Asynchronous Session Types and Progress for Object Oriented Languages[⋆]

Mario Coppo[1], Mariangiola Dezani-Ciancaglini[1], and Nobuko Yoshida[2]

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

Abstract. A session type is an abstraction of a sequence of heterogeneous values sent over one channel between two communicating processes. Session types have been introduced to guarantee consistency of the exchanged data and, more recently, *progress* of the session, i.e. the property that once a communication has been established, well-formed programs will never starve at communication points. A relevant feature which influences progress is whether the communication is synchronous or asynchronous. In this paper, we first formulate a typed asynchronous multi-threaded object-oriented language with thread spawning, iterative and higher order sessions. Then we study its progress through a new effect system. As far as we know, ours is the first session type system which assures progress in asynchronous communication.

1 Introduction

Distributed and concurrent programming paradigms are increasingly interesting, owing to the huge amount of distributed applications and services spread on the Internet. This gives a strong motivation to the study of specifications and implementations of these programs together with techniques for the formal verification of their properties. One of the crucial aspects is that of protocol specification: this consists in checking the coherence and safety of sequences of message interchanges that take place between a number of parties cooperating in carrying out some specific task. The use of type systems to formalise this kind of protocols has interested many researchers: in particular, *session types* [16, 25] are recently focussed as a promising type discipline for structuring hand-shake communications. Interaction between processes is achieved by specifying corresponding sequences of messages through private channels. Such sequences are associated with session types, that assures that the two parties at each end of a channel perform consistent and complementary actions. Session types are assigned to communication channels and are shared among processes. For example, the session type `begin.?int.!bool.end` expresses that, after beginning the session, (`begin`), an integer will be received (`?int`), then a boolean value will be sent (`!bool`), and finally it is closed (`end`).

[⋆] This work was partly funded by FP6-2004-510996 Coordination Action TYPES, EPSRC GR/T03208, EPSRC GR/S55538, EPSRC GR/T04724, EPSRC GR/S68071, and EU IST–2005–015905 MOBIUS project.

Session types have been studied for several different settings, *i.e.*, for π-calculus-based formalisms [2, 11, 14, 16, 25], for CORBA [26], for functional languages [13, 27], for boxed ambients [10], and recently, for CDL, a W3C standard description language for Web Services [3, 4, 17, 24, 28]. In [6] the notion of session types was investigated in the framework of object oriented languages. Such an integration has been attempted before only in [7, 26] and more recently in [5].

The integration of type-safe communication patterns with object-oriented programming idioms is done in [6] via the language MOOSE, a multi-threaded object-oriented core language augmented with session types. The type system of MOOSE has been designed not only to assure the type safety of the communication protocols, but also the *progress* property, i.e. that a communication session, when started, is executed without risk that processes in a session are blocked in a deadlock state. The first property is a consequence of the *subject reduction* property, which has been shown to be a critical one for calculi involving session types. A recent article [29] analyses this issue in details, comparing different reduction rules and typing systems appeared in the literature [2, 11, 16, 27].

The progress property, which also is an essential requirement for all kinds of applications, does not seem to have been considered before [6, 7] in the literature. The operational semantics of MOOSE, however, requires communications on a channel to be synchronous, i.e. they can take place only when both processes involved in a communication are ready to perform the corresponding action. This is a strong requirement that can sometime generate deadlocks. Take for instance the parallel of the following processes:

```
connect c0 begin.?int.end{
    connect c1 begin.!int.end{
        c0.send(3);
        c1.receive
        }
    }
```

Q_0

```
connect c0 begin.!int.end{
   connect c1 begin.?int.end{
        c1.send(5)
        };
   c0.receive
}
```

Q_1

Here connect c0 opens the session over channel c0, c0.send(3) sends value 3 via c0, and c0.receive receives a value via c0. These two processes in parallel, after having opened one connection on channel c0 and one on channel c1, cannot mutually exchange an integer on these channels. The resulting process would be stuck with the reduction rules of [6], since Q_0 and Q_1 are both waiting for a receiving action to synchronise.

In this paper we consider an *asynchronous* version of MOOSE, named AMOOSE: channels are buffered and can perform input and output actions at different times. This extension allows *the senders to send messages without being blocked*, reducing an overhead waiting for heavy synchronisation which the original synchronous session types require. Session types with asynchronous communication over buffered channels have been considered in [12, 22] for functional languages, and in [9] for operating system services, to enforce efficient and safe message exchanges. These papers do not consider the progress property. In Java, this asynchronous semantics is found in many communication APIs such as Socket [19] and NIO [20]. Further, with the asynchrony, we naturally

$$
\begin{array}{lrl}
\text{(type)} & \mathsf{t} ::= & C \mid \mathsf{bool} \mid \mathsf{s} \mid (\mathsf{s},\bar{\mathsf{s}}) \\
\text{(class)} & \mathit{class} ::= & \mathsf{class}\ C\ \mathsf{extends}\ C\ \{\ \tilde{\mathsf{f}}\,\tilde{\mathsf{t}}\ \ \widetilde{\mathit{meth}}\ \} \\
\text{(method)} & \mathit{meth} ::= & \mathsf{t}\,\mathsf{m}\ (\tilde{\mathsf{t}}\,\tilde{\mathsf{x}},\tilde{\rho}\,\tilde{\mathsf{y}})\ \{\mathsf{e}\} \\
\text{(expression)} & \mathsf{e} ::= & \mathsf{x} \mid \mathsf{v} \mid \mathsf{this} \mid \mathsf{e};\mathsf{e} \mid \mathsf{e}.\mathsf{f} := \mathsf{e} \mid \mathsf{e}.\mathsf{f} \mid \mathsf{e}.\mathsf{m}(\tilde{\mathsf{e}}) \mid \mathsf{new}\ C \\
 & \mid & \mathsf{new}\ (\mathsf{s},\bar{\mathsf{s}}) \mid \mathsf{NullExc} \mid \mathsf{spawn}\{\ \mathsf{e}\ \} \mid \mathsf{connect}\ \mathsf{a}\ \mathsf{s}\ \{\mathsf{e}\} \\
 & \mid & \mathsf{u}.\mathsf{receive} \mid \mathsf{u}.\mathsf{send}(\mathsf{e}) \mid \mathsf{u}.\mathsf{receiveS}(\mathsf{x})\{\mathsf{e}\} \mid \mathsf{u}.\mathsf{sendS}(\mathsf{u}) \\
 & \mid & \mathsf{u}.\mathsf{receiveIf}\{\mathsf{e}\}\{\mathsf{e}\} \mid \mathsf{u}.\mathsf{sendIf}(\mathsf{e})\{\mathsf{e}\}\{\mathsf{e}\} \\
 & \mid & \mathsf{u}.\mathsf{receiveWhile}\{\mathsf{e}\} \mid \mathsf{u}.\mathsf{sendWhile}(\mathsf{e})\{\mathsf{e}\} \\
\text{(identifier)} & \mathsf{a} ::= & \mathsf{c} \mid \mathsf{x} \\
\text{(channel)} & \mathsf{u} ::= & \mathsf{a} \mid \mathsf{k}^{+} \mid \mathsf{k}^{-} \\
\text{(value)} & \mathsf{v} ::= & \mathsf{c} \mid \mathsf{null} \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{o} \mid \mathsf{k}^{+} \mid \mathsf{k}^{-} \\
\text{(thread)} & P ::= & \mathsf{e} \mid P|P \\
\text{(heap)} & h ::= & [\,] \mid h::[\mathsf{o} \mapsto (C,\tilde{\mathsf{f}}:\tilde{\mathsf{v}})] \mid h::\mathsf{c} \mid h::[\mathsf{k}^{p} \mapsto \tilde{\mathsf{v}}]
\end{array}
$$

Fig. 1. Syntax, where syntax occurring only at runtime appears shaded.

obtain more programs with progress: in the above example, for instance, the sending actions transmit the output values to the buffered channels and running of Q_0 and Q_1 in parallel can progress and reach safely its natural end.

In [6] a single type system was defined to assure both type safety and progress. These two properties, however, are rather orthogonal: there seems to be no strong connection between them. In this paper we have chosen to define a type system for type safety and an effect system for progress. This de-coupling results in simpler systems and it allows a better understanding of the conditions needed to assure each property.

Structure of the paper. The syntax and operational semantics of AMOOSE will be introduced in Section 2, the typing system and the main definitions to formulate the subject reduction property will be introduced in Section 3. Progress properties will be discussed in Section 4. The complete proof of the subject reduction theorem is given in the Appendix.

2 Syntax and Operational Semantics

2.1 Syntax

In Fig. 1 we describe the syntax of AMOOSE, which is essentially that of the language MOOSE [6]; AMOOSE and MOOSE differ in the operational semantics, since in AMOOSE output is asynchronous, and the exchange of data between processes is realised via buffers in the queues associated to channels. 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*. They both can be parameters of procedures. We deviate from [6] introducing polarised live channels [11, 29].

Shared channels are only used to decide if two threads can communicate. After a connection is established the shared channel is replaced by a couple of fresh *live* channels having a different *polarity*, $+$ or $-$, one for each of the communicating threads. We denote by k^p in the same thread both the receiving channel of polarity p and the sending channel of opposite polarity $\bar{p}$: this will be clear from the operational semantics. Note that the meaning of polarities is different from that in [11], where polarities simply represent the two ends of a (unique) session channel. As a notational convention we will always use $\mathsf{c},\ldots$ to denote shared channels and $\mathsf{k}^p, \mathsf{k}^p_0, \mathsf{k}^p_1, \ldots$ to denote polarised live channels.

User syntax The metavariable t ranges over types for expressions, ρ ranges over running session types, C ranges over class names and s ranges over shared session types. Each session type s has one corresponding *dual*, denoted $\bar{\mathsf{s}}$, which is obtained by replacing each ! (output) by ? (input) and vice versa. We introduce the full syntax of types in § 3. Class and method declarations are as usual.

The syntax of user expressions e,e' is standard but for the channel constructor $\mathsf{new}\ (\mathsf{s},\bar{\mathsf{s}})$, 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 gives parameter, value, the self identifier this, sequence of expressions, assignment to fields, field access, method call, and object creation. The values are channels, null, and the literals true and false. Thread creation is declared using $\mathsf{spawn}\{\ \mathsf{e}\ \}$, in which the expression e is called the *thread body*. The expression $\mathsf{connect}\ \mathsf{u}\ \mathsf{s}\{\mathsf{e}\}$ starts a session: the channel u appears within the term $\{\mathsf{e}\}$ in session communications that agree with session type s. The remaining eight expressions, which realise the exchanges of data, are called *session expressions*, and start with "$\mathsf{u}.\_$"; we call u the *subject* of such expressions. In the below explanation session expressions are pairwise coupled: we say that expressions in the same pair and with the same subject are *dual* to each other.

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

Runtime syntax The runtime syntax (shown shaded in Fig. 1) extends the user syntax: it adds $\mathsf{NullExc}$ to expressions, denoting the null pointer error; includes polarised live channels; extends values to allow for object identifiers o, which denote references to instances of classes; finally, introduces threads running in parallel. Single and multiple *threads* are ranged over by P, P'. The expression $P \,|\, P'$ says that P and P' are running in parallel.

Heaps, ranged over h, are built inductively using the heap composition operator '::', and contain mappings of object identifiers to instances of classes, shared channels and mappings of polarised channels to queues of values. 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 shared channel c. The heap produced by composing $h :: [\mathsf{k}^p \mapsto \tilde{\mathsf{v}}]$ will map the live channel k^p to the queue $\tilde{\mathsf{v}}$. Heap membership for object identifiers and channels is checked using standard set notation, we therefore write it as $\mathsf{o} \in h$, $\mathsf{c} \in h$, and $\mathsf{k}^p \in h$. Heap update for objects is written $h[\mathsf{o} \mapsto (C, \tilde{\mathsf{f}} : \tilde{\mathsf{v}})]$, for polarised channels $h[\mathsf{k}^p \mapsto \tilde{\mathsf{v}}]$, and field update is written $(C, \tilde{\mathsf{f}} : \tilde{\mathsf{v}})[\mathsf{f} \mapsto \mathsf{v}]$. We assume that the heap is unordered, i.e. satisfying equivalences like

$$\upsilon :: \upsilon' \equiv \upsilon' :: \upsilon, \qquad h_1 \equiv h_1', h_2 \equiv h_2' \Rightarrow h_1 :: h_2 \equiv h_1' :: h_2'$$

where υ denotes generic heap elements. With some abuse of notation the operator " :: " denotes heap concatenation without making distinction between heaps and heap elements.

2.2 Operational Semantics

This subsection presents the operational semantics of AMOOSE: the main difference with respect to [6] is that in AMOOSE output is asynchronous and the values are exchanged through queues associated to live channels in the heap. In the reduction rules then the heap plays an essential role also in communications.

We only discuss the more interesting rules. First we list the evaluation contexts.

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

Since heaps associate queues only to live channels, we can reduce only session expressions whose subjects are live channels. Moreover shared channels are sent by send, while live channels are sent by sendS. For this reason there are no evaluation contexts of the shapes $E.\mathsf{send}(\mathsf{e})$, $\mathsf{k}^p.\mathsf{sendS}(E)$ etc.

Fig. 2 defines auxiliary functions used in the operational semantics and typing rules. We assume a fixed, global class table CT, which as usual contains *Object* as topmost class.

Expressions Fig. 3 shows the rules for execution of expressions which correspond to the sequential part of the language. The rules not involving communications are standard [1, 8, 18], but for the addition of a fresh shared channel to the heap (rule $\mathbf{NewS}^{\rightarrow}$). In rule $\mathbf{NewC}^{\rightarrow}$ the auxiliary function $\mathsf{fields}(C)$ examines the class table and returns the field declarations for C. The method invocation rule is $\mathbf{Meth}^{\rightarrow}$; the auxiliary function

Field lookup

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

Method lookup

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

Method type lookup

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

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

Method body lookup

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

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

τ is either $\mathtt{t}$ or ρ.

Fig. 2. Lookup Functions.

$\mathsf{mbody}(\mathtt{m}, C)$ looks up $\mathtt{m}$ 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 $\mathtt{o}$, and the formal parameters $\tilde{\mathtt{x}}$ are replaced by the actual parameters $\tilde{\mathtt{v}}$.

The operator ":" denotes queue concatenation without making distinction between elements and queues. Thus $\mathtt{v} : \tilde{\mathtt{v}}'$ denotes a queue beginning with $\mathtt{v}$ and $\tilde{\mathtt{v}}' : \mathtt{v}$ a queue ending with $\mathtt{v}$.

The send communication rules put values in the queues associated to the live channels with the same names and opposite polarity of the expression subjects. The receive communication rules instead take values in the queues associated to the expression subjects. In rules **SendS**$^{\to}$ and **ReceiveS**$^{\to}$ standard values are exchanged from expressions to queues. Rule **SendSS**$^{\to}$ puts a live channel in the queue: the opposite rule **ReceiveSS**$^{\to}$ (see Fig. 4) is discussed below since it spawns a new thread. In the conditional rules (**SendSIf-true**$^{\to}$, **SendSIf-false**$^{\to}$, **ReceiveSIf-true**$^{\to}$, **ReceiveSIf-false**$^{\to}$) depending on the value of the boolean, the execution proceeds with either the first or the second branch. The iterative rules (**SendSWhile**$^{\to}$, **ReceiveSWhile**$^{\to}$) simply express the iteration by means of the conditional.

Standard Reduction

Fld$^{\rightarrow}$
$$\frac{h(\mathsf{o}) = (C, \tilde{\mathsf{f}} : \tilde{\mathsf{v}})}{\mathsf{o}.\mathsf{f}_i, h \longrightarrow \mathsf{v}_i, h}$$

Seq$^{\rightarrow}$
$$\mathsf{v}; \mathsf{e}, h \longrightarrow \mathsf{e}, h$$

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

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

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

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

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

NullProp$^{\rightarrow}$
$$E[\mathsf{NullExc}], h \longrightarrow \mathsf{NullExc}, h$$

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

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

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

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

Asynchronous Communication Reduction

SendS$^{\rightarrow}$
$$\mathsf{k}^p.\mathsf{send}(\mathsf{v}), h :: [\mathsf{k}^{\bar{p}} \mapsto \tilde{\mathsf{v}}'] \longrightarrow \mathsf{null}, h :: [\mathsf{k}^{\bar{p}} \mapsto \tilde{\mathsf{v}}' : \mathsf{v}]$$

ReceiveS$^{\rightarrow}$
$$\mathsf{k}^p.\mathsf{receive}, h :: [\mathsf{k}^p \mapsto \mathsf{v} : \tilde{\mathsf{v}}'] \longrightarrow \mathsf{v}, h :: [\mathsf{k}^p \mapsto \tilde{\mathsf{v}}']$$

SendSS$^{\rightarrow}$
$$\mathsf{k}^p.\mathsf{sendS}(\mathsf{k}_0^q), h :: [\mathsf{k}^{\bar{p}} \mapsto \tilde{\mathsf{v}}'] \longrightarrow \mathsf{null}, h :: [\mathsf{k}^{\bar{p}} \mapsto \tilde{\mathsf{v}}' : \mathsf{k}_0^q]$$

SendSIf-true$^{\rightarrow}$
$$\mathsf{k}^p.\mathsf{sendIf}(\mathsf{true})\{\mathsf{e}_1\}\{\mathsf{e}_2\}, h :: [\mathsf{k}^{\bar{p}} \mapsto \tilde{\mathsf{v}}] \longrightarrow \mathsf{e}_1, h :: [\mathsf{k}^{\bar{p}} \mapsto \tilde{\mathsf{v}} : \mathsf{true}]$$

SendSIf-false$^{\rightarrow}$
$$\mathsf{k}^p.\mathsf{sendIf}(\mathsf{false})\{\mathsf{e}_1\}\{\mathsf{e}_2\}, h :: [\mathsf{k}^{\bar{p}} \mapsto \tilde{\mathsf{v}}] \longrightarrow \mathsf{e}_2, h :: [\mathsf{k}^{\bar{p}} \mapsto \tilde{\mathsf{v}} : \mathsf{false}]$$

ReceiveSIf-true$^{\rightarrow}$
$$\mathsf{k}^p.\mathsf{receiveIf}\{\mathsf{e}_1\}\{\mathsf{e}_2\}, h :: [\mathsf{k}^p \mapsto \mathsf{true} : \tilde{\mathsf{v}}] \longrightarrow \mathsf{e}_1, h :: [\mathsf{k}^p \mapsto \tilde{\mathsf{v}}]$$

ReceiveSIf-false$^{\rightarrow}$
$$\mathsf{k}^p.\mathsf{receiveIf}\{\mathsf{e}_1\}\{\mathsf{e}_2\}, h :: [\mathsf{k}^p \mapsto \mathsf{false} : \tilde{\mathsf{v}}] \longrightarrow \mathsf{e}_2, h :: [\mathsf{k}^p \mapsto \tilde{\mathsf{v}}]$$

SendSWhile$^{\rightarrow}$
$$\mathsf{k}^p.\mathsf{sendWhile}(\mathsf{e})\{\mathsf{e}_1\}, h \longrightarrow \mathsf{k}^p.\mathsf{sendIf}(\mathsf{e})\{\mathsf{e}_1; \mathsf{k}^p.\mathsf{sendWhile}(\mathsf{e})\{\mathsf{e}_1\}\}\{\mathsf{null}\}, h$$

ReceiveSWhile$^{\rightarrow}$
$$\mathsf{k}^p.\mathsf{receiveWhile}\{\mathsf{e}\}, h \longrightarrow \mathsf{k}^p.\mathsf{receiveIf}\{\mathsf{e}; \mathsf{k}^p.\mathsf{receiveWhile}\{\mathsf{e}\}\}\{\mathsf{null}\}, h$$

Fig. 3. Expression Reduction.

Struct

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

Spawn$^{\rightarrow}$

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

Par$^{\rightarrow}$

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

Str$^{\rightarrow}$

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

Connect$^{\rightarrow}$

$$E_1[\mathsf{connect\ c\ s}\{\mathsf{e}_1\}]\,|\,E_2[\mathsf{connect\ c}\ \bar{\mathsf{s}}\{\mathsf{e}_2\}],\ h$$
$$\longrightarrow\ E_1[\mathsf{e}_1[{}^{\mathsf{k}^+}\!/\mathsf{c}]]\,|\,E_2[\mathsf{e}_2[{}^{\mathsf{k}^-}\!/\mathsf{c}]],\ h :: [\mathsf{k}^+ \mapsto \varepsilon] :: [\mathsf{k}^- \mapsto \varepsilon] \quad \mathsf{k}^+, \mathsf{k}^- \notin h$$

ReceiveSS$^{\rightarrow}$

$$E[\mathsf{k}^p.\mathsf{receiveS}\,(\mathsf{x})\{\mathsf{e}\}], h :: [\mathsf{k}^p \mapsto \mathsf{k}_0^q : \tilde{\mathsf{v}}] \ \longrightarrow\ \mathsf{e}[{}^{\mathsf{k}_0^q}\!/\mathsf{x}]\ |\ E[\mathsf{null}], h :: [\mathsf{k}^p \mapsto \tilde{\mathsf{v}}]$$

Fig. 4. Thread Reduction.

An *elementary* expression reduction is a reduction defined by any of the expression reduction rules except rule **Cong$^{\rightarrow}$**.

Threads The reduction rules for threads, shown in Fig. 4, are given modulo the standard structural equivalence rules of the π-calculus [21], written $\equiv$. We define *multi-step* reduction as: $\twoheadrightarrow \stackrel{\mathsf{def}}{=} (\longrightarrow \cup \equiv)^*$.

When $\mathsf{spawn}\,\{\ \mathsf{e}\ \}$ is the active redex within an arbitrary evaluation context, the *thread body* e becomes a new thread, and the original spawn expression is replaced by null in the context. This is expressed by rule **Spawn$^{\rightarrow}$**.

Rule **Connect$^{\rightarrow}$** describes the opening of sessions: if two threads require a session on the same shared channel name c with dual session types, then two new fresh live channels k^+ and k^- with the same name but opposite polarities are created and added to the heap with empty queues. The freshness of the name k 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 channels k^+ and k^-, respectively.

In rule **ReceiveSS$^{\rightarrow}$** one thread awaits to receive a live channel, which will be bound to the variable x within the expression e. Notice that the receiver spawns a new thread to handle the consumption of the delegated session. This strategy avoids deadlocks in the presence of circular paths of session delegation [6].

We say that a heap h is *balanced* if $\mathsf{k}^p \in h$ implies $\mathsf{k}^{\bar{p}} \in h$. We only consider balanced heaps: it is easy to verify that reduction rules preserve balance of heaps.

Proposition 2.1. *If $P, h \longrightarrow P', h'$ and h is balanced, then h' is balanced too.*

3 The Type Assignment System and its Properties

The type system discussed in this section is designed to guarantee linearity of live channels and communication error freedom. These properties are consequences of the Subject Reduction Theorem. Instead this system does not assure progress, which we will consider in next section.

3.1 Types

The full syntax of types is given in Fig. 5.

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

Fig. 5. Syntax of types.

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

The *conditional* partial session type has the shape $\dagger\langle\pi_1,\pi_2\rangle$. When $\dagger$ is $!$, $\dagger\langle\pi_1,\pi_2\rangle$ describes sessions which send a boolean value and proceed with π_1 if the value is true, or π_2 if the value is false; when $\dagger$ is $?$, the behaviour is the same, except that the boolean that determines the branch is to be received instead. The *iterative* partial session type $\dagger\langle\pi\rangle^{*}$ describes sessions that respectively send or receive a boolean value, and if that value is true continue with π, *iterating*, while if the value is false, do nothing.

The partial session types $!(\eta)$ and $?(\eta)$ represent the exchange of a live channel, and therefore of an active session, with remaining communications determined by the ended session type η. Note that typing the live channel by η instead of π ensures that this channel is no longer used in the sending thread.

An *ended session type*, η, is a partial session type concatenated either with end or with a conditional whose branches in turn are both ended session types. It expresses a sequence of communications with its termination, *i.e.*, no further communications on that channel are allowed at the end. A conditional ended session type allows to type spawns or connects in the branches.

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

A *shared session type*, s, starts with the keyword begin and has one or more end-points, denoted by end. Between the start and each ending point, a sequence of session parts describe the communication protocol.

A *session type* θ is a running session type or a shared session type.

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

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

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

Note that $\theta = \overline{\theta'}$ if and only if $\theta' = \overline{\theta}$.

We type expressions and threads with respect to the global class table $\mathtt{CT}$, as reflected in the rules of Fig. 6 which define well-formed standard types. By $\mathsf{dom}(\mathtt{CT})$ we denote the domain of the class table $\mathtt{CT}$, *i.e.*, the set of classes declared in $\mathtt{CT}$. In Fig. 6 we also define subtyping, $<:$, on class names: we assume that the subclassing is acyclic as in [18]. In addition, we have $(\mathsf{s},\overline{\mathsf{s}}) <: \mathsf{s}$ and $(\mathsf{s},\overline{\mathsf{s}}) <: \overline{\mathsf{s}}$, as in standard π-calculus channel subtyping rules [15]: a channel on which both communication directions are allowed may also transmit data following only one of the two directions.

3.2 Typing Rules

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

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

where the *standard environment* Γ associates standard types to this, parameters and objects, while the *session environment* Σ contains only judgements for channel names and variables. Fig. 6 defines well-formedness of standard and session environments, where the domain of an environment is defined as usual and denoted by $\mathsf{dom}()$.

In Fig. 7, Fig. 8 and Fig. 9 we give the typing rules for expressions and threads. In the typing rules for 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 running session types since we want to avoid to have meaningless ε. As usual, $\perp$ stands for undefined.

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

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

Well-formed Standard Types

Class $\dfrac{C \in \mathsf{dom}(\mathtt{CT})}{\vdash C : \mathtt{tp}}$ **Wf-Session** $\dfrac{}{\vdash \mathsf{s} : \mathtt{tp}}$ **Pair** $\dfrac{}{\vdash (\mathsf{s}, \bar{\mathsf{s}}) : \mathtt{tp}}$ **Bool** $\dfrac{}{\vdash \mathsf{bool} : \mathtt{tp}}$

Subtyping

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

Standard Environments and Well-formed Standard Environments

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

Emp $\dfrac{}{\emptyset \vdash \mathsf{ok}}$ **EVar** $\dfrac{\vdash \mathsf{t} : \mathtt{tp} \quad \mathsf{x} \notin \mathsf{dom}(\Gamma)}{\Gamma, \mathsf{x} : \mathsf{t} \vdash \mathsf{ok}}$ **EOid** $\dfrac{C \in \mathsf{dom}(\mathtt{CT}) \quad \mathsf{o} \notin \mathsf{dom}(\Gamma)}{\Gamma, \mathsf{o} : C \vdash \mathsf{ok}}$ **Ethis** $\dfrac{C \in \mathsf{dom}(\mathtt{CT}) \quad \mathsf{this} \notin \mathsf{dom}(\Gamma)}{\Gamma, \mathsf{this} : C \vdash \mathsf{ok}}$

Session Environments and Well-formed Session Environments

$$\Sigma \;::=\; \emptyset \mid \Sigma,\, \mathsf{u} : \rho \qquad \textbf{SEmp}\; \dfrac{}{\emptyset \vdash \mathsf{ok}} \qquad \textbf{SERC}\; \dfrac{\mathsf{u} \notin \mathsf{dom}(\Sigma)}{\Sigma,\, \mathsf{u} : \rho \vdash \mathsf{ok}}$$

Fig. 6. Standard Types, Subtyping, and Environments.

– $\Sigma \circ \Sigma' \;=\; \begin{cases} \Sigma \setminus \Sigma' \;\cup\; \Sigma' \setminus \Sigma \;\cup\; \{\mathsf{u} : \Sigma(\mathsf{u}) \circ \Sigma'(\mathsf{u}) \;\mid\; \mathsf{u} \in \mathsf{dom}(\Sigma) \cap \mathsf{dom}(\Sigma')\} \\ \qquad\qquad \text{if} \quad \forall \mathsf{u} \in \mathsf{dom}(\Sigma) \cap \mathsf{dom}(\Sigma') :\; \Sigma(\mathsf{u}) \circ \Sigma'(\mathsf{u}) \neq \bot; \\ \bot \quad \text{otherwise.} \end{cases}$

The concatenation of two running session types ρ and ρ' is the unique running session 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. The extension to session environments is straightforward. The typing rules concatenate the session environments to take into account the order of execution of expressions. We adopt the convention that typing rules are applicable only when the session environments in the conclusions are defined.

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

Rule **Spawn** requires that all sessions used by the spawned thread are finally consumed, *i.e.*, they are all ended session types. This is necessary in order to avoid configurations that break the bilinearity condition. The consumption is guaranteed by the condition $\mathit{ended}(\Sigma)$, since we define:

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

Typing Rules for Values

$$\textbf{Null}\ \frac{\Gamma \vdash \mathsf{ok} \quad \vdash \mathsf{t} : \mathtt{tp}}{\Gamma;\emptyset \vdash \mathsf{null} : \mathsf{t}} \qquad \textbf{Oid}\ \frac{\Gamma,\mathsf{o} : C \vdash \mathsf{ok}}{\Gamma,\mathsf{o} : C;\emptyset \vdash \mathsf{o} : C} \qquad \textbf{True}\ \frac{\Gamma \vdash \mathsf{ok}}{\Gamma;\emptyset \vdash \mathsf{true} : \mathsf{bool}} \qquad \textbf{False}\ \frac{\Gamma \vdash \mathsf{ok}}{\Gamma;\emptyset \vdash \mathsf{false} : \mathsf{bool}} \qquad \textbf{Chan}\ \frac{\Gamma \vdash \mathsf{ok}}{\Gamma;\emptyset \vdash \mathsf{c} : \mathsf{s}}$$

Typing Rules for Standard Expressions

$$\textbf{Var}\ \frac{\Gamma,\mathsf{x} : \mathsf{t} \vdash \mathsf{ok}}{\Gamma,\mathsf{x} : \mathsf{t} \vdash \mathsf{x} : \mathsf{t}} \qquad \textbf{This}\ \frac{\Gamma,\mathsf{this} : C \vdash \mathsf{ok}}{\Gamma,\mathsf{this} : C \vdash \mathsf{this} : C} \qquad \textbf{Fld}\ \frac{\Gamma;\Sigma \vdash \mathsf{e} : C \quad \mathsf{f}\,\mathsf{t} \in \mathsf{fields}(C)}{\Gamma;\Sigma \vdash \mathsf{e.f} : \mathsf{t}}$$

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

$$\textbf{NewC}\ \frac{\Gamma \vdash \mathsf{ok} \quad C \in \mathsf{dom}(\mathtt{CT})}{\Gamma;\emptyset \vdash \mathsf{new}\ C : C} \qquad \textbf{NewS}\ \frac{\Gamma \vdash \mathsf{ok}}{\Gamma;\emptyset \vdash \mathsf{new}\ (\mathsf{s},\bar{\mathsf{s}}) : (\mathsf{s},\bar{\mathsf{s}})} \qquad \textbf{Spawn}\ \frac{\Gamma;\Sigma \vdash \mathsf{e} : \mathsf{t} \quad \mathit{ended}(\Sigma)}{\Gamma;\Sigma \vdash \mathsf{spawn}\{\ \mathsf{e}\ \} : \mathit{Object}} \qquad \textbf{NullPE}\ \frac{\Gamma \vdash \mathsf{ok} \quad \vdash \mathsf{t} : \mathtt{tp}}{\Gamma;\emptyset \vdash \mathsf{NullExc} : \mathsf{t}}$$

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

Fig. 7. Typing Rules for Expressions I.

Rule **Meth** retrieves the type of the method m from the class table using the auxiliary function $\mathsf{mtype}(\mathtt{m},C)$, defined in Fig. 2. The session environments of the premises are concatenated with $\{\mathsf{u}_1 : \rho_1,\ldots,\mathsf{u}_m : \rho_m\}$, which represents the communication protocols of the live channels $\mathsf{u}_1,\ldots,\mathsf{u}_m$ during the execution of the method body.

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

In rules **ReceiveIF** and **SendIF** *both* ρ_1 and ρ_2 are either partial session types or ended session types – this is guaranteed by the syntax of conditional session types.

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

Fig. 10 defines well-formed class tables. Rule **M-ok** type-checks the method bodies with respect to a class C taking as environments the association between formal parameters and their types and the association between this and C.

Typing Rules for Communication Expressions

Conn
$$\frac{\Gamma \vdash \mathsf{a} : \mathsf{begin}.\eta \quad \Gamma \backslash \mathsf{a} \,;\, \Sigma, \mathsf{a} : \eta \vdash \mathsf{e} : \mathsf{t}}{\Gamma; \Sigma \vdash \mathsf{connect\ a\ begin}.\eta\, \{\mathsf{e}\} : \mathsf{t}}$$

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

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

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

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

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

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

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

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

Non-structural Typing Rules for Expressions

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

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

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

Fig. 8. Typing Rules for Expressions II.

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

Par
$$\frac{\Gamma; \Sigma_i \vdash P_i : \mathsf{thread} \quad (i = 1,2)}{\Gamma; \Sigma_1 \cup \Sigma_2 \vdash P_1 \,|\, P_2 : \mathsf{thread}}$$

Fig. 9. Typing Rules for Threads.

3.3 Subject Reduction

We will consider only reductions of well-typed expressions and threads. We define types of run time entities in the standard way. The judgment is defined in Fig. 11. The judgment $h \vdash \mathsf{v} : \mathsf{t}$ guarantees that the runtime value v has type t; for objects we take subclasses into consideration in rule **HObjSubs**. The judgment $h \vdash \mathsf{o}$ guarantees that

M-ok
$$\frac{\{\mathtt{this}:C,\tilde{\mathtt{x}}:\tilde{\mathtt{t}}\};\ \{\tilde{\mathtt{y}}:\tilde{\rho}\}\vdash \mathtt{e}:\mathtt{t}}{\mathtt{t}\ \mathtt{m}\ (\tilde{\mathtt{t}}\ \tilde{\mathtt{x}},\tilde{\rho}\,\tilde{\mathtt{y}})\ \{\mathtt{e}\}:\mathtt{ok}\ \mathtt{in}\ C}$$

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

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

Fig. 10. Well-formed Class Tables.

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

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

HNull
$$\frac{C\in \mathsf{dom}(\mathtt{CT})}{h\vdash \mathsf{null}:C}$$

HObj
$$\frac{h(\mathsf{o})=(C,\dots)}{h\vdash \mathsf{o}:C}$$

HObjSubs
$$\frac{h\vdash \mathsf{o}:C'\qquad C'<:C}{h\vdash \mathsf{o}:C}$$

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

Fig. 11. Types of Runtime Entities.

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. Note that in rule **HObjSubs** the equality in the first premise simply asserts that there is an object o in the heap h, while the conclusion asserts that o is well-formed.

In order to formalise agreement between session environments and heaps, it is handy to introduce some definitions. We start by determining the initial and the final parts of a running session type.

A *basic session type* (s-basic type for short) is a session type of the form $\dagger\mathtt{t}$ or $\dagger(\eta)$ or $\dagger\langle\rho_1,\rho_2\rangle$ or $\dagger\langle\pi\rangle^*$. Let β be a s-basic type. We denote with $\beta\psi$ a session type which begins with β and has the form $\beta.\rho$ or $\beta.\mathsf{end}$ or β. In these cases let us call ψ the *continuation* of $\beta\psi$. If $\beta\psi$ stands for β or $\beta.\mathsf{end}$ we say that the continuation ψ is *light*. Further let us define

$$\psi^{\diamond}=\begin{cases}\rho & \text{if } \psi \text{ stands for } .\rho,\\ \varepsilon.\mathsf{end} & \text{if } \psi \text{ stands for } .\mathsf{end}\\ \varepsilon & \text{otherwise.}\end{cases}$$

The *core domain* of a session environment Σ (notation $\mathsf{cored}(\Sigma)$) is the set of subjects in Σ whose predicates do not belong to $\{\varepsilon,\varepsilon.\mathsf{end}\}$.

The *channel range* of an heap h (notation $\mathsf{ran}_c(h)$) is the set of live channels which occur in h inside value queues:

$$\mathsf{ran}_c(h) = \{\mathsf{k}^p \mid h(\mathsf{k}^q_0) = \tilde{\mathsf{v}} \colon \mathsf{k}^p :\tilde{\mathsf{v}}' \text{ for some } \mathsf{k}^q_0, \tilde{\mathsf{v}}, \tilde{\mathsf{v}}'\}.$$

A heap h agrees with a session environment Σ if each value which is in the queue associated to a live channel k^p in h has the type expected by $\Sigma(\mathsf{k}^p)$. We formalise this by means of an inductive definition on the (sum of) the sizes of the queues associated by h to the live channels in the core domain of Σ. The base step is when all these live channels are associated to empty queues and there are no channels in the heap waiting to be activated by $\mathsf{receiveS}$ expressions. In the induction cases each top value of a queue associated to a channel is checked against the running session type of that channel in the environment. If this check fails the heap and the session environment do not agree, otherwise both the queue and the environment are updated and the check is inductively applied to the resulting heap and session environment. The induction terminates since at each step a top value in a queue is popped out. Note that, when the considered value is a channel k^q_0 of type η, we add the statement $\mathsf{k}^q_0 : \eta$ to the session environment: this is necessary to type the expression receiving the channel k^q_0. Clearly the order in which the live channels in the heap are considered is not influential.

Definition 3.1. *Fig. 12 defines the* agreement $A(\Sigma;h)$ *of a session environment* Σ *with a heap* h*.*

$$A(\Sigma;h) = \begin{cases}
\mathsf{true} & \text{if } \mathsf{dom}(\Sigma) \cap \mathsf{ran}_c(h) = \emptyset \\
 & \text{and } \forall \mathsf{k}^p \in \mathsf{cored}(\Sigma).\, h(\mathsf{k}^p) = \varepsilon \\
A(\Sigma[\mathsf{k}^p \mapsto \psi^\diamond]; h[\mathsf{k}^p \mapsto \tilde{\mathsf{v}}']) & \text{if } h(\mathsf{k}^p) = \mathsf{v} : \tilde{\mathsf{v}}', \mathsf{v} \in \{\mathsf{true}, \mathsf{false}\} \\
 & \text{and } \Sigma(\mathsf{k}^p) = ?\mathsf{bool}\,\psi \\
A(\Sigma[\mathsf{k}^p \mapsto \psi^\diamond]; h[\mathsf{k}^p \mapsto \tilde{\mathsf{v}}']) & \text{if } h(\mathsf{k}^p) = \mathsf{o} : \tilde{\mathsf{v}}', h(\mathsf{o}) = (C', \tilde{\mathsf{f}}), C' <: C \\
 & \text{and } \Sigma(\mathsf{k}^p) = ?C\psi \\
A(\Sigma[\mathsf{k}^p \mapsto \psi^\diamond]; h[\mathsf{k}^p \mapsto \tilde{\mathsf{v}}']) & \text{if } h(\mathsf{k}^p) = \mathsf{c} : \tilde{\mathsf{v}}', \mathsf{t} \in \{\mathsf{s}, (\mathsf{s}, \bar{\mathsf{s}})\} \text{ and } \Sigma(\mathsf{k}^p) = ?\mathsf{t}\,\psi \\
A(\Sigma[\mathsf{k}^p \mapsto \psi^\diamond], \mathsf{k}^q_0 : \eta; h[\mathsf{k}^p \mapsto \tilde{\mathsf{v}}']) & \text{if } h(\mathsf{k}^p) = \mathsf{k}^q_0 : \tilde{\mathsf{v}}',\ \Sigma(\mathsf{k}^p) = ?(\eta)\psi \text{ and } \mathsf{k}^q_0 \notin \mathsf{dom}(\Sigma) \\
A(\Sigma[\mathsf{k}^p \mapsto \rho_1\psi^\diamond]; h[\mathsf{k}^p \mapsto \tilde{\mathsf{v}}']) & \text{if } h(\mathsf{k}^p) = \mathsf{true} : \tilde{\mathsf{v}}' \text{ and } \Sigma(\mathsf{k}^p) = ?\langle \rho_1, \rho_2 \rangle \psi \\
A(\Sigma[\mathsf{k}^p \mapsto \rho_2\psi^\diamond]; h[\mathsf{k}^p \mapsto \tilde{\mathsf{v}}']) & \text{if } h(\mathsf{k}^p) = \mathsf{false} : \tilde{\mathsf{v}}' \text{ and } \Sigma(\mathsf{k}^p) = ?\langle \rho_1, \rho_2 \rangle \psi \\
A(\Sigma[\mathsf{k}^p \mapsto \pi.!\langle \pi \rangle^* \psi^\diamond]; h[\mathsf{k}^p \mapsto \tilde{\mathsf{v}}']) & \text{if } h(\mathsf{k}^p) = \mathsf{true} : \tilde{\mathsf{v}}' \text{ and } \Sigma(\mathsf{k}^p) = ?\langle \pi \rangle^* \psi \\
A(\Sigma[\mathsf{k}^p \mapsto \psi^\diamond]; h[\mathsf{k}^p \mapsto \tilde{\mathsf{v}}']) & \text{if } h(\mathsf{k}^p) = \mathsf{false} : \tilde{\mathsf{v}}' \text{ and } \Sigma(\mathsf{k}^p) = ?\langle \pi \rangle^* \psi \\
\mathsf{false} & \text{otherwise}
\end{cases}$$

Fig. 12. Agreement between Session Environments and Heaps.

We are now able to formulate the agreement between environments and heaps though the following rule:

WfHeap

$$\frac{\forall \mathsf{o} \in \mathsf{dom}(h):\ h \vdash \mathsf{o} \qquad \forall \mathsf{o} \in \mathsf{dom}(\Gamma):\ h \vdash \mathsf{o} : \Gamma(\mathsf{o}) \qquad A(\Sigma; h)}{\Gamma; \Sigma \vdash h}$$

In the remaining of this section 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, the session environment used in the typing can be augmented by ending partial session types or by introducing ε-predicates. For example, $\Gamma; \Sigma \vdash \mathsf{x} : \mathsf{t}$ does *not* imply that $\Sigma = \emptyset$; instead, it implies that $\mathsf{cored}(\Sigma) = \emptyset$.

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 nonstructural rules.

Definition 3.2 (Weakening Order $\preceq$). $\Sigma \preceq \Sigma'$ *is the smallest partial order such that:* (1) *if* $\mathsf{u} \notin \mathsf{dom}(\Sigma)$*,* $\Sigma \preceq \Sigma, \mathsf{u} : \varepsilon$*; and* (2) $\Sigma, \mathsf{u} : \pi \preceq \Sigma, \mathsf{u} : \pi.\mathsf{end}$.

Note that $\preceq$ is defined in such a way that, if Σ is well-formed and $\Sigma \preceq \Sigma'$, then also Σ' is well-formed.

Generation Lemmas for standard expressions, communication expressions, and processes are given in the Appendix (see Lemmas A.2, A.3 and A.4) and make use of the relation $\preceq$. For example, $\Gamma; \Sigma \vdash \mathsf{u.send(e)} : \mathsf{t}$ implies $\mathsf{t} = \mathit{Object}$ and $\Gamma; \Sigma' \vdash \mathsf{e} : \mathsf{t}'$ and $\Sigma' \circ \{\mathsf{u} : !\mathsf{t}\} \preceq \Sigma$ for some Σ', t'.

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 3.3 (Weakening). *Let* $\Sigma \preceq \Sigma'$*, then*

1. $\Gamma; \Sigma \vdash \mathsf{e} : \mathsf{t}$ *implies* $\Gamma; \Sigma' \vdash \mathsf{e} : \mathsf{t}$*;*
2. $\Gamma; \Sigma \vdash P : \mathsf{thread}$ *implies* $\Gamma; \Sigma' \vdash P : \mathsf{thread}$.

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

Lemma 3.4 (Preservation of Typing under Structural Equivalence). *If* $\Gamma; \Sigma \vdash P : \mathsf{thread}$ *and* $P \equiv P'$*, then* $\Gamma; \Sigma \vdash P' : \mathsf{thread}$.

Lemma 3.5 states that the typing derivation of $E[\mathsf{e}]$ can be obtained by composing the subderivation of a typing for e, with a typing derivation for $E[\mathsf{x}]$. 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 3.5 (Subderivations). *If* $\Gamma; \Sigma \vdash E[\mathsf{e}] : \mathsf{t}$*, then there exist* Σ_1, Σ_2 *and* t' *such that* $\Sigma = \Sigma_1 \circ \Sigma_2$*, and* $\Gamma; \Sigma_1 \vdash \mathsf{e} : \mathsf{t}'$ *and* $\Gamma, \mathsf{x} : \mathsf{t}'; \Sigma_2 \vdash E[\mathsf{x}] : \mathsf{t}$*, where* x *is a fresh variable in* $E[-]$ *and* Γ.

On the other hand, Lemma 3.6 allows the combination of the typing of $E[\mathsf{x}]$ and the typing of e, provided that the contexts Σ_1 and Σ_2 used for the two typings can be composed through $\circ$, and that the type of e is the same as that of x in the first typing.

Lemma 3.6 (Context Substitution). *If* $\Gamma;\Sigma_1 \vdash \mathsf{e} : \mathsf{t}'$*, and* $\Gamma,\mathsf{x} : \mathsf{t}';\Sigma_2 \vdash E[\mathsf{x}] : \mathsf{t}$*, and* $\Sigma_1 \circ \Sigma_2$ *is defined, then* $\Gamma;\Sigma_1 \circ \Sigma_2 \vdash E[\mathsf{e}] : \mathsf{t}$.

For stating the Subject Reduction Theorem we need to introduce a partial order (called *evaluation order*) between running session types which takes into account that session types are consumed by reducing terms (Point 1). This evaluation order is also extended to pairs of session environments and heaps in two ways. The first order (Point 2) requires the types for the same live channels in the session environments are consistent through an expression reduction, i.e. that they take into account the consumed actions (first case) and, in the case that a live channel is transmitted or received, that this is correctly registered in the environment and in the heap (the other two cases). The second order (Point 3) extends the first one taking into account that new live channels can be created in a heap via evaluation of $\mathsf{connect}$ expressions.

Definition 3.7 (Evaluation Order).

1. $\sqsubseteq$ *is defined as the smallest partial order on running session types such that:* $\varepsilon \sqsubseteq \rho$*;* $\varepsilon.\mathsf{end} \sqsubseteq \eta$*;* $\pi_i \sqsubseteq \dagger\langle\pi_1,\pi_2\rangle$ *(*$i \in \{1,2\}$*);* $\eta_i \sqsubseteq \dagger\langle\eta_1,\eta_2\rangle$ *(*$i \in \{1,2\}$*);* $\dagger\langle\pi.\langle\pi\rangle^*,\varepsilon\rangle \sqsubseteq \dagger\langle\pi\rangle^*$*; and* $\pi \sqsubseteq \pi'$ *implies* $\pi \circ \rho \sqsubseteq \pi' \circ \rho$.
2. *We define* $\langle\Sigma';h'\rangle \sqsubseteq \langle\Sigma;h\rangle$ *if whenever* $\mathsf{k}^p,\ \mathsf{k}^{\bar{p}} \in h'$ *we have* $\mathsf{k}^p,\ \mathsf{k}^{\bar{p}} \in h$ *and moreover one of the following conditions is satisfied:*
   (a) $\mathsf{k}^p : \rho' \in \Sigma'$ *and* $\mathsf{k}^p : \rho \in \Sigma$ *and* $\rho' \sqsubseteq \rho$*;*
   (b) $\mathsf{k}^p \in \mathsf{cored}(\Sigma')$ *and* $\mathsf{k}^p \notin \mathsf{cored}(\Sigma)$ *and* $\mathsf{k}^p \notin \mathsf{ran}_c(h')$ *and* $\mathsf{k}^p \in \mathsf{ran}_c(h)$*;*
   (c) $\mathsf{k}^p \notin \mathsf{cored}(\Sigma')$ *and* $\mathsf{k}^p \in \mathsf{cored}(\Sigma)$ *and* $\mathsf{k}^p \in \mathsf{ran}_c(h')$ *and* $\mathsf{k}^p \notin \mathsf{ran}_c(h)$.
3. *We define* $\langle\Sigma';h'\rangle \sqsubseteq^\flat \langle\Sigma;h\rangle$ *if whenever* $\mathsf{k}^p,\ \mathsf{k}^{\bar{p}} \in h'$ *we have:*
   - *either* $\mathsf{k}^p,\ \mathsf{k}^{\bar{p}} \in h$ *and one of the conditions (2a), (2b), (2c) is satisfied;*
   - *or* $\mathsf{k}^p,\ \mathsf{k}^{\bar{p}} \notin h$ *and* $\mathsf{k}^p,\ \mathsf{k}^{\bar{p}} \notin \mathsf{dom}(\Sigma)$ *and* $\mathsf{k}^p : \rho,\ \mathsf{k}^{\bar{p}} : \overline{\rho} \in \Sigma'$ *for some* ρ.

Note that $\sqsubseteq$ and $\sqsubseteq^\flat$ as defined above are partial order relations.

We can now state the Subject Reduction theorem:

Theorem 3.8 (Subject Reduction).

1. $\Gamma;\Sigma \vdash \mathsf{e} : \mathsf{t}$ *and* $\Gamma;\Sigma \vdash h$ *and* $\mathsf{e},h \longrightarrow \mathsf{e}',h'$ *via an expression reduction imply* $\Gamma';\Sigma' \vdash \mathsf{e}' : \mathsf{t}$ *and* $\Gamma';\Sigma' \vdash h'$*, where* $\Gamma \subseteq \Gamma'$ *and* $\langle\Sigma';h'\rangle \sqsubseteq \langle\Sigma;h\rangle$.
2. $\Gamma;\Sigma \vdash \mathsf{e} : \mathsf{t}$ *and* $\Gamma;\Sigma \vdash h$ *and* $\mathsf{e},h \longrightarrow \mathsf{e}_1 \,|\, \mathsf{e}_2,h'$ *via a thread reduction imply* $\Gamma;\Sigma \vdash \mathsf{e}_1 \,|\, \mathsf{e}_2 : \mathsf{thread}$ *and* $\Gamma';\Sigma' \vdash h'$ *where* $\langle\Sigma';h'\rangle \sqsubseteq \langle\Sigma;h\rangle$.
3. $\Gamma;\Sigma \vdash P : \mathsf{thread}$ *and* $\Gamma;\Sigma \vdash h$ *and* $P,h \longrightarrow P',h'$ *imply* $\Gamma';\Sigma' \vdash P' : \mathsf{thread}$ *and* $\Gamma';\Sigma' \vdash h'$ *where* $\Gamma \subseteq \Gamma'$ *and* $\langle\Sigma';h'\rangle \sqsubseteq^\flat \langle\Sigma;h\rangle$.

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

4 Progress Properties

The Subject Reduction Theorem assures that, in well-typed processes, when a receiving expression is executed, the input value is consistent with the type of receiving channel. This does not guarantee that once a session started, all required communications will be really executed: a process could be stuck in a deadlock even if it is well-typed. The deadlock freedom is usually called *progress* in the literature, see e.g. [23]. Progress has not been considered in most previous works on synchronous and asynchronous session type systems [2, 9, 11, 12, 16, 22, 27]. Also in our system well typing does not guarantees progress, as the following example shows.

Example 4.1. Take the following processes P_0 and P_1:

$$P_0 = \mathsf{connect}\ \mathsf{c}_0\, \mathsf{s}_0 \{\mathsf{connect}\ \mathsf{c}_1\, \mathsf{s}_1 \{\mathsf{c}_1.\mathsf{receive}\,;\mathsf{c}_0.\mathsf{send}\,(3)\}\}$$
$$P_1 = \mathsf{connect}\ \mathsf{c}_0\, \bar{\mathsf{s}}_0 \{\mathsf{connect}\ \mathsf{c}_1\, \bar{\mathsf{s}}_1 \{\mathsf{c}_0.\mathsf{receive}\,;\mathsf{c}_1.\mathsf{send}\,(5)\}\}$$

where $\mathsf{s}_0 = \mathsf{begin}.?\mathsf{int}.\mathsf{end}$ and $\mathsf{s}_1 = \mathsf{begin}.!\mathsf{int}.\mathsf{end}$.

The process $P_0 \,|\, P_1$ running from an empty heap reduces to:

$$\mathsf{k}^+_1.\mathsf{receive}\,;\mathsf{k}^+_0.\mathsf{send}\,(3)\,|\,\mathsf{k}^-_0.\mathsf{receive}\,;\mathsf{k}^-_1.\mathsf{send}\,(5), \quad [\,]$$

which is stuck even if it is well-typed.

Following essentially ideas from [6] we propose an effect system which assures progress of AMOOSE processes.

We consider a process being stuck if all its non terminated threads are waiting for a communication on channels whose associated queues are empty, and which cannot be fed by any sending expression. More formally we have the following notion.

Definition 4.2. *A process P_0 has the* progress *property if $P_0, [\,] \longrightarrow\!\!\!\!\!\rightarrow P, h$ implies that one of the following holds.*

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

A process with the progress property can stop only if its component threads either have terminated their associated computation leading to values or at least one of them either throws an exception or it is waiting for a connection through the execution of a connect statement. In this last case a new process entering the system can restart the computation opening a new channel via the execution of the connect expression.

We now give a set of inference rules that assures that all processes satisfying them have the progress property. A difference with [6, 7] is that there the type system itself was assuring the progress property, while here we separated the two goals. More interestingly the asynchronicity of output allows more permissive requirements.

With the output being asynchronous, processes can only stop on receiving expressions. For this reason we require that in the body of a session opened on channel c all receiving expressions have c as a subject. In Example 4.1 we see that the expression $\mathsf{c}_0.\mathsf{receive}$ is in the body of the session opened on channel c_1. It is easy to verify that if this expression is moved past the end of the session opened on c_1, the resulting process has the progress property. Note that swapping the sending and receiving expressions in both P_0 and P_1 the resulting process would be stuck in the system of [6].

Output expressions can always be reduced, but they can in some cases produce deadlocks by sending channels whose session expressions cannot be executed by the receiving process (see Example 4.4). For this reason we require that in the body of a session opened on channel c all expressions sending channels have c as subject.

A method call must respect the same conditions, and this is assured by the new rules for well-formed methods of Fig. 13.

We will define formally in Definition 4.5 the notion of *critical expression:* for now a critical expression is an expression which can produce deadlock if its use is not disciplined. Critical expressions are mostly session expressions, but also a method call can be critical. The set containing the subject of a critical expression (this notion will be generalised to method calls too) is said to be the *hot set* of the expression. As motivated below, we will force all critical expressions occurring in the body a session to have the same hot set containing only the channel on which the session has been opened. The notion of hot set can be naturally propagated through composition and spawning.

A channel is *used* in an expression if it occurs in the expression as subject of a session expression, or as a channel communicated by a sendS expression, or as actual parameter with a running session type of a procedure call.

The judgements of our effect system have the form

$$\mathsf{e} \triangleright \mathcal{U};\ \mathcal{H}$$

where $\mathcal{U}$ (the *used channel set*) is the set of used channels in e and $\mathcal{H}$ is the hot set of e. The set of used channels is motivated essentially by rule **ReceiveS**$^{\triangleright}$ (see Fig. 14).

We define the *singleton-union* of two hot sets $\mathcal{H}_1$ and $\mathcal{H}_2$ (notation $\mathcal{H}_1 \uplus \mathcal{H}_2$) as:

$$\mathcal{H}_1 \uplus \mathcal{H}_2 = \begin{cases} \mathcal{H}_1 \cup \mathcal{H}_2 \ \text{ if } \mathcal{H}_1 = \mathcal{H}_2 \text{ or } \mathcal{H}_1 = \emptyset \text{ or } \mathcal{H}_2 = \emptyset, \\ \text{undefined otherwise.} \end{cases}$$

The used channels and the hot set of an expression are derived by the set of inference rules given in Fig. 13 and 14. The key observations are:

- a channel which is subject of a critical subexpression of an expression must be used in the whole expression (*i.e.*, if $\mathsf{e} \triangleright \mathcal{U};\ \mathcal{H}$, then $\mathcal{H} \subseteq \mathcal{U}$);
- a channel which is used in a typed expression must be the subject of an assumption in the session environment which types that expression (*i.e.*, if $\mathsf{e} \triangleright \mathcal{U};\ \mathcal{H}$ and $\Gamma;\Sigma \vdash \mathsf{e} : \mathsf{t}$, then $\mathcal{U} \subseteq \mathsf{dom}(\Sigma)$).

The rules in Fig. 13 are quite natural, except for those concerning method calls that will be discussed later. Rule **Seq**$^{\triangleright}$ takes the union of the two used channel sets and the singleton-union of the two hot sets.

As for the rules for communication expressions, note that a send expression can stay everywhere, its hot set is then the hot set of the expression which is sent (rule **Send**$^{\triangleright}$),

Well-Formed Methods

MCold$^{\triangleright}$
$$\frac{\mathsf{e} \triangleright \mathcal{U}; \emptyset \qquad \mathcal{U} \subseteq \{\tilde{\mathsf{y}}\}}{\mathsf{t\,m}\ (\tilde{\mathsf{t}}\,\tilde{\mathsf{x}}, \tilde{\rho}\,\tilde{\mathsf{y}})\ \{\mathsf{e}\}\ \text{ is } \mathtt{ok}\ \mathtt{in}\ C}$$

MHot$^{\triangleright}$
$$\frac{\mathsf{e} \triangleright \mathcal{U}; \{\mathsf{y}_1\} \qquad \mathcal{U} \subseteq \{\mathsf{y}_1\} \cup \{\tilde{\mathsf{y}}\}}{\mathsf{t\,m}\ (\tilde{\mathsf{t}}\,\tilde{\mathsf{x}}, \rho_1\,\mathsf{y}_1, \tilde{\rho}\,\tilde{\mathsf{y}})\ \{\mathsf{e}\}\ \text{ is } \mathtt{ok}\ \mathtt{in}\ C}$$

Progress Inference Rules for Values

Null$^{\triangleright}$ $\mathsf{null} \triangleright \emptyset; \emptyset$ **Oid**$^{\triangleright}$ $\mathsf{o} \triangleright \emptyset; \emptyset$ **True**$^{\triangleright}$ $\mathsf{true} \triangleright \emptyset; \emptyset$ **False**$^{\triangleright}$ $\mathsf{false} \triangleright \emptyset; \emptyset$ **Chan**$^{\triangleright}$ $\mathsf{c} \triangleright \emptyset; \emptyset$

Progress Inference Rules for Standard Expressions

Var$^{\triangleright}$ $\mathsf{x} \triangleright \emptyset; \emptyset$

This$^{\triangleright}$ $\mathsf{this} \triangleright \emptyset; \emptyset$

Fld$^{\triangleright}$
$$\frac{\mathsf{e} \triangleright \mathcal{U}; \mathcal{H}}{\mathsf{e.f} \triangleright \mathcal{U}; \mathcal{H}}$$

Seq$^{\triangleright}$
$$\frac{\mathsf{e} \triangleright \mathcal{U}; \mathcal{H} \qquad \mathsf{e}' \triangleright \mathcal{U}'; \mathcal{H}'}{\mathsf{e};\mathsf{e}' \triangleright \mathcal{U} \cup \mathcal{U}'; \mathcal{H} \uplus \mathcal{H}'}$$

FldAss$^{\triangleright}$
$$\frac{\mathsf{e} \triangleright \mathcal{U}; \mathcal{H} \qquad \mathsf{e}' \triangleright \mathcal{U}'; \mathcal{H}'}{\mathsf{e.f} := \mathsf{e}' \triangleright \mathcal{U} \cup \mathcal{U}'; \mathcal{H} \uplus \mathcal{H}'}$$

NewC$^{\triangleright}$ $\mathsf{new}\ C \triangleright \emptyset; \emptyset$

NewS$^{\triangleright}$ $\mathsf{new}\ (\mathsf{s}, \bar{\mathsf{s}}) \triangleright \emptyset; \emptyset$

Spawn$^{\triangleright}$
$$\frac{\mathsf{e} \triangleright \mathcal{U}; \mathcal{H}}{\mathsf{spawn}\{\ \mathsf{e}\ \} \triangleright \mathcal{U}; \mathcal{H}}$$

NullPE$^{\triangleright}$ $\mathsf{NullExc} \triangleright \emptyset; \emptyset$

MethCold$^{\triangleright}$
$$\frac{\mathsf{e} \triangleright \mathcal{U}; \mathcal{H} \quad \mathsf{e}_i \triangleright \mathcal{U}_i; \mathcal{H}_i \quad i \in \{1 \ldots n\} \qquad \mathsf{mtype}(\mathsf{m}, C) = \mathsf{t}_1, \ldots, \mathsf{t}_n, \rho_1, \ldots, \rho_m \overset{\ominus}{\to} \mathsf{t}}{\mathsf{e.m}(\mathsf{e}_1, \ldots, \mathsf{e}_n, \mathsf{u}_1, \ldots, \mathsf{u}_m) \triangleright \mathcal{U} \cup \mathcal{U}_1 \ldots \cup \mathcal{U}_n \cup \{\mathsf{u}_1, \ldots, \mathsf{u}_m\}; \mathcal{H} \uplus \mathcal{H}_1 \ldots \uplus \mathcal{H}_n}$$

MethHot$^{\triangleright}$
$$\frac{\mathsf{e} \triangleright \mathcal{U}; \mathcal{H} \quad \mathsf{e}_i \triangleright \mathcal{U}_i; \mathcal{H}_i \quad i \in \{1 \ldots n\} \qquad \mathsf{mtype}(\mathsf{m}, C) = \mathsf{t}_1, \ldots, \mathsf{t}_n, \rho_1, \ldots, \rho_m \overset{\oplus}{\to} \mathsf{t}}{\mathsf{e.m}(\mathsf{e}_1, \ldots, \mathsf{e}_n, \mathsf{u}_1, \ldots, \mathsf{u}_m) \triangleright \mathcal{U} \cup \mathcal{U}_1 \ldots \cup \mathcal{U}_n \cup \{\mathsf{u}_1, \ldots, \mathsf{u}_m\}; \mathcal{H} \uplus \mathcal{H}_1 \ldots \uplus \mathcal{H}_n \uplus \{\mathsf{u}_1\}}$$

Fig. 13. Well-Formed Methods and Progress Inference Rules for Values & Standard Expressions.

while the hot set of a receive expression is forced to be the set containing the subject of the expression (rule **Receive**$^{\triangleright}$).

We have two rules for the receiveS expressions. If the body of the receiveS expression has an empty hot set, then there are no restrictions on the possible channels used in it (rule **ReceiveSA**$^{\triangleright}$). Instead, if $\{\mathsf{x}\}$ is the hot set of the body of a $\mathsf{receiveS}(\mathsf{x})$, then we must require that no other channel is used in this body (rule **ReceiveS**$^{\triangleright}$), as the following example shows.

Example 4.3. Take the following processes P_2 and P_3:

$$P_2 = \mathsf{connect}\ \mathsf{c}_0\,\mathsf{s}_0\{\mathsf{connect}\ \mathsf{c}_1\,\mathsf{s}_1\{\mathsf{c}_1.\mathsf{sendS}\,(\mathsf{c}_0)\}\}$$
$$P_3 = \mathsf{connect}\ \mathsf{c}_0\ \bar{\mathsf{s}}_0\{\mathsf{connect}\ \mathsf{c}_1\bar{\mathsf{s}}_1\{\mathsf{c}_1.\mathsf{receiveS}\,(\mathsf{x})\{\mathsf{x}.\mathsf{receive};$$
$$\mathsf{connect}\ \mathsf{c}_2\ \mathsf{s}_2\{\mathsf{c}_2.\mathsf{sendS}\,(\mathsf{c}_0)\}\}\}\}$$

Progress Inference Rules for Communication Expressions

Conn$^{\rhd}$
$$\frac{\mathsf{e} \rhd \mathcal{U};\ \mathcal{H} \quad \mathcal{H} \subseteq \{\mathsf{a}\}}{\mathsf{connect\ a\ s}\ \{\mathsf{e}\} \rhd \mathcal{U} \setminus \{\mathsf{a}\};\emptyset}$$

Send$^{\rhd}$
$$\frac{\mathsf{e} \rhd \mathcal{U};\ \mathcal{H}}{\mathsf{u\,.send}\,(\ \mathsf{e}\ ) \rhd \mathcal{U} \cup \{\mathsf{u}\};\ \mathcal{H}}$$

Receive$^{\rhd}$
$$\mathsf{u\,.receive} \rhd \{\mathsf{u}\};\ \{\mathsf{u}\}$$

SendS$^{\rhd}$
$$\mathsf{u\,.sendS}\,(\mathsf{u}') \rhd \{\mathsf{u}\,,\,\mathsf{u}'\};\ \{\mathsf{u}\}$$

ReceiveS$^{\rhd}$
$$\frac{\mathsf{e} \rhd \{\mathsf{x}\};\ \{\mathsf{x}\}}{\mathsf{u\,.receiveS}\,(\mathsf{x})\{\mathsf{e}\} \rhd \{\mathsf{u}\};\ \{\mathsf{u}\}}$$

ReceiveSA$^{\rhd}$
$$\frac{\mathsf{e} \rhd \mathcal{U};\ \emptyset}{\mathsf{u\,.receiveS}\,(\mathsf{x})\{\mathsf{e}\} \rhd \mathcal{U} \setminus \{\mathsf{x}\} \cup \{\mathsf{u}\};\ \{\mathsf{u}\}}$$

SendIf$^{\rhd}$
$$\frac{\mathsf{e} \rhd \mathcal{U};\ \mathcal{H} \qquad \mathsf{e}_i \rhd \mathcal{U}_i;\ \mathcal{H}_i \quad i \in \{1,2\}}{\mathsf{u\,.sendIf}\,(\mathsf{e})\{\mathsf{e}_1\}\{\mathsf{e}_2\} \rhd \mathcal{U} \cup \mathcal{U}_1 \cup \mathcal{U}_2 \cup \{\mathsf{u}\};\ \mathcal{H} \uplus \mathcal{H}_1 \uplus \mathcal{H}_2}$$

ReceiveIf$^{\rhd}$
$$\frac{\mathsf{e}_i \rhd \mathcal{U}_i;\ \mathcal{H}_i \quad i \in \{1,2\}}{\mathsf{u\,.receiveIf}\,\{\mathsf{e}_1\}\{\mathsf{e}_2\} \rhd \mathcal{U}_1 \cup \mathcal{U}_2 \cup \{\mathsf{u}\};\ \mathcal{H}_1 \uplus \mathcal{H}_2 \uplus \{\mathsf{u}\}}$$

SendWhile$^{\rhd}$
$$\frac{\mathsf{e} \rhd \emptyset;\emptyset \qquad \mathsf{e}' \rhd \mathcal{U};\ \mathcal{H} \quad \mathcal{U} \subseteq \{\mathsf{u}\}}{\mathsf{u\,.sendWhile}\,(\mathsf{e})\{\mathsf{e}'\} \rhd \{\mathsf{u}\};\ \mathcal{H}}$$

ReceiveWhile$^{\rhd}$
$$\frac{\mathsf{e} \rhd \mathcal{U};\ \mathcal{H} \quad \mathcal{U} \subseteq \{\mathsf{u}\}}{\mathsf{u\,.receiveWhile}\,\{\mathsf{e}\} \rhd \{\mathsf{u}\};\ \{\mathsf{u}\}}$$

Fig. 14. Progress Inference Rules for Communication Expressions.

where $\mathsf{s}_0 = \mathsf{begin.?int\,.end}$ and $\mathsf{s}_1 = \mathsf{begin.!(?int\,.end).end}$ and $\mathsf{s}_2 = \mathsf{begin.!(!int\,.end).end}$.

The process $P_2 \,|\, P_3$ starting from an empty heap reduces to:

$$\mathsf{k}_0^+\mathsf{.receive}\,;\mathsf{connect}\ \mathsf{c}_2\,\mathsf{s}_2\{\mathsf{c}_2\mathsf{.sendS}\,(\mathsf{k}_0^-)\},\ [\mathsf{k}_0^+ \mapsto \varepsilon,\, \mathsf{k}_0{}^- \mapsto \varepsilon]$$

which is stuck. But also the original program does agree neither with rule **ReceiveS**$^{\rhd}$ nor with rule **ReceiveSA**$^{\rhd}$. In fact in this case $\{\mathsf{x}\}$ and $\{\mathsf{c}_0\}$ are respectively the hot set and the set of used channels of the receiveS expression body.

This example shows also that two live channels with the same name and opposite polarities can occur in the same thread.

The following example shows that some care must be taken also in handling the sendS expressions, that can as well destroy progress. We avoid this by forcing the hot sets of sendS expressions to contain their subjects.

Example 4.4. Let's consider the following processes P_4 and P_5:

$$P_4 = \mathsf{connect}\ \mathsf{c}_0\,\mathsf{s}_0\{\mathsf{connect}\ \mathsf{c}_1\,\mathsf{s}_1\{\mathsf{c}_0\mathsf{.sendS}\,(\mathsf{c}_1)\}\}$$
$$P_5 = \mathsf{connect}\ \mathsf{c}_0\,\overline{\mathsf{s}}_0\{\mathsf{connect}\ \mathsf{c}_1\,\overline{\mathsf{s}}_1\{\mathsf{c}_1\mathsf{.receive}\};\mathsf{c}_0\mathsf{.receiveS}\,(\mathsf{x})\{\mathsf{x\,.send}\,(3)\}\}$$

where $\mathsf{s}_0 = \mathsf{begin.!(!int.end).end}$ and $\mathsf{s}_1 = \mathsf{begin.!int.end}$. Then $P_4 \,|\, P_5$, starting from an empty heap, reduces to

$$\mathsf{k}_1^-.\mathsf{receive}\,;\mathsf{k}_0^-.\mathsf{receiveS}\,(\mathsf{x})\{\mathsf{x}.\mathsf{send}\,(3)\}, [\mathsf{k}_0^+ \mapsto \mathsf{k}_1{}^+,\, \mathsf{k}_0{}^- \mapsto \varepsilon,\, \mathsf{k}_1{}^+ \mapsto \varepsilon,\, \mathsf{k}_1{}^- \mapsto \varepsilon]$$

which is stuck. Note that P_4 cannot be typed since $\mathbf{SendS}^{\triangleright}$ requires c_0 as hot set.

The rules $\mathbf{SendWhile}^{\triangleright}$ and $\mathbf{ReceiveWhile}^{\triangleright}$ are justified by comparing them with the typing rules **SendWhile** and **ReceiveWhile** (see Fig. 8) and taking into account that the set of used channels must be a subset of the domain of the session environment.

According to the reduction rule $\mathbf{Meth}^{\rightarrow}$ a method call corresponds to the replacement of the method body for the call statement. So the used channels of the call can be identified with its live channel parameters. A method can have a non-empty hot set if its body contains critical expressions: in this case we convene the hot channel to be the first channel parameter. A method $\mathsf{t}\,\mathsf{m}\ (\tilde{\mathsf{t}}\,\tilde{\mathsf{x}}, \tilde{\rho}\,\tilde{\mathsf{y}})\ \{\mathsf{e}\}$ is *cold* if it is well-formed according to rule $\mathbf{MCold}^{\triangleright}$ (i.e. if the hot set of its body is empty) and *hot* if it is well-formed according rule $\mathbf{MHot}^{\triangleright}$, i.e. if the hot set of its body is $\{\mathsf{y}_1\}$. We add this information to the method type by decorating the arrow respectively by $\ominus$ and $\oplus$, i.e. we get $\tilde{\mathsf{t}}, \tilde{\rho} \overset{\ominus}{\rightarrow} \mathsf{t}$ and $\tilde{\mathsf{t}}, \tilde{\rho} \overset{\oplus}{\rightarrow} \mathsf{t}$. The *subject of a hot method call* is the actual parameter which replaces the formal parameter y_1.

A last remark concerns rule $\mathbf{Spawn}^{\triangleright}$, in which we require the hot set be preserved in the spawned expression. Referring to Example 4.1, let P_1' be the process obtained by replacing $\mathsf{c}_0.\mathsf{receive}\,;\mathsf{c}_1.\mathsf{send}\,(3)$ with $\mathsf{spawn}\,\{\mathsf{c}_0.\mathsf{receive}\,;\mathsf{c}_1.\mathsf{send}\,(3)\}$ in P_1. Then P_1' could be typed if in rule $\mathbf{Spawn}^{\triangleright}$ the hot set of the conclusion would be the empty set, but also $P_0 \,|\, P_1'$ leads to a deadlock.

We can now formally define the notion of critical expression.

Definition 4.5. *We say that an expression* e *is* critical *if it is either a* $\mathsf{receive}$*,* $\mathsf{receiveS}$*,* $\mathsf{receiveIf}$*,* sendS*,* $\mathsf{receiveWhile}$ *expression or it is a hot method call. If* e *is a critical expression, we denote by sub*(e) *its subject. A critical expression is* live *if its subject is a live channel.*

In the remaining of the present section we will show that the above rules assure the progress property. Obviously we must consider computations whose starting process is well-typed and closed.

We say that a process P is *initial* if $P = \prod_{1 \le i \le n} \mathsf{e}_i$, all e_i are user expressions and $\emptyset;\emptyset \vdash P, [\,]$ and $\mathsf{e}_i \triangleright \emptyset;\emptyset\ (1 \le i \le n)$.

The following two lemmas can be easily proved by induction on derivations.

A *direct* subexpression of e is a subexpression of e which does not occur in the body of a $\mathsf{connect}$ or $\mathsf{receiveS}$.

Lemma 4.6. *Let* $\mathsf{e} \triangleright \mathcal{U};\ \mathcal{H}$.

1. *If* $\mathcal{H} = \emptyset$*, then there are no critical direct subexpressions of* e.
2. *If* $\mathcal{H} = \{\mathsf{u}\}$*, then all critical direct subexpressions of* e *have* u *as subject.*
3. *All critical and live subexpressions of* e *are direct subexpressions of* e.

We use φ to range over t, (η), $\langle\rho,\rho'\rangle$, and $\langle\pi\rangle^*$ and we define:

$$\hat{\varphi} = \begin{cases} \mathsf{t} & \text{if } \varphi = \mathsf{t}, \\ \eta & \text{if } \varphi = (\eta), \\ \mathsf{bool} & \text{otherwise.} \end{cases}$$

By $|\rho|$ we denote the number of symbols which occur in ρ.

Lemma 4.7. *Assume $\emptyset;\emptyset \vdash P,[\,]$ and $P_0,[\,] \longrightarrow\!\!\!\!\rightarrow P,h$. Then there are Γ, Σ such that $\Gamma;\Sigma \vdash P:\mathsf{thread}$, and $\Gamma;\Sigma \vdash h$, and*

1. *ended(Σ) and*
2. *if $\mathsf{k}^p \in \mathsf{cored}(\Sigma)$, then one of the following conditions holds:*
   - $\Sigma(\mathsf{k}^p) = \overline{\Sigma(\mathsf{k}^{\bar{p}})}$*;*
   - $|\Sigma(\mathsf{k}^p)| > |\Sigma(\mathsf{k}^{\bar{p}})|$ *and* $\Sigma(\mathsf{k}^p) = ?\varphi\psi$ *and* $h = h' :: [\mathsf{k}^p \mapsto \mathsf{v} : \tilde{\mathsf{v}}]$ *and* $\Gamma;\emptyset \vdash \mathsf{v} : \hat{\varphi}$*;*
   - $|\Sigma(\mathsf{k}^{\bar{p}})| > |\Sigma(\mathsf{k}^p)|$ *and* $\Sigma(\mathsf{k}^{\bar{p}}) = ?\varphi\psi$ *and* $h = h' :: [\mathsf{k}^{\bar{p}} \mapsto \mathsf{v} : \tilde{\mathsf{v}}]$ *and* $\Gamma;\emptyset \vdash \mathsf{v} : \hat{\varphi}$*;*
   - $\mathsf{k}^{\bar{p}} \notin \mathsf{dom}(\Sigma)$ *and* $\mathsf{k}^{\bar{p}} \in \mathsf{ran}_c(h)$.

A last definition is handy for taking into account the order in which expressions are reduced.

Definition 4.8. *Let e be an expression and $\mathsf{e}_1, \mathsf{e}_2$ be two subexpressions of e. We say that e_1* precedes *e_2 in e if, for some contexts $C[-]$, $E[-]$ and $C'[-]$ we have $\mathsf{e} = C[\mathsf{e}']$ and $\mathsf{e}' = E[\mathsf{e}_1] = C'[\mathsf{e}_2]$.*

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

Recall that, according to our notational conventions, live channels are denoted by k^p. In the following we convene that the fresh live channels created reducing a thread take successive numbers according to the order of creation, i.e. they are named k_0, $\mathsf{k}_1, \ldots$. This means that if $P,h \longrightarrow\!\!\!\!\rightarrow Q,h' \longrightarrow\!\!\!\!\rightarrow R,h''$ and k_i is a channel created in the reduction $P,h \longrightarrow\!\!\!\!\rightarrow Q,h'$, and k_j is a channel created in the reduction $Q,h' \longrightarrow\!\!\!\!\rightarrow R,h''$, then $i < j$.

The following lemma relating the order of channel creation with their occurrences as hot sets is the key of our progress proof.

Lemma 4.9. *Let P_0 be initial and $P_0,[\,] \longrightarrow\!\!\!\!\rightarrow P,h$. Then*

1. *If e_1 precedes e_2 in P and e_1 is a live critical expression and $sub(\mathsf{e}_1) = \mathsf{k}_i^p$, then for all live channels k_j^q occurring in e_2 either $i > j$ or $i = j$ and $p = q$.*
2. *If a live channel k_j^p is in the queue associated to a channel k_i^q in h, then $i > j$.*

Proof. By induction on the reduction. The induction step is by cases on the last reduction rule. We give the most interesting cases.

Case Connect$^{\rightarrow}$: If the last applied rule was **Connect$^{\rightarrow}$**, then the last step of the reduction was of the form:

$$\begin{aligned} & E_1[\mathsf{connect}\ \mathsf{c}\ \mathsf{s}\{\mathsf{e}_1\}] \,|\, E_2[\mathsf{connect}\ \mathsf{c}\ \bar{\mathsf{s}}\{\mathsf{e}_2\}] \,|\, P',\ h \\ & \quad \longrightarrow E_1[\mathsf{e}_1[{}^{\mathsf{k}_i^+}\!/\mathsf{c}]] \,|\, E_2[\mathsf{e}_2[{}^{\mathsf{k}_i^-}\!/\mathsf{c}]] \,|\, P',\ h :: [\mathsf{k}_i \mapsto \varepsilon] \end{aligned}$$

for some P', where i is the highest index among those occurring in P, h. The only new channels in P are k^p_i, where i is now the highest index and occurs only in subexpressions of $\mathsf{e}_l[{}^{\mathsf{k}^p_i}/_{\mathsf{c}}]$ $(l = 1,2)$. All expressions that were preceded by e_l are now preceded by subexpressions of $\mathsf{e}_l[{}^{\mathsf{k}^p_i}/_{\mathsf{c}}]$ $(l = 1,2)$. Since the hot set of the connect expression has been inferred by rule **Conn**$^{\triangleright}$, then by Lemmas 4.6(2) and 4.6(3), all live critical expressions inside $\mathsf{e}_l[{}^{\mathsf{k}^p_i}/_{\mathsf{c}}]$ must have k^p_i as subject. From this, and induction hypothesis, Point (1) follows immediately. Point (2) is trivial by induction hypothesis.

Case ReceiveSS$^{\rightarrow}$: If the last applied rule was **ReceiveSS**$^{\rightarrow}$, then the last step of the reduction was of the form:

$$E[\mathsf{k}^p_i.\mathsf{receiveS}\,(\mathsf{x})\{\mathsf{e}\}],h :: [\mathsf{k}^p_i \mapsto \mathsf{k}^q_j : \tilde{\mathsf{v}}] \longrightarrow \mathsf{e}\,[{}^{\mathsf{k}^q_j}/_{\mathsf{x}}] \mid E[\mathsf{null}],h :: [\mathsf{k}^p_i \mapsto \tilde{\mathsf{v}}].$$

Note that Point (1) holds between subexpressions of $E[\mathsf{null}]$ by induction hypothesis. As for $\mathsf{e}\,[{}^{\mathsf{k}^p_j}/_{\mathsf{x}}]$ we distinguish two cases.

(a) If the hot set of $\mathsf{receiveS}$ expression has been inferred by rule **ReceiveSA**$^{\triangleright}$, then by Lemma 4.6(1) in $\mathsf{e}\,[{}^{\mathsf{k}^p_j}/_{\mathsf{x}}]$ there are no live and critical subexpressions and Point (1) holds trivially.

(b) If the hot set of $\mathsf{receiveS}$ expression has been inferred by rule **ReceiveS**$^{\triangleright}$, then only the channel k^p_j can be live in $\mathsf{e}\,[{}^{\mathsf{k}^p_j}/_{\mathsf{x}}]$. Thus Point (1) follows immediately.

In both cases Point (2) is trivial by induction hypothesis.

Case Meth$^{\rightarrow}$: If the last applied rule was **Meth**$^{\rightarrow}$ and the method has at least one live channel has parameter, then the last step in the reduction was of the form:

$$E[\mathsf{o}\,.\mathsf{m}\,(\tilde{\mathsf{v}}\,,\mathsf{k}^p_i\,,\tilde{\mathsf{k}}\,)]\,|\,P',h \longrightarrow E[\mathsf{e}\,[{}^{\mathsf{o}}/_{\mathtt{this}}][{}^{\tilde{\mathsf{v}}}/_{\tilde{\mathsf{x}}}][{}^{\mathsf{k}^p_i}/_{\mathsf{y}_1}][{}^{\tilde{\mathsf{k}}}/_{\tilde{\mathsf{y}}}]]\,|\,P',h$$

where $h(\mathsf{o}) = (C,\dots)$ and $\mathsf{mbody}(\mathsf{m}\,,C) = (\tilde{\mathsf{x}}\,,\mathsf{y}_1,\tilde{\mathsf{y}}\,,\,\mathsf{e}\,)$. The more interesting case is when the hot set was inferred by rule **MHot**$^{\triangleright}$. By definition $\mathsf{o}\,.\mathsf{m}\,(\tilde{\mathsf{v}}\,,\mathsf{k}^p_i\,,\tilde{\mathsf{k}}\,)$ precedes all expressions in $E[-]$ and therefore, by induction hypothesis, the index i of its subject is greater than the index of all live channels occurring in expressions in $E[-]$. By Lemma 4.6(2) and rule **MHot**$^{\triangleright}$ all critical expressions in e have y_1 as subject, that is replaced by k^p_i. Moreover note that all live channels k^q_j with $i \neq j$ replacing the formal parameters in $\mathsf{e}\,[{}^{\mathsf{o}}/_{\mathtt{this}}][{}^{\tilde{\mathsf{v}}}/_{\tilde{\mathsf{x}}}][{}^{\mathsf{k}^p_i}/_{\mathsf{y}_1}][{}^{\tilde{\mathsf{k}}}/_{\tilde{\mathsf{y}}}]$ occur in the hot method call and then, by induction hypothesis, $j \leq i$. Point (1) follows then immediately. Point (2) is trivial.

Case Spawn$^{\rightarrow}$: If the last applied rule was **Spawn**$^{\rightarrow}$, then the last step in the reduction was of the form:

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

and Points (1) and (2) follow immediately by induction hypothesis.

We conclude now with the desired progress theorem.

Theorem 4.10 (Progress). *Assume P_0 is initial and it satisfies the progress inference rules. Then P_0 has the progress property.*

Proof. If P_0 is initial we have $\emptyset;\emptyset \vdash P_0;[\,]$. Assume now that $P_0,[\,] \longrightarrow\!\!\!\rightarrow P,h$. By the subject reduction property we have $\Gamma;\Sigma \vdash P:\mathsf{thread}$ and $\Gamma;\Sigma \vdash h$ for some Γ,Σ.

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

The only interesting case is $P \equiv V \,|\, Q$, where V is a parallel of values and Q is a parallel of evaluation contexts containing irreducible session expressions. Note that an irreducible process can only have a receiving expression in the evaluation context. Let $Q \equiv \prod_{1\leq l\leq n} E_l[\mathsf{e}_l]$. Let k_i be the live channel name with the higher index which occurs in P. By Lemma 4.9(1) a receiving expression e_r having k_i^p as subject must then be in evaluation position of some thread $E_r[\mathsf{e}_r]$ of P and so there must be a statement $\mathsf{k}_i^p :?\varphi\psi \in \Sigma$ by Lemmas 3.5 and A.3. By Lemma 4.7(2) then:

(a) either $h = h' :: [\mathsf{k}_i^p \mapsto \mathsf{v} : \tilde{\mathsf{v}}]$ and $\Gamma;\emptyset \vdash \mathsf{v} : \hat{\varphi}$, so the process cannot be stuck on a receiving expression on k_i^p,

(b) or there must be a statement $\mathsf{k}_i^{\bar{p}} : !\varphi\overline{\psi} \in \Sigma$. This implies that that there must be a sending subexpression e'_s (sending a value of type $\hat{\varphi}$) of some e_s $(1 \leq s \leq n)$ with subject $\mathsf{k}_i^{\bar{p}}$ that, by Lemma 4.9, cannot be blocked by any receiving expression, except possibly a receiving expression with subject k_i^p itself preceding e'_s in e_s. This is impossible by Lemma 4.9(1).

Acknowlegments The authors are indebted to Sophia Drossopoulou who first suggested to explore asynchronous communication rules for sessions in object oriented languages.

References

1. G. Bierman, M. Parkinson, and A. Pitts. MJ: An Imperative Core Calculus for Java and Java with Effects. Technical Report 563, Univ. of Cambridge Computer Laboratory, 2003.
2. E. Bonelli, A. Compagnoni, and E. Gunter. Correspondence Assertions for Process Synchronization in Concurrent Communications. *Journal of Functional Programming*, 15(2):219–248, 2005.
3. M. Carbone, K. Honda, and N. Yoshida. A Theoretical Basis of Communication-centered Concurrent Programming. Web Services Choreography Working Group mailing list, to appear as a WS-CDL working report.
4. M. Carbone, K. Honda, and N. Yoshida. Structured Communication-Centred Programming for Web Services. In *ESOP'07*, LNCS. Springer-Verlag, 2007. To appear.
5. M. Dezani-Ciancaglini, S. Drossopoulou, E. Giachino, and N. Yoshida. Bounded Session Types for Object-Oriented Languages. http://www.di.unito.it/ dezani/papers/ddgy.pdf, 2007.
6. 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.
7. 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.
8. S. Drossopoulou. Advanced Issues in Object Oriented Languages Course Notes. http://www.doc.ic.ac.uk/˜scd/Teaching/AdvOO.html.
9. M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, , and S. Levi. Language Support for Fast and Reliable Message-based Communication in Singularity OS. In W. Zwaenepoel, editor, *EuroSys2006*, ACM SIGOPS, pages 177–190. ACM Press, 2006.

10. 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.
11. S. Gay and M. Hole. Subtyping for Session Types in the Pi-Calculus. *Acta Informatica*, 42(2/3):191–225, 2005.
12. S. Gay and V. T. Vasconcelos. A New Approach to Functional Session Types, 2006. http://www.di.fc.ul.pt/ vv/papers/gay.vasconcelos:new-functional-sessions.pdf.
13. S. Gay, V. T. Vasconcelos, and A. Ravara. Session Types for Inter-Process Communication. TR 2003–133, Department of Computing, University of Glasgow, 2003.
14. K. Honda. Types for Dyadic Interaction. In E. Best, editor, *CONCUR'93*, volume 715 of *LNCS*, pages 509–523. Springer-Verlag, 1993.
15. K. Honda. Composing Processes. In G. L. Steele, editor, *POPL'96*, pages 344–357. ACM Press, 1996.
16. 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.
17. K. Honda, N. Yoshida, and M. Carbone. Web Services, Mobile Processes and Types. *EATCS Bulletin*, 2007. To appear.
18. 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.
19. S. Microsystems Inc. The Java Tutorial: All About Sockets. http://java.sun.com/docs/books/tutorial/networking/sockets/.
20. S. Microsystems Inc. New IO APIs. http://java.sun.com/j2se/1.4.2/docs/guide/nio/index.html.
21. R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, Parts I and II. *Information and Computation*, 100(1), 1992.
22. M. Neubauer and P. Thiemann. Session Types for Asynchronous Communication. Universität Freiburg, 2004.
23. B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
24. S. Sparkes. Conversation with Steve Ross-Talbot. *ACM Queue*, 4(2):14–23, 2006.
25. 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.
26. 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.
27. 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.
28. Web Services Choreography Working Group. Web Services Choreography Description Language. http://www.w3.org/2002/ws/chor/.
29. N. Yoshida and V. T. Vasconcelos. Language Primitives and Type Disciplines for Structured Communication-based Programming Revisited. In *SecRet'06*, ENTCS. Elsevier, 2007. To appear.

A Proof of Subject Reduction

Lemma A.1. *1.* $\Sigma_1 \preceq \Sigma_1'$, *and* $\Sigma_1' \circ \Sigma_2$ *defined, imply* $\Sigma_1 \circ \Sigma_2$ *defined, and* $\Sigma_1 \circ \Sigma_2 \preceq \Sigma_1' \circ \Sigma_2$.

2. $\Sigma \cup \Sigma' \vdash \mathsf{ok}$ *and* $\emptyset \preceq \Sigma'$ *imply* $\Sigma \preceq \Sigma \cup \Sigma'$.

Proof. Easy from Definition 3.2.

Lemma A.2 **(Generation for Standard Expressions).**

1. $\Gamma;\Sigma \vdash \mathsf{x} : \mathsf{t}$ *implies* $\emptyset \preceq \Sigma$ *and* $\mathsf{x} : \mathsf{t}' \in \Gamma$ *for some* $\mathsf{t}' <: \mathsf{t}$.
2. $\Gamma;\Sigma \vdash \mathsf{c} : \mathsf{t}$ *implies* $\emptyset \preceq \Sigma$ *and* t *is a shared session type.*
3. $\Gamma;\Sigma \vdash \mathsf{null} : \mathsf{t}$ *implies* $\emptyset \preceq \Sigma$.
4. $\Gamma;\Sigma \vdash \mathsf{v} : \mathsf{t}$ *with* $\mathsf{v} \in \{\mathsf{true}, \mathsf{false}\}$ *implies* $\emptyset \preceq \Sigma$ *and* $\mathsf{t} = \mathsf{bool}$.
5. $\Gamma;\Sigma \vdash \mathsf{o} : \mathsf{t}$ *implies* $\emptyset \preceq \Sigma$ *and* $\mathsf{o} : C \in \Gamma$ *for some* $C <: \mathsf{t}$.
6. $\Gamma;\Sigma \vdash \mathsf{NullExc} : \mathsf{t}$ *implies* $\emptyset \preceq \Sigma$.
7. $\Gamma;\Sigma \vdash \mathsf{this} : \mathsf{t}$ *implies* $\emptyset \preceq \Sigma$ *and* $\mathsf{this} : C \in \Gamma$ *for some* $C <: \mathsf{t}$.
8. $\Gamma;\Sigma \vdash \mathsf{e}_1;\mathsf{e}_2 : \mathsf{t}$ *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 $(i \in \{1,2\})$.
9. $\Gamma;\Sigma \vdash \mathsf{e.f} := \mathsf{e}' : \mathsf{t}$ *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.
10. $\Gamma;\Sigma \vdash \mathsf{e.f} : \mathsf{t}$ *implies* $\Gamma;\Sigma \vdash \mathsf{e} : C$ *and* $\mathsf{f}\,\mathsf{t} \in \mathsf{fields}(C)$ *for some* C.
11. $\Gamma;\Sigma \vdash \mathsf{e.m}(\mathsf{e}_1,\ldots,\mathsf{e}_n,\mathsf{u}_1,\ldots,\mathsf{u}_m) : \mathsf{t}$ $(n,m \geq 0)$*, implies* $\Gamma;\Sigma_0 \vdash \mathsf{e} : C$*, and* $\Gamma;\Sigma_i \vdash \mathsf{e}_i : \mathsf{t}_i$ *for* $1 \leq i \leq n$*, and* $\Sigma_0 \circ \Sigma_1 \ldots \circ \Sigma_n \circ \{\mathsf{u}_1 : \rho_1, \ldots, \mathsf{u}_m : \rho_m\} \preceq \Sigma$ *and* $\mathsf{mtype}(\mathsf{m}, C) = \mathsf{t}_1, \ldots, \mathsf{t}_n, \rho_1, \ldots, \rho_m \to \mathsf{t}$*, for some* $\Sigma_0, \Sigma_i, \mathsf{t}_i, \mathsf{u}_j, \rho_j, C$ *(*$1 \leq i \leq n$*,* $1 \leq j \leq m$*).*
12. $\Gamma;\Sigma \vdash \mathsf{new}\ C : \mathsf{t}$ *implies* $\emptyset \preceq \Sigma$ *and* $C <: \mathsf{t}$.
13. $\Gamma;\Sigma \vdash \mathsf{new}\ (\mathsf{s}, \bar{\mathsf{s}}) : \mathsf{t}$ *implies* $\emptyset \preceq \Sigma$ *and* $(\mathsf{s}, \bar{\mathsf{s}}) <: \mathsf{t}$.
14. $\Gamma;\Sigma \vdash \mathsf{spawn}\{\ \mathsf{e}\ \} : \mathsf{t}$ *implies ended*(Σ')*,* $\Sigma' \preceq \Sigma$*,* $\mathsf{t} = \mathit{Object}$ *and* $\Gamma;\Sigma' \vdash \mathsf{e} : \mathsf{t}$ *for some* Σ', t'.

Proof. By induction on typing derivations. The inductive step is by case analysis over the shape of the expression being typed, and then over the last rule applied. For all points the proof is non trivial only in the cases in which the last applied rule is a non-structural one. We just show one paradigmatic case of the inductive step.

(14) If the expression being typed has the shape $\mathsf{spawn}\{\ \mathsf{e}\ \}$, let's consider the case in which the last applied rule is **WeakES**, the other cases are similar. Then

$$\frac{\Gamma;\Sigma \vdash \mathsf{spawn}\{\ \mathsf{e}\ \} : \mathsf{t}}{\Gamma;\Sigma, \mathsf{u} : \varepsilon \vdash \mathsf{spawn}\{\ \mathsf{e}\ \} : \mathsf{t}}$$

By induction hypothesis there exist Σ', t', such that $\Sigma' \preceq \Sigma$, and *ended*(Σ') and $\mathsf{t} = \mathit{Object}$ and $\Gamma;\Sigma' \vdash \mathsf{e} : \mathsf{t}'$. Since $\Sigma \preceq \Sigma, \mathsf{u} : \varepsilon$ the property follows immediately by transitivity of $\preceq$.

Lemma A.3 **(Generation for Communication Expressions).**

1. $\Gamma;\Sigma \vdash \mathsf{connect\ a\ s}\ \{\mathsf{e}\} : \mathsf{t}$ *implies* $\mathsf{s} = \mathsf{begin}.\eta$*, and* $\Gamma;\emptyset \vdash \mathsf{a} : \mathsf{begin}.\eta$ *and* $\Gamma \setminus \mathsf{a};\Sigma, \mathsf{a} : \eta \vdash \mathsf{e} : \mathsf{t}$*, for some* η.
2. $\Gamma;\Sigma \vdash \mathsf{u.receive} : \mathsf{t}$ *implies* $\{\mathsf{u} : ?\mathsf{t}\} \preceq \Sigma$.
3. $\Gamma;\Sigma \vdash \mathsf{u.send}(\mathsf{e}) : \mathsf{t}$ *implies* $\mathsf{t} = \mathit{Object}$ *and* $\Gamma;\Sigma' \vdash \mathsf{e} : \mathsf{t}'$ *and* $\Sigma' \circ \{\mathsf{u} : !\mathsf{t}'\} \preceq \Sigma$ *for some* Σ', t'.
4. $\Gamma;\Sigma \vdash \mathsf{u.receiveS}(\mathsf{x})\{\mathsf{e}\} : \mathsf{t}$ *implies* $\mathsf{t} = \mathit{Object}$ *and* $\Gamma \setminus \mathsf{x}\,;\Sigma', \mathsf{x} : \eta \vdash \mathsf{e} : \mathsf{t}'$ *and ended*$(\Sigma', \mathsf{x} : \eta)$ *and* $\{\mathsf{u} : ?(\eta)\} \circ \Sigma' \preceq \Sigma$ *for some* $\Sigma', \mathsf{t}', \eta \neq \varepsilon.\mathsf{end}$.
5. $\Gamma;\Sigma \vdash \mathsf{u.sendS}(\mathsf{u}') : \mathsf{t}$ *implies* $\mathsf{t} = \mathit{Object}$ *and* $\{\mathsf{u}' : \eta, \mathsf{u} : !(\eta)\} \preceq \Sigma$ *for some* $\eta \neq \varepsilon.\mathsf{end}$.

6. $\Gamma;\Sigma \vdash \mathsf{u.receiveIf}\{\mathsf{e}_1\}\{\mathsf{e}_2\} : \mathsf{t}$ *implies* $\Gamma;\Sigma',\mathsf{u} : \rho_i \vdash \mathsf{e}_i : \mathsf{t}$ *($i \in \{1,2\}$) and* $\Sigma',\mathsf{u} : ?\langle\rho_1,\rho_2\rangle \preceq \Sigma$ *for some* Σ',ρ_1,ρ_2.
7. $\Gamma;\Sigma \vdash \mathsf{u.sendIf}(\mathsf{e})\{\mathsf{e}_1\}\{\mathsf{e}_2\} : \mathsf{t}$ *implies* $\Gamma;\Sigma_1 \vdash \mathsf{e} : \mathsf{bool}$ *and* $\Gamma;\Sigma_2,\mathsf{u} : \rho_i \vdash \mathsf{e}_i : \mathsf{t}$ *($i \in \{1,2\}$) and* $\Sigma_1 \circ \Sigma_2,\mathsf{u} :!\langle\rho_1,\rho_2\rangle \preceq \Sigma$ *for some* $\Sigma_1,\Sigma_2,\rho_1,\rho_2$.
8. $\Gamma;\Sigma \vdash \mathsf{u.receiveWhile}\{\mathsf{e}\} : \mathsf{t}$ *implies* $\Gamma;\{\mathsf{u} : \pi\} \vdash \mathsf{e} : \mathsf{t}$ *and* $\{\mathsf{u} :?\langle\pi\rangle^*\} \preceq \Sigma$ *for some* π.
9. $\Gamma;\Sigma \vdash \mathsf{u.sendWhile}(\mathsf{e})\{\mathsf{e}'\} : \mathsf{t}$ *implies* $\Gamma;\emptyset \vdash \mathsf{e} : \mathsf{bool}$ *and* $\Gamma;\{\mathsf{u} : \pi\} \vdash \mathsf{e}' : \mathsf{t}$ *and* $\{\mathsf{u} :!\langle\pi\rangle^*\} \preceq \Sigma$ *for some* π.

Proof. Similar to that of Lemma A.2.

Lemma A.4 (Generation for Threads).

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

Proof. All three cases are trivial.

Lemma 3.4 (Preservation of Typing under Structural Equivalence) *If* $\Gamma;\Sigma \vdash P : \mathsf{thread}$ *and* $P \equiv P'$, *then* $\Gamma;\Sigma \vdash P' : \mathsf{thread}$.

Proof. By induction on the proof of $P \equiv P'$. If the proof is obtained by the commutativity or associativity the property follows by easily by Lemmas A.4(2). The case of composition with a fixed process is trivial. In the case of composition with null, if $\Gamma;\Sigma \vdash P : \mathsf{thread}$ we have immediately $\Gamma;\Sigma \vdash P \,|\, \mathsf{null} : \mathsf{thread}$.
As for the opposite direction assume $\Gamma;\Sigma \vdash P \,|\, \mathsf{null} : \mathsf{thread}$. Then there are Σ_1,Σ_2 such that $\Sigma = \Sigma_1 \cup \Sigma_2$ and $\Gamma;\Sigma_1 \vdash P : \mathsf{thread}$ and $\Gamma;\Sigma_2 \vdash \mathsf{null} : \mathsf{thread}$. By Lemma A.2(3) we have that $\emptyset \preceq \Sigma_2$ and then, by Lemma A.1(2), $\Sigma_1 \preceq \Sigma_1 \cup \Sigma_2$. By applying Lemma 3.3(2) to $\Gamma;\Sigma_1 \vdash P : \mathsf{thread}$ we conclude $\Gamma;\Sigma_1 \cup \Sigma_2 \vdash P : \mathsf{thread}$.

Lemma A.5 (Preservation of Typing under Substitution).

1. *If* $\Gamma,\mathsf{x} : \mathsf{t}\,;\Sigma \vdash \mathsf{e} : \mathsf{t}'$ *and* $\Gamma;\emptyset \vdash \mathsf{v} : \mathsf{t}$, *then* $\Gamma;\Sigma \vdash \mathsf{e}[\mathsf{v}/\mathsf{x}] : \mathsf{t}'$.
2. *If* $\Gamma\backslash\mathsf{u}\,;\Sigma \vdash \mathsf{e} : \mathsf{t}$ *and* $\mathsf{k}^p \notin \mathsf{dom}(\Sigma)$, *then* $\Gamma;\Sigma[\mathsf{k}^p/\mathsf{u}] \vdash \mathsf{e}[\mathsf{k}^p/\mathsf{u}] : \mathsf{t}$.
3. *If* $\Gamma,\mathsf{this} : C;\Sigma \vdash \mathsf{e} : \mathsf{t}$ *and* $\Gamma;\emptyset \vdash \mathsf{o} : C$, *then* $\Gamma;\Sigma \vdash \mathsf{e}[\mathsf{o}/\mathsf{this}] : \mathsf{t}$.

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

Lemma 3.5 (Subderivations) *If* $\Gamma;\Sigma \vdash E[\mathsf{e}] : \mathsf{t}$, *then there exist* Σ_1,Σ_2 *and* t' *such that* $\Sigma = \Sigma_1 \circ \Sigma_2$, *and* $\mathsf{dom}(\Sigma_1) = \mathsf{cored}(\Sigma_1)$, *and* $\Gamma;\Sigma_1 \vdash \mathsf{e} : \mathsf{t}'$ *and* $\Gamma,\mathsf{x} : \mathsf{t}';\Sigma_2 \vdash E[\mathsf{x}] : \mathsf{t}$, *where* x *is a fresh variable in* $E[-],\Gamma$.

Proof. By induction on E, and using Generation Lemmas. For example, if $E = [-];\mathsf{e}'$, then $\Gamma;\Sigma \vdash \mathsf{e};\mathsf{e}' : \mathsf{t}$ implies $\Sigma = \Sigma_1 \circ \Sigma_2$ and $\Gamma;\Sigma_1 \vdash \mathsf{e} : \mathsf{t}'$ and $\Gamma;\Sigma_2 \vdash \mathsf{e}' : \mathsf{t}$ by Lemma A.2(8). We conclude $\Gamma,\mathsf{x} : \mathsf{t}';\Sigma_2 \vdash \mathsf{x};\mathsf{e}' : \mathsf{t}$ by rules **Var** and **Seq**.

Lemma 3.6 (Context Substitution) *If* $\Gamma;\Sigma_1 \vdash \mathsf{e} : \mathsf{t}'$, *and* $\Gamma,\mathsf{x} : \mathsf{t}';\Sigma_2 \vdash E[\mathsf{x}] : \mathsf{t}$, *and* $\Sigma_1 \circ \Sigma_2$ *is defined, then* $\Gamma;\Sigma_1 \circ \Sigma_2 \vdash E[\mathsf{e}] : \mathsf{t}$.

Lemma A.6. *1. Let* $\mathsf{k}^p \notin \mathsf{cored}(\Sigma')$ *and* $\Sigma' \circ \{\mathsf{k}^p : \rho\} \preceq \Sigma$*. Then* $\Sigma(\mathsf{k}^p) = \rho\psi$ *and* $\Sigma' \preceq \Sigma[\mathsf{k}^p \mapsto \psi^\diamond]$ *for some light* ψ*.*

2. Let $\{\mathsf{k}^p : \rho\} \circ \Sigma' \preceq \Sigma$*. Then* $\Sigma(\mathsf{k}^p) = \rho\psi$ *and* $\Sigma' \preceq \Sigma[\mathsf{k}^p \mapsto \psi^\diamond]$ *for some* ψ*.*

Proof. All cases are easy. In case (1) note that $\Sigma'(\mathsf{k}^p)$ can be either undefined or ε.

Lemma A.7. *Let* $\mathsf{e}, h \longrightarrow \mathsf{e}', h'$ *via an elementary expression reduction. Then* $\Gamma; \Sigma \vdash \mathsf{e} : \mathsf{t}$ *and* $\Gamma; \Sigma \vdash h$ *imply* $\Gamma'; \Sigma' \vdash \mathsf{e}' : \mathsf{t}$ *and* $\Gamma'; \Sigma' \vdash h'$*, where* $\Gamma \subseteq \Gamma'$ *and* $\langle \Sigma'; h' \rangle \sqsubseteq \langle \Sigma; h \rangle$*.*

Proof. The proof is by cases on the kind of expression reduction. We consider two paradigmatic cases.
Rule **SendS**$^{\rightarrow}$.
Let $\mathsf{e} = \mathsf{k}^p.\mathsf{send}\,(\mathsf{v})$ and $h = h'' :: [\mathsf{k}^{\bar{p}} \mapsto \tilde{\mathsf{v}}']$. We have:

$$\mathsf{k}^p.\mathsf{send}\,(\mathsf{v}), h'' :: [\mathsf{k}^{\bar{p}} \mapsto \tilde{\mathsf{v}}'] \longrightarrow \mathsf{null}, h'' :: [\mathsf{k}^{\bar{p}} \mapsto \tilde{\mathsf{v}}' : \mathsf{v}]$$

By Lemmas A.3(3) and A.2(1)-(4) we have $\mathsf{t} = \mathit{Object}$ and for some Σ'', t':
1) $\Gamma; \Sigma'' \vdash \mathsf{v} : \mathsf{t}'$,
2) $\emptyset \preceq \Sigma''$,
3) $\Sigma'' \circ \{\mathsf{k}^p : !\mathsf{t}'\} \preceq \Sigma$.
By 2), 3), and Lemma A.6(1) we get:
4) $\Sigma(\mathsf{k}^p) = !\mathsf{t}'\psi$ with ψ light,
5) $\emptyset \preceq \Sigma[\mathsf{k}^p \mapsto \psi^\diamond]$.
Let $\Gamma' = \Gamma$, $\Sigma' = \Sigma[\mathsf{k}^p \mapsto \psi^\diamond]$ *and* $h' = h'' :: [\mathsf{k}^{\bar{p}} \mapsto \tilde{\mathsf{v}}' : \mathsf{v}]$.
By 5), rule **Null** and Lemma 3.3(1):
6) $\Gamma; \Sigma' \vdash \mathsf{null} : \mathsf{t}$.
By Definition 3.1 $A(\Sigma'; h')$ trivially holds, since $\mathsf{cored}(\Sigma') = \emptyset$ and $\mathsf{dom}(\Sigma') = \mathsf{dom}(\Sigma)$. Moreover $\Sigma(\mathsf{k}^p) = !\mathsf{t}'\psi$ and $\Sigma(\mathsf{k}^p) = \psi^\diamond$ imply $\langle \Sigma'; h' \rangle \sqsubseteq \langle \Sigma; h \rangle$.

Rule **SendSS**$^{\rightarrow}$.
Let $\mathsf{e} = \mathsf{k}^p.\mathsf{send}\,(\mathsf{k}^q_0)$ and $h = h'' :: [\mathsf{k}^{\bar{p}} \mapsto \tilde{\mathsf{v}}]$. We have:

$$\mathsf{k}^p.\mathsf{sendS}\,(\mathsf{k}^q_0), h'' :: [\mathsf{k}^{\bar{p}} \mapsto \tilde{\mathsf{v}}] \longrightarrow \mathsf{null}, h'' :: [\mathsf{k}^{\bar{p}} \mapsto \tilde{\mathsf{v}} : \mathsf{k}^q_0]$$

By Lemmas A.3(5) we have $\mathsf{t} = \mathit{Object}$ and for some $\eta \neq \varepsilon.\mathsf{end}$:
1) $\{\mathsf{k}^p : !(\eta), \mathsf{k}^q_0 : \eta\} \preceq \Sigma$.
By 1), and Lemma A.6(1) (note that η is ended) we get:
2) $\Sigma(\mathsf{k}^p) = !(\eta)\psi$ with ψ light,
3) $\Sigma(\mathsf{k}^q_0) = \eta$,
4) $\emptyset \preceq \Sigma \setminus \mathsf{k}^q_0[\mathsf{k}^p \mapsto \psi^\diamond]$.
Let $\Gamma' = \Gamma$, $\Sigma' = \Sigma \setminus \mathsf{k}^q_0[\mathsf{k}^p \mapsto \psi^\diamond]$ *and* $h' = h'' :: [\mathsf{k}^{\bar{p}} \mapsto \tilde{\mathsf{v}} : \mathsf{k}^q_0]$.
By 4), rule **Null** and Lemma 3.3(1):
5) $\Gamma; \Sigma' \vdash \mathsf{null} : \mathsf{t}$.
By Definition 3.1 $A(\Sigma'; h')$ trivially holds, since $\mathsf{cored}(\Sigma') = \emptyset$ and $\mathsf{k}^q_0 \notin \mathsf{dom}(\Sigma')$. Lastly we get $\langle \Sigma'; h' \rangle \sqsubseteq \langle \Sigma; h \rangle$ from $\Sigma(\mathsf{k}^p) = !\mathsf{t}'\psi$ and $\Sigma'(\mathsf{k}^p) = \psi^\diamond$ and $\mathsf{k}^q_0 \in \mathsf{cored}(\Sigma)$ (which implies $\mathsf{k}^q_0 \notin \mathsf{ran}_c(h)$ by $A(\Sigma; h)$) and $\mathsf{k}^q_0 \notin \mathsf{cored}(\Sigma')$ and $\mathsf{k}^q_0 \in \mathsf{ran}_c(h')$.

It is handy to extend to heaps the concatenation operator defined for running session types and session environments at page 10.

Definition A.8 (Heap Concatenation). *The* concatenation *of two heaps h and h' (notation $h \circ h'$) is the minimal heap such that:*

- $h \circ h'(\mathsf{o}) = (C, \tilde{\mathsf{f}} : \tilde{\mathsf{v}})$ *if* $h(\mathsf{o}) = (C, \tilde{\mathsf{f}} : \tilde{\mathsf{v}})$ *and* $\mathsf{o} \notin h'$*;*
- $h \circ h'(\mathsf{o}) = (C, \tilde{\mathsf{f}} : \tilde{\mathsf{v}})$ *if* $h'(\mathsf{o}) = (C, \tilde{\mathsf{f}} : \tilde{\mathsf{v}})$ *and* $\mathsf{o} \notin h$*;*
- $\mathsf{c} \in h \circ h'$ *if* $\mathsf{c} \in h$ *and* $\mathsf{c} \notin h'$*;*
- $\mathsf{c} \in h \circ h'$ *if* $\mathsf{c} \in h'$ *and* $\mathsf{c} \notin h$*;*
- $h \circ h'(\mathsf{k}^p) = \tilde{\mathsf{v}} : \tilde{\mathsf{v}}'$ *if* $h(\mathsf{k}^p) = \tilde{\mathsf{v}}$ *and* $h'(\mathsf{k}^p) = \tilde{\mathsf{v}}'$*;*
- $h \circ h'(\mathsf{k}^p) = \tilde{\mathsf{v}}$ *if* $h(\mathsf{k}^p) = \tilde{\mathsf{v}}$ *and* $\mathsf{k}^p \notin h'$*;*
- $h \circ h'(\mathsf{k}^p) = \tilde{\mathsf{v}}$ *if* $h'(\mathsf{k}^p) = \tilde{\mathsf{v}}$ *and* $\mathsf{k}^p \notin h$*.*

From Definitions 3.1, 3.7 and A.8 we can easily show:

Lemma A.9. *1.* $A(\Sigma_1 \circ \Sigma_2; h)$ *implies* $h = h_1 \circ h_2$ *and* $A(\Sigma_1; h_1)$ *and* $A(\Sigma_2; h_2)$ *for some* h_1, h_2*.*

2. $A(\Sigma_1 \circ \Sigma_2; h_1 \circ h_2)$ *and* $A(\Sigma_1; h_1)$ *and* $\langle \Sigma_1; h_1 \rangle \sqsubseteq \langle \Sigma_1'; h_1' \rangle$ *imply* $A(\Sigma_1' \circ \Sigma_2; h_1' \circ h_2)$ *and* $\langle \Sigma_1 \circ \Sigma_2; h_1 \circ h_2 \rangle \sqsubseteq \langle \Sigma_1' \circ \Sigma_2; h_1' \circ h_2 \rangle$*.*

Theorem 3.8 (Subject Reduction).

1. $\Gamma; \Sigma \vdash \mathsf{e} : \mathsf{t}$ *and* $\Gamma; \Sigma \vdash h$ *and* $\mathsf{e}, h \longrightarrow \mathsf{e}', h'$ *via an expression reduction imply* $\Gamma'; \Sigma' \vdash \mathsf{e}' : \mathsf{t}$ *and* $\Gamma'; \Sigma' \vdash h'$*, where* $\Gamma \subseteq \Gamma'$ *and* $\langle \Sigma'; h' \rangle \sqsubseteq \langle \Sigma; h \rangle$*.*

2. $\Gamma; \Sigma \vdash \mathsf{e} : \mathsf{t}$ *and* $\Gamma; \Sigma \vdash h$ *and* $\mathsf{e}, h \longrightarrow \mathsf{e}_1 \mid \mathsf{e}_2, h'$ *via a thread reduction imply* $\Gamma; \Sigma \vdash \mathsf{e}_1 \mid \mathsf{e}_2 : \mathsf{thread}$ *and* $\Gamma'; \Sigma' \vdash h'$ *where* $\langle \Sigma'; h' \rangle \sqsubseteq \langle \Sigma; h \rangle$*.*

3. $\Gamma; \Sigma \vdash P : \mathsf{thread}$ *and* $\Gamma; \Sigma \vdash h$ *and* $P, h \longrightarrow P', h'$ *imply* $\Gamma'; \Sigma' \vdash P' : \mathsf{thread}$ *and* $\Gamma'; \Sigma' \vdash h'$ *where* $\Gamma \subseteq \Gamma'$ *and* $\langle \Sigma'; h' \rangle \sqsubseteq^{\flat} \langle \Sigma; h \rangle$*.*

Proof. (1) An arbitrary expression reduction is of the shape $E[\mathsf{e}], h \longrightarrow E[\mathsf{e}'], h'$ where $\mathsf{e}, h \longrightarrow \mathsf{e}', h'$ is an elementary expression reduction. The proof follows from Lemmas A.7, 3.5, and 3.6 using Lemma A.9.
(2) We consider the case of rule **ReceiveSS**$^{\rightarrow}$, in which we have $h = h'' :: [\mathsf{k}^p \mapsto \mathsf{k}_0^q : \tilde{\mathsf{v}}]$ and:

$$E[\mathsf{k}^p.\mathsf{receiveS}\,(\mathsf{x})\{\mathsf{e}_1\}], h'' :: [\mathsf{k}^p \mapsto \mathsf{k}_0^q : \tilde{\mathsf{v}}] \longrightarrow \mathsf{e}_1[{}^{\mathsf{k}_0^q}\!/\mathsf{x}] \mid E[\mathsf{null}], h'' :: [\mathsf{k}^p \mapsto \tilde{\mathsf{v}}].$$

By Lemma 3.5 there are $\Sigma_1, \Sigma_2, \mathsf{t}'$ such that:
1) $\Sigma = \Sigma_1 \circ \Sigma_2$,
2) $\Gamma; \Sigma_1 \vdash \mathsf{k}^p.\mathsf{receiveS}\,(\mathsf{x})\{\mathsf{e}_1\} : \mathsf{t}'$,
3) $\Gamma, \mathsf{y} : \mathsf{t}'; \Sigma_2 \vdash E[\mathsf{y}] : \mathsf{t}$.
By Lemma A.3(4) and 2) we get $\mathsf{t}' = \mathit{Object}$ and
4) $\Gamma \setminus \mathsf{x}; \Sigma_1', \mathsf{x} : \eta \vdash \mathsf{e}_1 : \mathsf{t}''$,
5) $\mathit{ended}(\Sigma_1', \mathsf{x} : \eta)$,
6) $\{\mathsf{k}^p : ?(\eta)\} \circ \Sigma_1' \preceq \Sigma_1$,
for some $\Sigma_1', \mathsf{t}'', \eta \neq \varepsilon.\mathsf{end}$.
Notice that $A(\Sigma; h)$ implies:
7) $\mathsf{k}_0^q \notin \mathsf{dom}(\Sigma)$
and then by 1) and 6) we get:
8) $\mathsf{k}_0^q \notin \mathsf{dom}(\Sigma_1')$ and $\mathsf{k}_0^q \notin \mathsf{dom}(\Sigma_2)$.
1) and 6) imply by Lemma A.1(1)

9) $\{\mathsf{k}^p : ?(\eta)\} \circ \Sigma'_1 \circ \Sigma_2 \preceq \Sigma$
and then by Lemma A.6(2) for some ψ:
10) $\Sigma(\mathsf{k}^p) = ?(\eta)\psi$,
11) $\Sigma'_1 \circ \Sigma_2 \preceq \Sigma[\mathsf{k}^p \mapsto \psi^\diamond]$.
5) implies by definition of $\circ$:
12) $\Sigma'_1 \circ \Sigma_2 = \Sigma'_1 \cup \Sigma_2$
and then using 7), 8) and 11):
13) $\Sigma'_1, \mathsf{k}^q_0 : \eta \cup \Sigma_2 \preceq \Sigma[\mathsf{k}^p \mapsto \psi^\diamond], \mathsf{k}^q_0 : \eta$.
Let $\Gamma' = \Gamma$, $\Sigma' = \Sigma[\mathsf{k}^p \mapsto \psi^\diamond], \mathsf{k}^q_0 : \eta$ *and* $h' = h'' :: [\mathsf{k}^{\bar{p}} \mapsto \tilde{\mathsf{v}}]$.
Applying Lemma A.5(2) to 4) and 8) we derive:
14) $\Gamma; \Sigma'_1, \mathsf{k}^q_0 : \eta \vdash \mathsf{e}_1[{}^{\mathsf{k}^q_0}\!/\!_{\mathsf{x}}] : \mathsf{t}''$.
By rule **Null** we have $\Gamma; \emptyset \vdash \mathsf{null} : \mathsf{t}'$ and then by 3) and Lemma 3.6 we get:
15) $\Gamma; \Sigma_2 \vdash E[\mathsf{null}] : \mathsf{t}$.
By applying rules **Start** and **Par** to 14) and 15) we derive:
16) $\Gamma; \Sigma'_1, \mathsf{k}^q_0 : \eta \cup \Sigma_2 \vdash \mathsf{e}_1[{}^{\mathsf{k}^q_0}\!/\!_{\mathsf{x}}] \mid E[\mathsf{null}] : \mathsf{thread}$
which implies by 13) and Lemma 3.3(2):
17) $\Gamma; \Sigma' \vdash \mathsf{e}_1[{}^{\mathsf{k}^q_0}\!/\!_{\mathsf{x}}] \mid E[\mathsf{null}] : \mathsf{thread}$.
By Definition 3.1 $A(\Sigma'; h') = A(\Sigma; h)$. Lastly we get $\langle \Sigma'; h' \rangle \sqsubseteq \langle \Sigma; h \rangle$ from $\Sigma(\mathsf{k}^p) = ?(\eta)\psi$ and $\Sigma'(\mathsf{k}^p) = \psi^\diamond$ and $\mathsf{k}^q_0 \in \mathsf{ran}_c(h)$.

(3) The interesting case is when the reduction is obtained by an application of rule **Connect$^{\rightarrow}$**:

$$\begin{array}{l} E_1[\mathsf{connect\ a\ s}\{\mathsf{e}_1\}] \mid E_2[\mathsf{connect\ a\ \bar{s}}\{\mathsf{e}_2\}],\ h \\ \quad \longrightarrow\ E_1[\mathsf{e}_1[{}^{\mathsf{k}^+}\!/\!_{\mathsf{a}}]] \mid E_2[\mathsf{e}_2[{}^{\mathsf{k}^-}\!/\!_{\mathsf{a}}]],\ h :: [\mathsf{k}^+ \mapsto \varepsilon] :: [\mathsf{k}^- \mapsto \varepsilon] \quad \mathsf{k}^+, \mathsf{k}^- \notin h \end{array}$$

By Lemma A.4(2) and (1) we have for some Σ_i, t_i:
1) $\Sigma = \Sigma_1 \cup \Sigma_2$,
2) $\Gamma; \Sigma_i \vdash E_i[\mathsf{connect\ u\ s}\{\mathsf{e}_i\}] : \mathsf{t}_i \quad (i = 1, 2)$.
By Lemma 3.5 there are $\Sigma^1_i, \Sigma^2_i, \mathsf{t}'_i$ and fresh x_i $(i = 1, 2)$ such that:
3) $\Sigma_i = \Sigma^1_i \circ \Sigma^2_i$,
4) $\Gamma; \Sigma^1_i \vdash \mathsf{connect\ a\ s}\{\mathsf{e}_i\} : \mathsf{t}'_i$,
5) $\Gamma, \mathsf{x}_i : \mathsf{t}'_i; \Sigma^2_i \vdash E[\mathsf{x}_i] : \mathsf{t}_i$.
By Lemma A.3(1) we have for some η:
6) $\mathsf{s} = \mathsf{begin}.\eta$,
7) $\Gamma \setminus \mathsf{a}; \Sigma^1_i, \mathsf{a} : \eta_i \vdash \mathsf{e}_i : \mathsf{t}'_i$,
where $\eta_1 = \eta$ and $\eta_2 = \overline{\eta}$.
Let $\Gamma' = \Gamma$, $\Sigma' = \Sigma, \mathsf{k}^p : \eta, \mathsf{k}^{\bar{p}} : \overline{\eta}$ *and* $h' = h :: [\mathsf{k}^+ \mapsto \varepsilon] :: [\mathsf{k}^- \mapsto \varepsilon]$.
Let now k^i stand for k^+ if $i = 1$ and for k^- if $i = 2$. Since the k^i are fresh by 7) and Lemma A.5(2) we have:
8) $\Gamma; \Sigma^1_i, \mathsf{k}^i : \eta_i \vdash \mathsf{e}_i[{}^{\mathsf{k}^i}\!/\!_{\mathsf{a}}] : \mathsf{t}'_i$,
and from 5) and 8), by Lemma 3.6:
9) $\Gamma; \Sigma_i, \mathsf{k}^i : \eta_i \vdash E_i[\mathsf{e}_i[{}^{\mathsf{k}^i}\!/\!_{\mathsf{a}}]] : \mathsf{t}_i$.
In fact note that $(\Sigma^1_i, \mathsf{k}^i : \eta_i) \circ \Sigma^2_i$ must be defined since $\Sigma^1_i \circ \Sigma^2_i$ is defined and k^i is fresh. For the same reason $(\Sigma^1_i, \mathsf{k}^i : \eta_i) \circ \Sigma^2_i = \Sigma_i, \mathsf{k}^i : \eta_i$. From 9) by rules **Start** and **Par** we get:
10) $\Gamma; \Sigma' \vdash E_1[\mathsf{e}_1[{}^{\mathsf{k}^+}\!/\!_{\mathsf{a}}]] \mid E_2[\mathsf{e}_2[{}^{\mathsf{k}^-}\!/\!_{\mathsf{a}}]] : \mathsf{thread}$.
By Definition 3.1 $A(\Sigma; h)$ implies $A(\Sigma'; h')$ since the heaps h and h' only differ for $[\mathsf{k}^+ \mapsto \varepsilon] :: [\mathsf{k}^- \mapsto \varepsilon]$. Lastly $\langle \Sigma'; h' \rangle \sqsubseteq^\flat \langle \Sigma; h \rangle$ by the last clause of Definition 3.7(3).