(Σ_1) *and* $(\Sigma_1 \cup \Sigma_1') \circ \Sigma_2$ *defined imply*
  (a) $\Sigma_1' \circ \Sigma_2$ *defined,*
  (b) $(\Sigma_1 \cup \Sigma_1') \circ \Sigma_2 = \Sigma_1 \| \Sigma_1' \circ \Sigma_2$.
5. $\Sigma \preceq \Sigma'$ *implies* $\Sigma \setminus \mathsf{u} \preceq \Sigma' \setminus \mathsf{u}$.
6. $\{\mathsf{u} : \tau\} \preceq \Sigma$ *implies*
  (a) $\Sigma(\mathsf{u}) \in \{\tau, \tau.\mathsf{end}, \updownarrow\}$ *and* $\mathcal{R}(\Sigma \setminus \mathsf{u}) \subseteq \{\varepsilon, \varepsilon.\mathsf{end}, \updownarrow\}$;
  (b) $\{\mathsf{u} : \tau'\} \preceq \Sigma[\mathsf{u} \mapsto \tau']$ *for all* τ'.
7. $\Sigma, \mathsf{u} : \rho \preceq \Sigma'$ *and* $\Sigma' \circ \Sigma''$ *defined imply*
  (a) $\Sigma'[\mathsf{u} \mapsto \rho'] \circ \Sigma''$ *defined for all* ρ', *proviso that* ρ' *is ended only if* ρ *is ended;*
  (b) $\Sigma, \mathsf{u} : \rho' \preceq \Sigma'[\mathsf{u} \mapsto \rho']$ *for all* ρ'.
8. $\Sigma_1 \| \Sigma_2$ *defined and*
  (a) $\{\mathsf{u} : !\mathsf{t}\} \circ \Sigma_1' \preceq \Sigma_1$ *and* $\{\mathsf{u} : ?(X <: \mathsf{t}')\} \circ \Sigma_2' \preceq \Sigma_2$ *imply* $\mathsf{t} <: \mathsf{t}'$,
  (b) $\{\mathsf{u} : !\eta\} \circ \Sigma_1' \preceq \Sigma_1$ *and* $\{\mathsf{u} : ?(X <: \eta')\} \circ \Sigma_2' \preceq \Sigma_2$ *imply* $\eta <: \eta'$.
9. $\Sigma_1 \circ \Sigma_2 \| \Sigma_3 \circ \Sigma_4$ *defined, and* $\Sigma_1', \mathsf{u} : \rho \preceq \Sigma_1$ *and* $\Sigma_3', \mathsf{u} : \rho' \preceq \Sigma_3$, *where* ρ *and* ρ' *are atomic, imply:*
  (a) $\rho \bowtie \rho'$;
  (b) $\Sigma_1[\mathsf{u} \mapsto \rho_1] \circ \Sigma_2 \| \Sigma_3[\mathsf{u} \mapsto \rho_2] \circ \Sigma_4 = \Sigma_1 \circ \Sigma_2 \| \Sigma_3 \circ \Sigma_4$, *for all* ρ_1, ρ_2 *such that* $\rho_1 \bowtie \rho_2$ *and* ρ_1, ρ_2 *are ended only if* ρ, ρ' *are ended too.*
10. $\Sigma_1 \circ \Sigma_2 \| \Sigma_3 \circ \Sigma_4$ *defined, and* $\{\mathsf{u} : \pi\} \preceq \Sigma_1$ *and* $\{\mathsf{u} : \pi'\} \circ \Sigma_3' = \Sigma_3$ *and* $\pi \bowtie \pi'$ *imply* $\Sigma_1[\mathsf{u} \mapsto \varepsilon] \circ \Sigma_2 \| \{\mathsf{u} : \varepsilon\} \circ \Sigma_3' \circ \Sigma_4 = \Sigma_1 \circ \Sigma_2 \| \Sigma_3 \circ \Sigma_4$.

Proof. For (1) notice that $\emptyset \preceq \Sigma_1$ implies $\mathcal{R}(\Sigma_1) \subseteq \{\varepsilon, \varepsilon.\mathsf{end}, \updownarrow\}$ and that $\Sigma_1 \| \Sigma_2$ defined implies $\Sigma_1(\mathsf{u}) \bowtie \Sigma_2(\mathsf{u})$ for all $\mathsf{u} \in \mathcal{D}(\Sigma_1) \cap \mathcal{D}(\Sigma_2)$.
For (2) one can obtain Σ_1' and Σ_2' by applying to Σ_1 and Σ_2 the same transformations which build Σ from $\Sigma_1 \| \Sigma_2$.
(3) follows easily from the definitions of "$\preceq$" and of "$\circ$".
(4a) is immediate. For (4b), $ended(\Sigma_1)$ and $(\Sigma_1 \cup \Sigma_1') \circ \Sigma_2$ defined imply that $\mathcal{D}(\Sigma_1) \cap \mathcal{D}(\Sigma_2) = \emptyset$.
(5) follows from the definition of "$\preceq$".
(6a) follows from the definition of "$\preceq$" and (6b) is a consequences of (6a).
(6a) implies (7a) and (7b).
The definition of "$\|$", (5) and (6a) imply (8a), (8b) and (9a). Points (9b) and (10) follow from the observation that in all the equated session environments the predicates of u are $\updownarrow$.

Lemma 5.4 (Preservation of Typing under Structural Equivalence). *If* $\Delta; \Gamma; \Sigma \vdash P : \mathsf{thread}$ *and* $P \equiv P'$ *then* $\Delta; \Gamma; \Sigma \vdash P' : \mathsf{thread}$.

Proof. By induction on the derivation of $\equiv$.
For the case where $P' = P \mid \mathsf{null}$, we use Lemma A.3(2), and obtain $\Sigma = \Sigma_1 \| \Sigma_2$ and $\Delta; \Gamma; \Sigma_1 \vdash P : \mathsf{thread}$ and $\Delta; \Gamma; \Sigma_2 \vdash \mathsf{null} : \mathsf{thread}$. Using Lemma A.3(1) and Lemma A.1(3) we get $\Delta; \Gamma; \Sigma_2 \vdash \mathsf{null} : \mathsf{t}_2$, and $\emptyset \preceq \Sigma_2$. Using Lemma A.5(1), we obtain that $\Sigma_1 \preceq \Sigma$, and from that, with Lemma 5.3(2) we obtain that $\Delta; \Gamma; \Sigma \vdash P : \mathsf{thread}$.
For the other two basic cases use Lemmas A.3(1) and A.4(2)-(3). For the induction case use Lemma A.3(1) and induction hypothesis.

The next goal is to prove that term substitution preserves types (Lemma A.6).

Lemma A.6 (Preservation of Typing under Substitution).
1. *If* $\Delta; \Gamma \setminus \mathsf{u} \,;\, \Sigma \vdash \mathsf{e} : \mathsf{t}$ *and* c *is fresh then* $\Delta; \Gamma \,;\, \Sigma[\mathsf{c}/\mathsf{u}] \vdash \mathsf{e}[\mathsf{c}/\mathsf{u}] : \mathsf{t}$.
2. *If* $\Delta; \Gamma, \mathsf{this} : C \,;\, \Sigma \vdash \mathsf{e} : \mathsf{t}$ *and* $\Delta; \Gamma \,;\, \emptyset \vdash \mathsf{o} : C$ *then* $\Delta; \Gamma \,;\, \Sigma \vdash \mathsf{e}[\mathsf{o}/\mathsf{this}] : \mathsf{t}$.
3. *If* $\Delta, X <: \mathsf{t}'; \Gamma, \mathsf{x} : X; \Sigma \vdash \mathsf{e} : \mathsf{t}$ *and* $\Delta; \Gamma; \emptyset \vdash \mathsf{v} : \mathsf{t}''$ *and* $\Delta \vdash \mathsf{t}'' <: \mathsf{t}'$, *then* $\Delta; \Gamma \,;\, \Sigma[\mathsf{t}''/X] \vdash \mathsf{e}[\mathsf{v}/\mathsf{x}] : \mathsf{t}$.
4. *If* $\Delta, X <: \eta; \Gamma; \{\mathsf{x} : X\} \vdash \mathsf{e} : \mathsf{t}$ *and* c *is fresh and* $\Delta \vdash \eta' <: \eta$, *then* $\Delta; \Gamma \,;\, \{\mathsf{c} : \eta'\} \vdash \mathsf{e}[\mathsf{c}/\mathsf{x}] : \mathsf{t}$.

Proof. All points are proven by induction on derivations.

A.3 Types in Subderivations, and Substitutions within Contexts

Lemma 5.5 (Subderivations). *If* $\Delta; \Gamma; \Sigma \vdash E[\mathsf{e}] : \mathsf{t}$ *then there exist* $\Sigma_1, \Sigma_2, \mathsf{x}, \mathsf{t}', \mathsf{x}$ *fresh in* E, Γ, *such that* $\Sigma = \Sigma_1 \circ \Sigma_2$, *and* $\Delta; \Gamma; \Sigma_1 \vdash \mathsf{e} : \mathsf{t}'$, *and* $\Delta; \Gamma, \mathsf{x} : \mathsf{t}'; \Sigma_2 \vdash E[\mathsf{x}] : \mathsf{t}$.

Proof. By induction on E, and using Generation Lemmas. For example if $E = [\,]; \mathsf{e}'$, then $\Delta; \Gamma; \Sigma \vdash \mathsf{e}; \mathsf{e}' : \mathsf{t}$ implies $\Sigma = \Sigma_1 \circ \Sigma_2$ and $\Gamma; \Sigma_1; \pi \vdash \mathsf{e} : \mathsf{t}'$ and $\Gamma; \Sigma_2; \pi \vdash \mathsf{e}' : \mathsf{t}$ by Lemma A.1(9). Then we get $\Delta; \Gamma, \mathsf{x} : \mathsf{t}'; \Sigma_2 \vdash \mathsf{x}; \mathsf{e}' : \mathsf{t}$ by rules **Var** and **Seq**.

Lemma 5.6 (Context Substitution). *If* $\Delta; \Gamma; \Sigma_1 \vdash \mathsf{e} : \mathsf{t}'$, *and* $\Delta; \Gamma, \mathsf{x} : \mathsf{t}'; \Sigma_2 \vdash E[\mathsf{x}] : \mathsf{t}$, *and* $\Sigma_1 \circ \Sigma_2$ *is defined, then* $\Delta; \Gamma; \Sigma_1 \circ \Sigma_2 \vdash E[\mathsf{e}] : \mathsf{t}$.

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

Theorem 5.7 (Subject Reduction).

1. $\Delta; \Gamma; \Sigma \vdash \mathsf{e} : \mathsf{t}$, *and* $\Gamma; \Sigma \vdash h$, *and* $\mathsf{e}, h \longrightarrow \mathsf{e}', h'$ *imply* $\Delta; \Gamma'; \Sigma \vdash \mathsf{e}' : \mathsf{t}$, *and* $\Gamma'; \Sigma \vdash h'$, *with* $\Gamma \subseteq \Gamma'$.
2. $\Delta; \Gamma; \Sigma \vdash P; h$ *and* $P, h \longrightarrow P', h'$ *imply* $\Delta; \Gamma'; \Sigma' \vdash P; h'$ *with* $\Gamma \subseteq \Gamma'$ *and* $\Sigma \subseteq \Sigma'$.

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

Rule Spawn-R.

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

0) $h' = h$ and $P' = E[\mathsf{null}] \mid \mathsf{e}$,
1) $\Delta; \Gamma; \Sigma \vdash E[\mathsf{spawn}\,\{\ \mathsf{e}\ \}] : \mathsf{t}$,
2) $\Gamma; \Sigma \vdash h$.

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

Applying Lemma 5.5 on 1) we obtain for some $\mathsf{t}', \Sigma_1, \Sigma_2$:

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

From 3) and Lemma A.1(13), we get for some t'', Σ_1':

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

From 5), type rule **Null**, and Lemma 5.6, we get:

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

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

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

From 11), 12) and rule **Par** we get:

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

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

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

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

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

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

Therefore, from 13), 14), 15), and Lemma 5.3(2), we obtain:

16) $\Delta; \Gamma; \Sigma \vdash \mathsf{e} \mid E[\mathsf{null}] : \mathsf{thread}$.

The case concludes by taking $\Sigma' = \Sigma$, $\Gamma' = \Gamma$ and with 15) and 0).

Rule Connect-R.

Then, we have that

0) $P = E_1[\mathsf{connect\ c\ s}\{\mathsf{e}_1\}] \mid E_2[\mathsf{connect\ c\ s}'\{\mathsf{e}_2\}]$,
1) $h' = h :: \mathsf{c}'$, with c' is fresh in h,
2) $P' = E_1[\mathsf{e}_1[\mathsf{c}'/\mathsf{c}]] \mid E_2[\mathsf{e}_2[\mathsf{c}'/\mathsf{c}]]$.

The aim of the next steps is to derive types for e_1 and for e_2.

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

3) $\Sigma = \Sigma_1 \| \Sigma_2$,
4) $\Delta; \Gamma; \Sigma_i \vdash E_i[\mathsf{connect\ c\ s}_i\{\mathsf{e}_i\}] : \mathsf{t}_i$ for $(i \in \{1,2\})$,
5) $\Gamma; \Sigma \vdash h$,

where $\mathsf{s}_1 \bowtie \mathsf{s}_2$.

From 4), applying Lemma 5.5, there exist Σ_{11}, Σ_{12}, Σ_{21}, Σ_{22}, $\mathsf{t}'_1, \mathsf{t}'_2$, such that:

6) $\Sigma_i = \Sigma_{i1} \circ \Sigma_{i2}$,
7) $\Delta; \Gamma; \Sigma_{i1} \vdash \mathsf{connect\ c\ s}_i\{\mathsf{e}_i\} : \mathsf{t}'_i$ $(i \in \{1,2\})$,
8) $\Delta; \Gamma, \mathsf{x}_i : \mathsf{t}'_i; \Sigma_{i2} \vdash E_i[\mathsf{x}_i] : \mathsf{t}_i$ $(i \in \{1,2\})$.

From 7), and Lemma A.2(1) we obtain for some η_1, η_2:

9) $\Delta; \Gamma; \emptyset \vdash \mathsf{c} : \mathsf{s}_i$,
10) $\mathsf{s}_i = \mathsf{begin}.\eta_i$,
11) $\Delta; \Gamma \setminus \mathsf{c}; \Sigma_{i1}, \mathsf{c} : \eta_i \vdash \mathsf{e}_i : \mathsf{t}'_i$ $(i \in \{1,2\})$.

The aim of the next steps is to derive types for P' in a session environment Σ', so that $\Sigma \subseteq \Sigma'$.

From 1) and 11), and Lemma A.6(1),we get:

12) $\Delta; \Gamma; \Sigma_{i1}, \mathsf{c}' : \eta_i \vdash \mathsf{e}_i[\mathsf{c}'/\mathsf{c}] : \mathsf{t}'_i$ $(i \in \{1,2\})$.

From 12), 8) and Lemma 5.6, we obtain (notice that $(\Sigma_{i1}, \mathsf{c}' : \eta_i) \circ \Sigma_{i2}$ is defined by 6) since c' is fresh):

13) $\Delta; \Gamma; (\Sigma_{i1}, \mathsf{c}' : \eta_i) \circ \Sigma_{i2} \vdash E_i[\mathsf{e}_i[\mathsf{c}'/\mathsf{c}]] : \mathsf{t}'_i$ $(i \in \{1,2\})$.

Applying rules **Start** and **Par** on 13), and also the fact that

$$(\Sigma_{11}, \mathsf{c}' : \eta_1) \circ \Sigma_{12} \| (\Sigma_{21}, \mathsf{c}' : \eta_2) \circ \Sigma_{22} = \Sigma, \mathsf{c}' :\updownarrow,$$

since $\mathsf{s}_1 \bowtie \mathsf{s}_2$ implies $\eta_1 \bowtie \eta_2$, we obtain:

14) $\Delta; \Gamma; \Sigma, \mathsf{c}' :\updownarrow \vdash E_1[\mathsf{e}_1[\mathsf{c}'/\mathsf{c}]] \mid E_2[\mathsf{e}_2[\mathsf{c}'/\mathsf{c}]] : \mathsf{thread}$.

Take

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

This gives, trivially that:

16) $\Sigma \subseteq \Sigma'$.

Also, from 1) and 5) we obtain:

17) $\Gamma; \Sigma' \vdash h'$.

The case concludes by considering 14), 15), 16) and 17).

Rule ComS-R.

Therefore, we have that

0) $P = E_1[\mathsf{c}\,.\mathsf{send}\,(\mathsf{v}\,)] \mid E_2[\mathsf{c}\,.\mathsf{receive}\,(\mathsf{x}\,)\{\mathsf{e}\,\}], \quad P' = E_1[\mathsf{null}\,] \mid E_2[\mathsf{e}\,[\mathsf{v}/\mathsf{x}]]$,

1) $h' = h, \quad \Gamma; \Sigma \vdash h$.

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

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

3) $\Delta; \Gamma; \Sigma_2 \vdash E_2[\mathsf{c}\,.\mathsf{receive}\,(\mathsf{x}\,)\{\mathsf{e}\,\}] : \mathsf{t}_2$,

4) $\Sigma = \Sigma_1 \| \Sigma_2$.

The aim of the next steps is to derive types for $\mathsf{c}\,.\mathsf{receive}\,(\mathsf{x}\,)\{\mathsf{e}\,\}$ *and* $\mathsf{c}\,.\mathsf{send}\,(\mathsf{v}\,)$*, and for* $E_1[\mathsf{x}\,]$ *and* $E_2[\mathsf{x}\,]$.

From 2) and Lemma 5.5, we obtain for some $\Sigma_{11}, \Sigma_{12}, \mathsf{t}'_1$:

5) $\Delta; \Gamma; \Sigma_{11} \vdash \mathsf{c}\,.\mathsf{send}\,(\mathsf{v}\,) : \mathsf{t}'_1$,

6) $\Delta; \Gamma, \mathsf{z} : \mathsf{t}'_1; \Sigma_{12} \vdash E_1[\mathsf{z}\,] : \mathsf{t}_1$,

7) $\Sigma_1 = \Sigma_{11} \circ \Sigma_{12}$.

From 5) and Lemmas A.2(2)and A.1(2), A.1(3), A.1(4), we obtain for some t''_1:

8) $\Delta; \Gamma; \emptyset \vdash \mathsf{v} : \mathsf{t}''_1$,

9) $\{\mathsf{c} : !\mathsf{t}''_1\} \preceq \Sigma_{11}$.

From 3), and Lemma 5.5, we obtain for some $\Sigma_{21}, \Sigma_{22}, \mathsf{t}'_2$:

10) $\Delta; \Gamma; \Sigma_{21} \vdash \mathsf{c}\,.\mathsf{receive}\,(\mathsf{x}\,)\{\mathsf{e}\,\} : \mathsf{t}'_2$,

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

12) $\Sigma_2 = \Sigma_{21} \circ \Sigma_{22}$.

From 10), by Lemma A.2(3), we obtain for some X, t''_2 and Σ'_{21}:

13) $\{\mathsf{c} : ?(X <: \mathsf{t}''_2)\} \circ \Sigma'_{21} = \Sigma_{21}$,

14) $\Delta, X <: \mathsf{t}''_2; \Gamma, \mathsf{x} : X; \Sigma'_{21} \vdash \mathsf{e} : \mathsf{t}'_2$.

The aim of the next steps is to derive types for $E_1[\mathsf{null}]$ *and* $E_2[\mathsf{e}\,[\mathsf{v}/\mathsf{x}]]$.

From 9), and 7), and Lemma A.5(7a) and (6b), we obtain:

15) $\Sigma_{11}[\mathsf{c} \mapsto \varepsilon] \circ \Sigma_{12}$ is defined,

16) $\{\mathsf{c} : \varepsilon\} \preceq \Sigma_{11}[\mathsf{c} \mapsto \varepsilon]$.

By rules **Null**, and **WeakES**, we obtain $\Delta; \Gamma; \{\mathsf{c} : \varepsilon\} \vdash \mathsf{null} : \mathsf{t}'_1$.

Then, by 16) and Lemma 5.3(1) we obtain:

17) $\Delta; \Gamma; \Sigma_{11}[\mathsf{c} \mapsto \varepsilon] \vdash \mathsf{null} : \mathsf{t}'_1$.

From 6), 15) , 17), and Lemma 5.6, we obtain:

18) $\Delta; \Gamma; \Sigma_{11}[\mathsf{c} \mapsto \varepsilon] \circ \Sigma_{12} \vdash E_1[\mathsf{null}] : \mathsf{t}_1$.

From 7), 9) by Lemma A.5(3) we get:

19) $\{\mathsf{c} : !\mathsf{t}''_1\} \circ \Sigma_{12} \preceq \Sigma_1$.

12) and 13) imply:

20) $\{\mathsf{c} : ?(X <: \mathsf{t}''_2)\} \circ \Sigma'_{21} \circ \Sigma_{22} = \Sigma_2$.

19), and 20) imply by Lemma A.5(8a)

21) $\mathsf{t}''_1 <: \mathsf{t}''_2$.

Therefore, with 8) and 14) we obtain by Lemma A.6(3):

22) $\Delta; \Gamma; \Sigma'_{21} \vdash \mathsf{e}[\mathsf{v}/\mathsf{x}] : \mathsf{t}'_2$,

and then by 11) and Lemma 5.6 and possibly rule **WeakES**:

23) $\Delta; \Gamma; \{\mathsf{c} : \varepsilon\} \circ \Sigma'_{21} \circ \Sigma_{22} \vdash E_2[\mathsf{e}[\mathsf{v}/\mathsf{x}]] : \mathsf{t}_2$.

Furthermore, from 4), 7), 12), 9), 13), 21) and Lemma A.5(10), we obtain:

24) $\Sigma_{11}[\mathsf{c} \mapsto \varepsilon] \circ \Sigma_{12} \| \{\mathsf{c} : \varepsilon\} \circ \Sigma_{21} \circ \Sigma_{22} = \Sigma_{11} \circ \Sigma_{12} \| \Sigma_{21} \circ \Sigma_{22} = \Sigma$.

*The case concludes by applying rules **Par** and **Start** to 18) and 23) taking 24) and 1) into account.*

Rule ComSS-R.

We have:

0) $P = E_1[\mathsf{c}.\mathsf{sendS}(\mathsf{c}')] \mid E_2[\mathsf{c}.\mathsf{receiveS}(\mathsf{x})\{\mathsf{e}\}]$,
1) $P' = E_1[\mathsf{null}] \mid \mathsf{e}[\mathsf{c}/\mathsf{x}] \mid E_2[\mathsf{null}]$,
2) $h' = h, \quad \Gamma; \Sigma \vdash h$,
3) $\Delta; \Gamma; \Sigma \vdash P : \mathsf{thread}$.

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

4) $\Delta; \Gamma; \Sigma_1 \vdash E_1[\mathsf{c}.\mathsf{sendS}(\mathsf{c}')] : \mathsf{t}_1$,
5) $\Delta; \Gamma; \Sigma_2 \vdash E_2[\mathsf{c}.\mathsf{receiveS}(\mathsf{x})\{\mathsf{e}\}] : \mathsf{t}_2$,
6) $\Sigma = \Sigma_1 \| \Sigma_2$.

The aim of the next steps is to derive types for $E_1[\mathsf{null}]$ and $E_2[\mathsf{null}]$.

From 4), Lemma 5.5 and Lemma A.2(4) we get for some $\Sigma_{11}, \Sigma_{12}, \mathsf{t}'_1, \eta \neq \varepsilon.\mathsf{end}$:

7) $\Delta; \Gamma; \Sigma_{11} \vdash \mathsf{c}.\mathsf{sendS}(\mathsf{c}') : \mathsf{t}'_1$,
8) $\Delta; \Gamma, \mathsf{y} : \mathsf{t}'_1; \Sigma_{12} \vdash E_1[\mathsf{y}] : \mathsf{t}_1$,
9) $\Sigma_1 = \Sigma_{11} \circ \Sigma_{12}$,
10) $\mathsf{t}'_1 = \mathit{Object}$,
11) $\{\mathsf{c} :!(\eta), \mathsf{c}' : \eta\} \preceq \Sigma_{11}$.

11) and Lemma A.5(5) imply

12) $\{\mathsf{c}' : \eta\} \preceq \Sigma_{11} \setminus \mathsf{c}$,

which gives by $\eta \neq \varepsilon.\mathsf{end}$ and Lemma A.5(6a), for some Σ'_{11}:

13) $\Sigma_{11} = \Sigma'_{11}, \mathsf{c}' : \eta$.

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

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

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

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

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

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

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

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

From 18), 16), and using Lemma 5.3(1) we obtain:

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

From 8), 19), 17) and Lemma 5.6, we obtain:

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

From 5), Lemma 5.5 and Lemma A.2(5) we get for some $\Sigma_{21}, \Sigma_{22}, \mathsf{t}'_2, \eta' \neq \varepsilon.\mathsf{end}$:

21) $\Delta\,;\,\Gamma\,;\,\Sigma_{21} \vdash \mathsf{c}\,.\mathsf{receiveS}\,(\mathsf{x}\,)\{\mathsf{e}\,\} : \mathsf{t}\,'_2$
22) $\Delta\,;\,\Gamma, \mathsf{z} : \mathsf{t}\,'_2\,;\,\Sigma_{22} \vdash E_2[\mathsf{z}\,] : \mathsf{t}\,_2$,
23) $\Sigma_2 = \Sigma_{21} \circ \Sigma_{22}$,
24) $\mathsf{t}\,'_2 = \mathit{Object}$,
25) $\{\mathsf{c} :?(X <: \eta')\} \preceq \Sigma_{21}$,
26) $\Delta, X <: \eta'\,;\,\Gamma \setminus \mathsf{x}\,\,;\,\{\mathsf{x} : X\} \vdash \mathsf{e} : \mathsf{t}\,'$.

With a proof similar to that of 20) we can show:

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

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

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

28) $\Sigma_{11} \circ \Sigma_{12} = \{\mathsf{c}\,' : \eta\} \| \Sigma'_{11} \circ \Sigma_{12}$.

6), 9), 23) and 28) imply:

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

From 29), 15), 25) by Lemma A.5(9a) we get:

30) $!(\eta) \bowtie ?(X <: \eta')$,

which implies by definition of $\bowtie$:

31) $\eta <: \eta'$.

From 26) and 31) using Lemma A.6(4) we obtain:

32) $\Delta\,;\,\Gamma\,;\,\{\mathsf{c}\,' : \eta\} \vdash \mathsf{e}\,[\mathsf{c}'/\mathsf{x}] : \mathsf{t}\,'$.

Again from 29), 15), 25) by Lemma A.5(9b) we get:

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

6), 9), 28), 23), and 33) imply:

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

The case concludes by applying rules **Par** *and* **Start** *to 20), 27), 32) by taking into account 34) and 2).*

Rule ComSCaseSuccess-R.

Then, we have that:

0) $P \;= E_1[\mathsf{c}\,.\mathsf{sendCase}\,(\mathsf{o}\,)\{\,C_1 \triangleright \mathsf{e}\,_1 ; \cdots ; C_n \triangleright \mathsf{e}\,_n\}] \mid$
$E_2[\mathsf{c}\,.\mathsf{receiveCase}\,(\mathsf{x}\,)\{\,C'_1 \triangleright \mathsf{e}\,'_1 ; \cdots ; C'_m \triangleright \mathsf{e}\,'_m\}]$,
1) $h' = h$,
2) $\Gamma ; \Sigma \vdash\, h$,
3) $P' \;= E_1[\mathsf{e}\,_i] \mid E_2[\mathsf{e}\,_k[\mathsf{o}/\mathsf{x}]], h$,
4) $h(\mathsf{o}\,) = (C, \dots)$ and $C <: C_i$ and $\forall j < i (C \not<: C_j)$, where $i \in \{1, \dots, n\}$,
and $C <: C'_k$ and $\forall l < k (C \not<: C'_l)$, where $k \in \{1, \dots, m\}$.

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

5) $\Sigma = \Sigma_1 \| \Sigma_2$,
6) $\Delta ; \Gamma ; \Sigma_1 \vdash\, E_1[\mathsf{c}\,.\mathsf{sendCase}\,(\mathsf{o}\,)\{\,C_1 \triangleright \mathsf{e}\,_1 ; \cdots ; C_n \triangleright \mathsf{e}\,_n\}] : \mathsf{t}\,_1$,
7) $\Delta ; \Gamma ; \Sigma_2 \vdash\, E_2[\mathsf{c}\,.\mathsf{receiveCase}\,(\mathsf{x}\,)\{\,C'_1 \triangleright \mathsf{e}\,'_1 ; \cdots ; C'_m \triangleright \mathsf{e}\,'_m\}] : \mathsf{t}\,_2$.

4) and 2) imply

8) $\Gamma(\mathsf{o}\,) = C$.

From 6), 7) applying Lemma 5.5, there exist Σ_{11}, Σ_{12}, Σ_{21}, Σ_{22}, $\mathtt{t}'_1$, $\mathtt{t}'_2$ so that:

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

10) $\Delta; \Gamma; \Sigma_{11} \vdash \mathtt{c}.\mathsf{sendCase}(\mathtt{o})\{C_1 \triangleright \mathtt{e}_1; \cdots ; C_n \triangleright \mathtt{e}_n\} : \mathtt{t}'_1$,

11) $\Delta; \Gamma, \mathtt{y} : \mathtt{t}'_1; \Sigma_{12} \vdash E_1[\mathtt{y}] : \mathtt{t}_1$,

12) $\Delta; \Gamma; \Sigma_{21} \vdash \mathtt{c}.\mathsf{receiveCase}(\mathtt{x})\{C'_1 \triangleright \mathtt{e}'_1; \cdots ; C'_m \triangleright \mathtt{e}'_m\} : \mathtt{t}'_2$,

13) $\Delta; \Gamma, \mathtt{z} : \mathtt{t}'_2; \Sigma_{22} \vdash E_2[\mathtt{z}] : \mathtt{t}_2$.

The aim of the next steps is to find types for $\mathtt{e}_i$, *and* $E_1[\mathtt{e}_i]$.

From 10), 8), and Lemmas A.2(6) and A.1(4), we obtain for some Σ_0, Σ'_{11}, $\rho_1, \ldots, \rho_n$:

14) $\Delta; \Gamma; \Sigma_0 \vdash \mathtt{o} : C'$ for some C' such that $C <: C'$,

15) $\emptyset \preceq \Sigma_0$,

16) $\Delta; \Gamma; \Sigma'_{11}, \mathtt{c} : \rho_j \vdash \mathtt{e}_j : \mathtt{t}'_1 \ \forall j \in \{1, ..., n\}$,

17) $\Sigma_0 \circ \Sigma'_{11}, \mathtt{c} : !(\!|C_1.\rho_1, \ldots, C_n.\rho_n|\!) \preceq \Sigma_{11}$.

15) and 17) imply by Lemma A.5(3) and transitivity of $\preceq$:

18) $\Sigma'_{11}, \mathtt{c} : !(\!|C_1.\rho_1, \ldots, C_n.\rho_n|\!) \preceq \Sigma_{11}$,

and then by Lemma A.5(7b) and A.5(7a) and 9)

19) $\Sigma'_{11}, \mathtt{c} : \rho_i \preceq \Sigma_{11}[\mathtt{c} \mapsto \rho_i]$,

20) $\Sigma_{11}[\mathtt{c} \mapsto \rho_i] \circ \Sigma_{12}$ is defined.

From 16) and 19) we get by Lemma 5.3(1):

21) $\Delta; \Gamma; \Sigma_{11}[\mathtt{c} \mapsto \rho_i] \vdash \mathtt{e}_i : \mathtt{t}'_1$,

which together with 11), 20) implies by Lemma 5.6:

22) $\Delta; \Gamma; \Sigma_{11}[\mathtt{c} \mapsto \rho_i] \circ \Sigma_{12} \vdash E_1[\mathtt{e}_i] : \mathtt{t}_1$.

The aim of the next steps is to show that the type of $\mathtt{c}$ *used to type* $\mathtt{e}_i$ *is dual to that used to type* $\mathtt{e}'_k$, *and to find types for* $\mathtt{e}'_k[\mathtt{o}/\mathtt{x}]$ *and* $E_2[\mathtt{e}'_k[\mathtt{o}/\mathtt{x}]]$.

From 12), and Lemma A.2(7), we obtain for some Σ'_{21}, $\mathtt{t}'_2$, $\rho'_1, \ldots, \rho'_m$:

23) $\Delta, X_l <: C'_l; \Gamma, \mathtt{x} : X_l; \Sigma'_{21}, \mathtt{c} : \rho'_l \vdash \mathtt{e}'_l : \mathtt{t}'_2 \ \forall l \in \{1, ..., m\}$,

24) $\Sigma'_{21}, \mathtt{c} : ?(\!|(X_1 <: C'_1).\rho'_1, \ldots, (X_m <: C'_m).\rho'_m|\!) \preceq \Sigma_{21}$.

Because of 18), 24), being $\Sigma_{11} \circ \Sigma_{12} \| \Sigma_{21} \circ \Sigma_{22}$ defined, and by Lemma A.5(9a) we obtain that:

25) $!(\!|C_1.\rho_1, \ldots, C_n.\rho_n|\!) \bowtie ?(\!|(X_1 <: C'_1).\rho'_1, \ldots, (X_m <: C'_m).\rho'_m|\!)$.

Therefore, by definition of the duality relation:

26) $\rho_j \bowtie \rho'_l[^{C_j \curlyvee C_l}/_{X_l}] \ \forall j \in \{1, ..., n\}$ and $\forall l \in \{1, ..., m\}$.

4) and 8) imply by rules **Oid** and **Sub** (notice that $C_i \curlyvee C'_k \neq \bot$ by 4)):

27) $\Delta; \Gamma; \emptyset \vdash \mathtt{o} : C_i \curlyvee C'_k$.

Similarly to previous case we can derive:

28) $\Sigma_{21}[\mathtt{c} \mapsto \rho'_k[^{C_i \curlyvee C'_k}/_{X_k}]] \circ \Sigma_{22}$ defined,

29) $\Delta, X_k <: C'_k; \Gamma, \mathtt{x} : X; \Sigma_{21}[\mathtt{c} \mapsto \rho'_k] \vdash \mathtt{e}'_k : \mathtt{t}'_2$.

Applying Lemma A.6(3) to 29), 27), 26), and 4), taking into account that X_k can occur only in ρ'_k we derive:

30) $\Delta; \Gamma; \Sigma_{21}[\mathtt{c} \mapsto \rho'_k[^{C_i \curlyvee C'_k}/_{X_k}]] \vdash \mathtt{e}'_k[\mathtt{o}/\mathtt{x}] : \mathtt{t}'_2$,

which together with 13), 28) implies by Lemma 5.6:

31) $\Delta; \Gamma; \Sigma_{21}[\mathtt{c} \mapsto \rho'_k[^{C_i \curlyvee C'_k}/_{X_k}]] \circ \Sigma_{22} \vdash E_2[\mathtt{e}'_k[\mathtt{o}/\mathtt{x}]] : \mathtt{t}_2$.

From 26), 9), 19), 24), 5) by Lemma A.5(9b) we obtain that:

32) $\Sigma_{11}[\mathsf{c} \mapsto \rho_k] \circ \Sigma_{12} \parallel \Sigma_{21}[\mathsf{c} \mapsto \rho'_k[C_i \curlyvee C'_k/X_k]] \circ \Sigma_{22} \;=\; \Sigma_{11} \circ \Sigma_{12} \parallel \Sigma_{21} \circ \Sigma_{22} = \Sigma$.

The case concludes by applying rules **Par** *and* **Start** *to 22), 31), and taking into account 32) and 1), 2).*

Rule ComSWhile-R.

Then, we have that:

0) $P \;=\; E_1[\mathsf{c}\,.\mathsf{sendWhile}\,(\mathsf{e}\,)\{C_1 \triangleright \mathsf{e}_1; \cdots ; C_n \triangleright \mathsf{e}_n\}\{D_1 \triangleright \mathsf{d}_1; \cdots ; D_m \triangleright \mathsf{d}_m\}] \mid E_2[\mathsf{c}\,.\mathsf{receiveWhile}\,(\mathsf{x}\,)\{C'_1 \triangleright \mathsf{e}'_1; \cdots ; C'_{n'} \triangleright \mathsf{e}'_{n'}\}\{D'_1 \triangleright \mathsf{d}'_1; \cdots ; D'_{m'} \triangleright \mathsf{d}'_{m'}\}]$,
1) $h' = h$,
2) $\Gamma; \Sigma \vdash h$,
3) $P' \;=\; E_1[\hat{\mathsf{e}}\,] \mid E_2[\check{\mathsf{e}}\,]$,

where we are using the shorthands:

4) $\hat{\mathsf{e}} = \mathsf{c}\,.\mathsf{sendCase}\,(\mathsf{e}\,)\{C_1 \triangleright \mathsf{e}^{\natural}_1; \cdots ; C_n \triangleright \mathsf{e}^{\natural}_n, D_1 \triangleright \mathsf{d}_1; \cdots ; D_m \triangleright \mathsf{d}_m\}$,
5) $\check{\mathsf{e}} = \mathsf{c}\,.\mathsf{receiveCase}\,(\mathsf{x}\,)\{C'_1 \triangleright \mathsf{e}^{\flat}_1; \cdots ; C'_{n'} \triangleright \mathsf{e}^{\flat}_{n'}, D'_1 \triangleright \mathsf{d}'_1; \cdots ; D'_{m'} \triangleright \mathsf{d}'_{m'}\}$,
6) $\mathsf{e}^{\natural}_i \equiv \mathsf{e}_i; \mathsf{c}\,.\mathsf{sendWhile}\,(\mathsf{e}\,)\{C_1 \triangleright \mathsf{e}_1; \cdots ; C_n \triangleright \mathsf{e}_n\}\{D_1 \triangleright \mathsf{d}_1; \cdots ; D_m \triangleright \mathsf{d}_m\}$,
7) $\mathsf{e}^{\flat}_k = \mathsf{e}'_k; \mathsf{c}\,.\mathsf{receiveWhile}\,(\mathsf{x}\,)\{C'_1 \triangleright \mathsf{e}'_1; \cdots ; C'_{n'} \triangleright \mathsf{e}'_{n'}\}\{D'_1 \triangleright \mathsf{d}'_1; \cdots ; D'_{m'} \triangleright \mathsf{d}'_{m'}\}$.

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

8) $\Sigma = \Sigma_1 \| \Sigma_2$,
9) $\Delta; \Gamma; \Sigma_1 \vdash E_1[\mathsf{c}\,.\mathsf{sendWhile}\,(\mathsf{e}\,)\{C_1 \triangleright \mathsf{e}_1; \cdots ; C_n \triangleright \mathsf{e}_n\}\{D_1 \triangleright \mathsf{d}_1; \cdots ; D_m \triangleright \mathsf{d}_m\}] : \mathsf{t}_1$,
10) $\Delta; \Gamma; \Sigma_2 \vdash E_2[\mathsf{c}\,.\mathsf{receiveWhile}\,(\mathsf{x}\,)\{C'_1 \triangleright \mathsf{e}'_1; \cdots ; C'_{n'} \triangleright \mathsf{e}'_{n'}\}\{D'_1 \triangleright \mathsf{d}'_1; \cdots ; D'_{m'} \triangleright \mathsf{d}'_{m'}\}] : \mathsf{t}_2$.

From 9), 10) applying Lemma 5.5, there exist Σ_{11},Σ_{12}, Σ_{21}, Σ_{22}, t'_1, t'_2 so that:

11) $\Sigma_1 = \Sigma_{11} \circ \Sigma_{12}, \quad \Sigma_2 = \Sigma_{21} \circ \Sigma_{22}$,
12) $\Delta; \Gamma; \Sigma_{11} \vdash \mathsf{c}\,.\mathsf{sendWhile}\,(\mathsf{e}\,)\{C_1 \triangleright \mathsf{e}_1; \cdots ; C_n \triangleright \mathsf{e}_n\}\{D_1 \triangleright \mathsf{d}_1; \cdots ; D_m \triangleright \mathsf{d}_m\} : \mathsf{t}'_1$,
13) $\Delta; \Gamma, \mathsf{y} : \mathsf{t}'_1; \Sigma_{12} \vdash E_1[\mathsf{y}] : \mathsf{t}_1$,
14) $\Delta; \Gamma; \Sigma_{21} \vdash \mathsf{c}\,.\mathsf{receiveWhile}\,(\mathsf{x}\,)\{C'_1 \triangleright \mathsf{e}'_1; \cdots ; C'_{n'} \triangleright \mathsf{e}'_{n'}\}\{D'_1 \triangleright \mathsf{d}'_1; \cdots ; D'_{m'} \triangleright \mathsf{d}'_{m'}\} : \mathsf{t}'_2$,
15) $\Delta; \Gamma, \mathsf{z} : \mathsf{t}'_2; \Sigma_{22} \vdash E_2[\mathsf{z}] : \mathsf{t}_2$.

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

From 12), and Lemma A.2(8), we obtain for some $\pi_1, \ldots, \pi_{n+m}, \mathsf{t}'_1$:

16) $\{\mathsf{c} : !(\!|(C_1.\pi_1, \ldots, C_n.\pi_n)^*, D_1.\pi_{n+1}, \ldots, D_m.\pi_{n+m}|\!)\} \preceq \Sigma_{11}$,
17) $\Delta; \Gamma; \emptyset \vdash \mathsf{e} : C$,
18) $\Delta; \Gamma; \{\mathsf{u} : \pi_i\} \vdash \mathsf{e}_i : \mathsf{t}'_1$ and $C_i <: C$ for all $i \in \{1, \ldots, n\}$,
19) $\Delta; \Gamma; \{\mathsf{u} : \pi_{n+j}\} \vdash \mathsf{d}_j : \mathsf{t}'_1$ and $D_j <: C$ for all $j \in \{1, \ldots, m\}$.

We will be using $\hat{\pi}$ as a shorthand defined as follows:

20) $\hat{\pi} = !(\!|C_1.\pi^{\natural}_1, \ldots, C_n.\pi^{\natural}_n, D_1.\pi_{n+1}, \ldots, D_m.\pi_{n+m}|\!)$, where
21) $\pi^{\natural}_i = \pi_i.!(\!|(C_1.\pi_1, \ldots, C_n.\pi_n)^*, D_1.\pi_{n+1}, \ldots, D_m.\pi_{n+m}|\!)$ for $i \in \{1, \ldots, n\}$.

By application of type rules **Null**, **SendCase**, **Seq**, **SendWhile** on 17) and 18), and using the shorthands 4) and 20) we obtain:

22) $\Delta; \Gamma; \{\mathsf{c} : \hat{\pi}\} \vdash \hat{\mathsf{e}} : \mathsf{t}'_1$.

By application of Lemma A.5(6b) on 16) we get that:

23) $\{\mathsf{c} : \hat{\pi}\} \preceq \Sigma_{11}[\mathsf{c} \mapsto \hat{\pi}]$,

which together with 22) implies by Lemma 5.3(1)

24) $\Delta; \Gamma; \Sigma_{11}[\mathsf{c} \mapsto \hat{\pi}] \vdash \hat{\mathsf{e}} : \mathsf{t}'_1$.

Applying Lemma A.5(7a) to 16) and 11) we have that:

25) $\Sigma_{11}[\mathsf{c} \mapsto \hat{\pi}] \circ \Sigma_{12}$ is defined.

Therefore applying Lemma 5.6 on 13), 24) and 25) we obtain:

26) $\Delta; \Gamma; \Sigma_{11}[\mathsf{c} \mapsto \hat{\pi}] \circ \Sigma_{12} \vdash E_1[\hat{\mathsf{e}}\,] : \mathsf{t}_1$.

The aim of the next steps is to find types for $\check{\mathsf{e}}$ *and* $E_2[\check{\mathsf{e}}\,]$.

By arguments similar to those used to get 16) and 18), we obtain from 14) for some $\pi'_1, \ldots, \pi'_{n'+m'}, \mathsf{t}'_2$:

27) $?(\!|((X_1 <: C'_1).\pi'_1, \ldots, (X_{n'} <: C'_{n'}).\pi'_{n'})^*,$
$(Y_1 <: D'_1).\pi'_{n'+1}, \ldots, (Y_{m'} <: D'_{m'}).\pi'_{n'+m'}|\!) \preceq \Sigma_{21}$,

28) $\Delta; \Gamma; \{\mathsf{u} : \pi'_k\} \vdash \mathsf{e}'_k : \mathsf{t}'_2$ for all $k \in \{1, \ldots, n'\}$,

29) $\Delta; \Gamma; \{\mathsf{u} : \pi_{n'+l}\} \vdash \mathsf{d}_l : \mathsf{t}'_2$ for all $l \in \{1, \ldots, m'\}$.

We use the shorthand

30) $\check{\pi} = ?(\!|(X_1 <: C'_1).\pi^\flat_1, \ldots, (X_{n'} <: C'_{n'}).\pi^\flat_{n'},$
$(Y_1 <: D'_1).\pi'_{n'+1}, \ldots, (Y_{m'} <: D'_{m'}).\pi'_{n'+m'}|\!)$, where

31) $\pi^\flat_k = \pi'_k.?(\!|((X_1 <: C'_1).\pi'_1, \ldots, (X_{n'} <: C'_{n'}).\pi'_{n'})^*,$
$(Y_1 <: D'_1).\pi'_{n'+1}, \ldots, (Y_{m'} <: D'_{m'}).\pi'_{n'+m'}|\!)$ for $k \in \{1, \ldots, n'\}$.

Then, by arguments similar to those used to get 26), we obtain that:

32) $\Delta; \Gamma; \Sigma_{21}[\mathsf{c} \mapsto \check{\pi}] \circ \Sigma_{22} \vdash E_2[\check{\mathsf{e}}\,] : \mathsf{t}_2$.

The aim of the next steps is to show that the type of c *used to type* $\hat{\mathsf{e}}$ *is dual to that used to type* $\check{\mathsf{e}}$*, and that the parallel composition of the session environments used to type* $E_1[\hat{\mathsf{e}}\,]$ *and* $E_2[\check{\mathsf{e}}\,]$ *is the same as* Σ.

Because of 16), 27), being $\Sigma_{11} \circ \Sigma_{12} \| \Sigma_{21} \circ \Sigma_{22}$ defined, and by Lemma A.5(9a) we obtain that:

33) $!(\!|(C_1.\pi_1, \ldots, C_n.\pi_n)^*, D_1.\pi_{n+1}, \ldots, D_m.\pi_{n+m}|\!) \bowtie$
$?(\!|((X_1 <: C'_1).\pi'_1, \ldots, (X_{n'} <: C'_{n'}).\pi'_{n'})^*, (Y_1 <: D'_1).\pi'_{n'+1}, \ldots, (Y_{m'} <: D'_{m'}).\pi'_{n'+m'}|\!)$,

which implies, by definition of the duality relation:

34) $\hat{\pi} \bowtie \check{\pi}$.

Therefore, by Lemma A.5(9b) and using 11), 16), 27), 8) we obtain that:

35) $\Sigma_{11}[\mathsf{c} \mapsto \hat{\pi}] \circ \Sigma_{12} \parallel \Sigma_{21}[\mathsf{c} \mapsto \check{\pi}] \circ \Sigma_{22} \;=\; \Sigma_{11} \circ \Sigma_{12} \parallel \Sigma_{21} \circ \Sigma_{22} = \Sigma$.

The case concludes by applying rules **Par** *and* **Start** *to 26), 32), and taking into account 35) and 1), 2).*