Session Types for Object-Oriented Languages *

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

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

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

1 Introduction

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

A *session* is such a sequence of interactions between two parties. It starts after a connection has been established. During the session, each party may execute its own local

* This work was funded in part by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project, and FP6-2004-510996 Coordination Action TYPES, and by MIUR Cofin'04 project McTafi, and by EPSRC GR/T03208, GR/S55538, GR/T04724 and GR/S68071. This paper reflects only the authors' views and the Community is not liable for any use that may be made of the information contained therein.

computation, interleaved with several communications with the other party. Communications take the form of sending and receiving values over a channel, and additionally, throughout interaction between the two parties, there should be a perfect matching of sending actions in one with receiving actions in the other, and vice versa. This form of structured interaction is found in many application scenarios.

Session types have been proposed in [18], aiming to characterise such sessions, in terms of the types of messages received or sent by a party. For example, the session type `begin.!int.!int.?bool.end` expresses that two `int`-values will be sent, then a `bool`-value will be expected to be received, and then the protocol will be complete. Thus, session types specify the communication behaviour of a piece of software, and can be used to verify the safety, in terms of communication, of the composition of several pieces of software executing in parallel. Session types have been studied for several different settings, *i.e.*, for π-calculus-based formalisms [6, 7, 13, 18, 20, 26], for CORBA [27], for functional languages [15, 29], and recently, for CDL, a W3C standard description language for web services [3, 8, 30].

In this paper we study the incorporation of session types into object-oriented languages. To our knowledge, except for our earlier work [10], such an integration has not been attempted so far. We propose the language MOOSE, a multi-threaded object-oriented core language augmented with session types, which supports thread spawning, iterative sessions, and higher-order sessions.

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

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

expressivity We wanted to be able to express common case studies from the literature on session types and concurrent programming idioms [24], as well as those from the WC3 standard documents [8, 30]. In order to achieve expressivity, we introduced the ability to spawn new threads, to send and receive sessions (*i.e.*, higher-order sessions), conditional, and iterative sessions.

type preservation The guarantee that execution preserves types, *i.e.*, the subject reduction property, proved to be an intricate task. In fact, several session type systems in the literature fail to preserve typability after reduction of certain subtle configurations, which we identified through a detailed analysis of how types of communication channels evolve during reduction. Type preservation requires linear usage of channels; in order to guarantee this we had to prevent aliasing of channels, manifested by the fact that channel types cannot be assigned to fields.

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

The paper is organised as follows: §2 illustrates the basic ideas through an example. §3 defines the syntax of the language. §4 presents the operational semantics. §5 describes design decisions. §6 illustrates the typing system. §7 states basic theorems on type safety and communication safety. §8 discusses the related work, and §9 concludes.

A full version of this paper, [1], includes complete definitions and proofs. Also, more detailed explanations and examples can be found in [24].

2 Example: Business Protocol

We describe a typical collaboration pattern that appears in many web service business protocols [8, 30] using MOOSE. This simple protocol contains essential features by which we can demonstrate the expressivity of MOOSE: it requires a combination of session establishing, higher-order session passing, spawn, conditional and deadlock-freedom during the session.

In Fig. 1 we show the sequence diagram for the protocol which models the purchasing of items as follows: first, the `Seller` and `Buyer` participants initiate interaction over channel `c1`; then, the `Buyer` sends a product id to the `Seller`, and receives a price quote in return; finally, the `Buyer` may either accept or reject this price. If the price received is acceptable then the `Seller` connects with the `Shipper` over channel `c2`. First the `Seller` sends to the `Shipper` the details of the purchased item. Then the `Seller` delegates its part of the remaining activity with the `Buyer` to the `Shipper`, that is realised by sending `c1` over `c2`. Now the `Shipper` will await the `Buyer`'s address, before responding with the delivery date. If the price is not acceptable, then the interaction terminates.

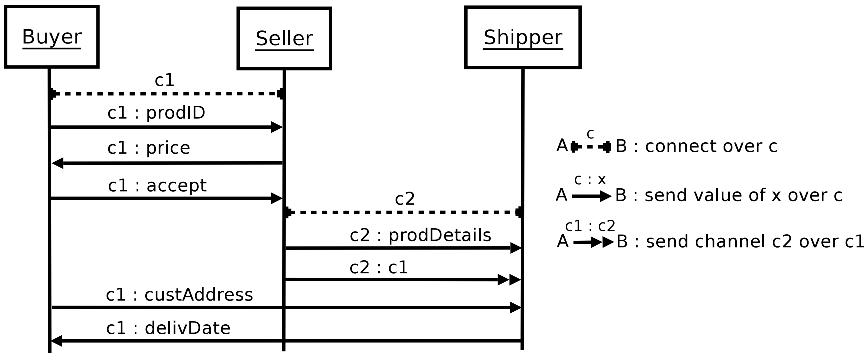

Fig. 1. Sequence diagram for item purchasing protocol.

In Fig. 2 we declare the necessary session types, and in Fig. 3 we encode the given scenario in MOOSE, using one class per protocol participant. The session types `BuyProduct` and `RequestDelivery` describe the communication patterns between `Buyer` and `Seller`, and `Seller` and `Shipper`, respectively. The session type `BuyProduct` models the sending of a `String`, then the reception of a `double`, and finally a conditional behaviour, in which a `bool` is (implicitly) sent before a branch is followed: the first branch requires that an `Address` is sent, then a `DeliveryDetails` received, and finally that the session is closed;

the second branch models an empty communication sequence and the closing of the session. We write $\overline{\texttt{BuyProduct}}$ for the *dual* type, which is constructed by taking `BuyProduct` and changing occurrences of ! to ? and vice versa; hence, these types represent the two complementary behaviours associated with a session, in which the sending of a value in one end corresponds to its reception at the other. In other words, $\overline{\texttt{BuyProduct}}$ is the same as `begin.?String.!double.?<?Address.!DeliveryDetails.end,end>`. Note that in the case of the conditional, the thread with ! in its type decides which branch is to be followed and communicates the boolean value, while the other thread passively awaits the former thread's decision. The session type `RequestDelivery` describes sending a `ProductDetails` instance, followed by sending a 'live' session channel of type `?Address.!DeliveryDetails.end`.

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

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

Sessions can start when two compatible `connect ...` statements are active. In Fig. 3, the first component of `connect` is the shared channel that is used to start communication, the second is the session type, and the third is the *session body*, which implements the session type. The method `buy` of class `Buyer` contains a `connect` statement that implements the session type `BuyProduct`, while the method `sell` of class `Seller` contains a `connect` statement over the same channel and the dual session type. When a `Buyer` and a `Seller` are executing concurrently their respective methods, they can engage in a session, which will result in a fresh channel being replaced for occurrences of the shared channel `c1` within both session bodies; freshness guarantees that the new channel only occurs in these two session bodies, therefore the objects can proceed to perform their interactions without the possibility of external interference.

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

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

```
class Buyer {

  Address addr;

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

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

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

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

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

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

communication. Notice that the code in lines 25-26 is within a `spawn`, which reduces to a new thread with the enclosed expression as its body.

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

The above example shows how MOOSE achieves deadlock-freedom during a session: because sessions take place between threads with complementary communication patterns, whenever we have `c.send(v)` eventually an expression of the shape `c.receive` will appear in the other thread, unless the thread diverges or an exception occurs or there is a connect instruction waiting for the dual connect instruction. Likewise for the other communication expressions. Therefore, no session will remain incomplete, because for every action we can guarantee that the co-action will eventually appear and the communication will take place, again, unless one of the above mentioned cases occurs.

3 A Concurrent Object Oriented Language with Sessions

In Fig. 4 we describe the syntax of MOOSE. We distinguish *user syntax*, *i.e.*, source level code, and *runtime syntax*, which includes null pointer exceptions, threads and heaps. The syntax is based on FJ [21] with the addition of imperative and communication primitives similar to those from [4, 6, 10, 18, 20, 29]. We designed MOOSE as a multi-threaded concurrent language for simplicity of presentation although it can be extended to model distribution; see § 8.

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

User syntax The metavariable t ranges over types for expressions, C ranges over class names and s ranges over session types. Each session type s has one corresponding *dual*, denoted $\overline{\mathsf{s}}$, which is obtained by replacing each ! (output) by ? (input) and vice versa. We use η to denote the type of a live channel. We introduce the full syntax of types in § 6. Class and method declarations are as expected, except for the restriction that at most one parameter can be a live channel. This condition is explained in Example 5.4.

The syntax of user expressions e, e' is standard but for the channel constructor $\mathsf{new}\ (\mathsf{s}, \overline{\mathsf{s}})$, and the *communication expressions*, *i.e.*, $\mathsf{connect}\ \mathsf{u}\ \mathsf{s}\{\mathsf{e}\}$ and all the expressions in the last three lines.

The first line gives parameter, value, the receiver this, sequence of expressions, assignment to fields, field access, method call, object and channel creation, and $\mathsf{new}\ (\mathsf{s}, \overline{\mathsf{s}})$, which builds a fresh shared channel used to establish a private session. 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 sends its result via channel u and if the result was true continues with e' and iterates, otherwise ends.

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

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

4 Operational Semantics

This section presents the operational semantics of MOOSE, which is inspired by the standard small step call-by-value reduction of [4, 5, 25] and mainly of [10]. We only discuss the more interesting rules. First we list the evaluation contexts.

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

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

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

Seq
$$\frac{\mathsf{e}_1, h \longrightarrow \mathsf{v}, h'}{\mathsf{e}_1;\mathsf{e}_2, h \longrightarrow \mathsf{e}_2, h'}$$

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

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

NewS
$$\frac{\mathsf{c} \notin h}{\mathsf{new}\ (\mathsf{s}, \overline{\mathsf{s}}), h \longrightarrow \mathsf{c}, h \cdot \mathsf{c}}$$

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

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

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

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

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

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

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

Fig. 5. Expression Reduction

Expressions Fig. 5 shows the rules for execution of expressions which correspond to the sequential part of the language. These are standard [5, 11, 21], but for the addition of a fresh shared channel to the heap (rule **NewS**). In rule **NewC** the auxiliary function

Struct $\quad P \,|\, \mathsf{null} \equiv P \qquad P \,|\, P_0 \equiv P_0 \,|\, P \qquad P \,|\, (P_0 \,|\, P_1) \equiv (P \,|\, P_0) \,|\, P_1$

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

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

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

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

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

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

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

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

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

Fig. 6. Thread Reduction

$\mathsf{fields}(C)$ examines the class table and returns the field declarations for C. The method invocation rule is **Meth**; the auxiliary function $\mathsf{mbody}(\mathsf{m}, C)$ looks up m in the class C, and returns a pair consisting of the formal parameter names and the method's code. The result is the method body where the keyword `this` is replaced by the receiver's object identifier `o`, and the formal parameters $\tilde{\mathsf{x}}$ are replaced by the actual parameters $\tilde{\mathsf{v}}$. Note that the replacement of `this` by `o` cannot lead to unwanted behaviours since the receiver cannot change during the execution of the method body.

Threads The reduction rules for threads, shown in Fig. 6, are given modulo the standard structural equivalence rules of the π-calculus [23], written ≡. 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.

Rule **Connect** describes the opening of sessions: if two threads require a session on the same channel name c with dual session types, then a new fresh channel c' is created and added to the heap. The freshness of c' guarantees privacy and bilinearity of the session communication between the two threads. Finally, the two connect expressions are replaced by their respective session bodies where the shared channel c has been substituted by the live channel c'.

Rule **ComS** gives simple session communication: value v is sent by one thread and received by another. Rule **ComSS** formalises the act of delegating a session. One thread awaits to receive a live channel, which will be bound to variable x within expression e, and another thread is ready to send such a channel. Notice that when the channel is exchanged, the receiver spawns a new thread to handle the consumption of the delegated session. This strategy is necessary in order to avoid deadlocks in the presence of circular paths of session delegation; see Example 4.1.

In rules **ComSIf-true** and **ComSIf-false**, depending on the value of the boolean, execution proceeds with either the first or the second branch. Rule **CommSWhile** simply expresses the iteration by means of the conditional. This operation allows to repeat a sequence of actions within a single session, which is convenient when describing practical communication protocols (see [8, 10]).

The following example justifies some aspects of our operational semantics.

Example 4.1. demonstrates the reason for the definition of rule **ComSS** which creates a new thread out of the expression in which the sent channel replaces the channel variable. A more natural and simpler formulation of this rule would avoid spawning a new thread:

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

However, using the above version of the rule the threads P_1 and P_2 in the table below reduce to

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

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

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

P_1

```
connect c1 begin.!int.end{
 connect c2 begin.!(!int.end).end {
  c2.sendS(c1)
 }
}
```

P_2

5 Motivating the Design of the Type System

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

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

Example 5.1. demonstrates that bilinearity is required for type preservation, and that in order to guarantee bilinearity we need to restrict aliases on live channels. Assume in the following, that in the threads P_1, P_2 and P_3 the variables $\mathtt{x}$, $\mathtt{y}$ and $\mathtt{z}$, all point to the same live channel $\mathtt{c}$ in heap h.

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

It is clear that P_3 expects to receive first an integer and then a boolean via channel $\mathtt{c}$; but P_3 could communicate first with P_1 and then with P_2 (or vice versa) receiving two integers. Therefore we need to distinguish a *shared* channel (one where a connection has not been established yet) from a *live* channel (one where a connection has been established). In order to make this distinction, shared channel types start with begin. To avoid the creation of aliases on live channels, we do not allow live channel types to be used as the types of fields, nor do we allow more than one live channel parameter in methods.

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

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

P_1

```
connect c1 s̄1 {
   c1.receive;
   c1.send(3)
}
```

P_2

```
connect c2 s̄2 {
   c2.receiveS(x){ x.send(4) }
}
```

P_3

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

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

where c'_1 and c'_2 are the fresh live channels that replaced respectively c_1 and c_2 when the sessions began. Clearly, this configuration violates the bilinearity condition.

We therefore need a notion of whether a live channel has been *consumed*, *i.e.*, whether it can still be used for the communication of values. There is no explicit user syntax for consuming channels. Instead, channels are implicitly consumed 1) at the end of a connection, 2) when they are sent over a channel, and 3) when they are used within spawn. However, types distinguish consumed channels using the end suffix. Hence, when a live channel is passed as parameter in a method call it can potentially become consumed. In § 6.1 we show that P_1 is type incorrect for any s_1 and s_2.

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

Example 5.3. demonstrates how session interleaving may cause deadlocks.

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

P_1

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

P_2

In the above example we have indefinite waiting after establishing the connection, because P_1 cannot progress unless P_2 reaches the statement $\mathsf{c}_1.\mathsf{receive}$, and P_2 cannot progress unless P_1 reaches the statement $\mathsf{c}_2.\mathsf{receive}$, and so we have a deadlock at a communication point. Note that *nesting* of sessions does not affect progress. Let us consider the following processes:

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

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

In order to avoid interleaving at live channels, we require that within each "scope" no more than one live channel can be used for communication; we call this the "hot set." The formal definition can be found in § 6. In § 6.1, we show that P_1 and P_2 are type incorrect.

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

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

P_1

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

P_2

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

We argue that the above conditions on live channels are not that restrictive. First, we can represent most of the communication protocols in the session types literature, as well as traditional synchronisation [24, § 3], while at the same time ensuring progress.

Secondly, since these conditions are only essential for progress, if we remove hot sets from typing judgements, we will obtain a more relaxed type system which allows deadlock on live channels, but still preserves type safety.

6 Type System

Types The full syntax of the types is given below.

$$\begin{array}{lll} \mathsf{t} ::= C \mid \mathsf{bool} \mid \mathsf{s} \mid (\mathsf{s},\overline{\mathsf{s}}) & \dagger ::= ! \mid ? & \\ \mathsf{s} ::= \mathsf{begin}.\rho & \rho ::= \pi.\mathsf{end} \mid \pi.\dagger\langle\rho,\rho\rangle & \eta ::= \pi \mid \rho \\ \pi ::= \varepsilon \mid \dagger\mathsf{t} \mid \dagger(\rho) \mid \dagger\langle\pi,\pi\rangle \mid \dagger\langle\pi\rangle^* \mid \pi.\pi & & \end{array}$$

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

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

The *conditional* session part has the shape $\dagger\langle\pi_1,\pi_2\rangle$: when $\dagger$ is ! this type describes sessions which send a boolean value and proceed with π_1 if the value is true, or π_2 if the value is false; when $\dagger$ is ? the behaviour is the same, except that the boolean that determines the branch is to be received instead. The *iterative* session part $\dagger\langle\pi\rangle^*$ describes sessions that respectively send or receive a boolean value, and if that value is true continue with π, *iterating*, while if the value is false, continue to following session parts, if any. Session parts can be composed into sequences using '.', hence forming longer session parts inductively; note that we use ε for the empty sequence. A *complete session part* is a session part concatenated either with end or with a conditional whose branches in turn are both complete session parts. We use ρ to range over complete session parts and η to range over both complete and incomplete session parts. Each session type s has a corresponding dual, denoted $\overline{\mathsf{s}}$, which is obtained as follows:

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

Type System We type expressions and threads with respect to a fixed class table, so only the classes declared in this table are types. We could easily extend the syntax to allow dynamic class creation, but this is orthogonal to session typing. We use the same table to judge subtyping $<:$ on class names: we assume the subtyping between classes causes no cycle as in [21]. 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 [19]: a channel on which both communication directions are allowed may also transmit data following only one of the two directions.

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

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

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

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

When typing expressions we need also to take into account which is the unique (if any) channel identifier currently used to communicate data. This is necessary in order to avoid session interleaving. Therefore we record a third set, the *hot set* $\mathcal{S}$, which can be either empty or can contain a single channel identifier belonging to the session environment. Thus the typing judgements for expressions have the shape:

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

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

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

Spawn

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

Weak

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

Meth

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

MethLin

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

Fig. 7. Some Typing Rules for Standard Expressions

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

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

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

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

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

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

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

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

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

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

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

Fig. 8. Typing Rules for Communication Expressions

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

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

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

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

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

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

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

Rule **MethLin** retrieves the type of the method $\mathtt{m}$ from the class table using the auxiliary function $\mathsf{mtype}(\mathtt{m}, C)$. This rule expects the last actual parameter u to be a channel identifier that will be used within the method body directly as if it was part of an open session. Therefore the hot sets of all the premises and of the conclusion must be $\{\mathsf{u}\}$. The session environments of the premises are also concatenated with $\{\mathsf{u} : \eta\}$ which represents the communication protocol of the live channel u during the execution of the method body.

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

Methods The following rules define well-formed methods.

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

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

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

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

Using the above operator, the typing rules for processes are straightforward. Rule **Start** promotes an expression to the thread level; and rule **Par** types a composition of threads if the composition of their session environments is defined.

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

6.1 Justifying Examples

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

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

Example 5.2: The thread P_1 cannot be typed since:

ConnI

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

ReceiveI

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

ReceiveSI

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

SendSI

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

Fig. 9. Some Inference Rules

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

Examples 5.3: Neither thread can be typed. For example, to type the expressions in line 3 in P_1 using rule **Send**, $\{\mathsf{c}_1\}$ and $\{\mathsf{c}_2\}$ should be the hot sets, respectively. Thus rule **Seq** cannot type the composition of these two expressions, since this rule requires the premises to share the same hot set.

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

6.2 Inference of Session Environments, Hot Sets, and Session Types for `connect`

Although the type system is flexible enough to express interesting protocols, typing as described so far is somewhat inconvenient, in that it requires a) the hot sets, and the session environments to be assumed (or "guessed"), and b) the session types to be stated explicitly for `connect` expressions.

To address (a), in this section, we develop *inference rules* for expressions and threads which have the shape

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

and which express that session environments and hot sets are derived rather than assumed. Based on these rules, at the end of the section, we address (b) and show how session types can be inferred for `connect` expressions.

Fig. 9 gives some of the inference rules. The rules are applicable only if all sets in the conclusion are defined. We extend the syntax of types with *type variables*, ranged over by ϕ, which stand for types, and *part of session type variables*, ranged over by ψ, which stand for part of session types. Rule **ReceiveI** introduces ϕ, since we do not know the type of the data that will be received. Rule **SendSI** introduces ψ, since we do not know the type of the channel that will be sent.

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

Because in rule **ConnI** we do not know if the session environment inferred for e contains a premise for u, we define:

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

Furthemore, the operator $\downarrow$ appends end to η if η is a session part, propagates inside the final branches of η if η is of the shape $\pi.\dagger\langle\eta_1,\eta_2\rangle$, and does nothing otherwise.

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

The following proposition states that inference computes the least session environments and hot sets.

Proposition 6.1. *1. If* $\Gamma;\Sigma;\mathcal{S} \vdash \mathsf{e} : \mathsf{t}$ *then* $\Gamma \vdash \mathsf{e} : \mathsf{t}' \parallel \Sigma';\mathcal{S}'$ *where* $\sigma(\mathsf{t}') = \mathsf{t}$ *and* $\sigma(\Sigma') \subseteq \Sigma$ *for some inference substitution* σ *and* $\mathcal{S}' \subseteq \mathcal{S}$.

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

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

As an example we show the inference for the thread P_1 of Example 4.1 in Fig. 10.

We can now address (b), *i.e.*, the inference of session types in connect expressions. This requires to modify the syntax by dropping the session types in the connect expressions. It is enough to modify the inference rule for connect avoiding to use the inference substitution for obtaining the required session types. Thus, the new inference rule is:

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

With this rule, users would not need to declare session types explicitly in connect; for example, they could write connect c {c.send (true); c.send (false)} instead of writing connect c begin.!bool.!bool.end {c.send (true); c.send (false)}.

Since explicit declarations are useful for program documentation, the inclusion of type inference for `connect` should be up to the individual language designer.

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

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

Fig. 10. An Example of Inference

7 Type Safety and Communication Safety

7.1 Subject Reduction

We will consider only reductions of well-typed expressions and threads. We define agreement between environments and heaps in the standard way and we denote it by $\Gamma \vdash h : \mathtt{ok}$. A convenient notation is $\Gamma;\Sigma;\mathcal{S} \vdash \mathsf{e};h$, which is short for $\Gamma;\Sigma;\mathcal{S} \vdash \mathsf{e} : \mathsf{t}$ for some t and $\Gamma \vdash h : \mathtt{ok}$. Similarly $\Gamma;\Sigma \vdash P;h$ means $\Gamma;\Sigma \vdash P : \mathsf{thread}$ and $\Gamma \vdash h : \mathtt{ok}$. We first show that the type system of §6 satisfies the subject reduction property.

Theorem 7.1 (Subject Reduction).

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

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

Definition 7.2 (Prefix Order on Session Environments).

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

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

Lemma 7.3 (Subderivations).

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

The proof is by induction on evaluation contexts.

7.2 Communication Safety

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

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

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

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

These properties hold for a thread obtained by reducing a well-typed closed thread in which all expressions are user expressions. We write $\prod_{0\leq i<n} \mathsf{e}_i$ for $\mathsf{e}_0 \mid \mathsf{e}_1 \mid \ldots \mid \mathsf{e}_{n-1}$. We say a thread P is *initial* if $\emptyset;\emptyset \vdash P : \mathsf{thread}$ is derivable and $P \equiv \prod_{0\leq i<n} \mathsf{e}_i$ where e_i is a user expression. Notice that this implies $\emptyset;\emptyset \vdash P;\emptyset$. For stating **P1**, we add a new constant $\mathsf{CommErr}$ (*communication error*) to the syntax and the following rule:

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

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

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

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

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

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

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

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

$$\mathsf{c.receive} \propto ?t \quad \mathsf{c.send}(\mathsf{v}) \propto !t \quad \mathsf{c.receiveS}(\mathsf{x})\{\mathsf{e}\} \propto ?(\rho) \quad \mathsf{c.sendS}(\mathsf{c}') \propto !(\rho)$$
$$\mathsf{c.receiveIf}\{\mathsf{e}_1\}\{\mathsf{e}_2\} \propto ?\langle \eta_1,\eta_2\rangle \quad \mathsf{c.sendIf}(\mathsf{v})\{\mathsf{e}_1\}\{\mathsf{e}_2\} \propto !\langle \eta_1,\eta_2\rangle$$
$$\mathsf{c.receiveWhile}\{\mathsf{e}\} \propto ?\langle \pi\rangle^* \quad \mathsf{c.sendWhile}(\mathsf{v})\{\mathsf{e}\} \propto !\langle \pi\rangle^*$$

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

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

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

The proof of Theorem 7.5 argues that if the configuration does not contain waiting connects or null pointer errors, but contains an irreducible session expression e_1, then by subject reduction and well-formedness of the session environment, the rest of the thread independently moves or it contains the dual of that irreducible expression, e_2. Then by Lemma 7.7, $\mathsf{e}_1 \propto \pi$ and $\mathsf{e}_2 \propto \overline{\pi}$. Therefore e_1 and e_2 are dual of each other and can communicate.

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

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

Finally we state the main property (**P3**) of our typing system. For this purpose, we define the partial order $\sqsubseteq$ on live channel types as the smallest partial order such that: $\eta \sqsubseteq \pi.\eta$; $\pi_i.\eta \sqsubseteq \dagger\langle\pi_1, \pi_2\rangle.\eta$ $(i \in \{1,2\})$; $\rho_i \sqsubseteq \dagger\langle\rho_1, \rho_2\rangle$ $(i \in \{1,2\})$; and $\dagger\langle\pi.\langle\pi\rangle^*, \varepsilon\rangle.\eta \sqsubseteq \langle\pi\rangle^*.\eta$.

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

Theorem 7.9 (Communication-Order Preservation). *Let P_0 be initial. Assume that $P_0, \emptyset \longrightarrow\!\!\!\!\!\rightarrow E[\mathsf{e}_0]\,|\,Q, h \longrightarrow P', h'$ where e_0 is an irreducible session expression with subject c. Then:*

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

8 Related work

Linear typing systems Session types for the π-calculus originate from linear typing systems [19, 22], whose main aim is to guarantee that a channel is used exactly or at most once within a term.

In the context of programming languages, [12] proposes a type system for checking protocols and resource usage in order to enforce linearity of variables in the presence of aliasing. They implemented the typing system in Vault [9], a low level C-like language. The main issue that they had to address is that a shared component should not refer to linear components, since aliasing of the shared component can result in non-linear usage of any linear elements to which it provides access. To relax this condition, they proposed operations for safe sharing, and for controlled linear usage. In our system non-interference is ensured by operational semantics in which substitution of shared fresh channels takes place when reducing connect, and therefore we do not need explicit constructs for this purpose. Finally, note that the system of [12] is not readily applicable in a concurrent setting, and hence in channel-based communication.

Programming languages and sessions In [29] the authors extend previous work [15], and define a concurrent functional language with session primitives. Their language supports sending of channels and higher-order values, and incorporates branching and selection, along with recursive sessions and channel sharing. Moreover, it incorporates the multi-threading primitive fork, whose operational semantics is similar to that of spawn. Finally, their system allows live channels as parameters to functions, and tracks aliasing of channels; as a result, their system is polymorphic.

In [27], the authors formalise an extension to CORBA interfaces based on session types, which are used to determine the order in which available operations can be invoked. The authors define *protocols* consisting of *sessions*, and use labelled branches and selection to model method invocation within each session. Labelled branches are also used to denote exceptions, and their system incorporates recursive session types. However, run-time checks are considered in order to check protocol conformance, and there is no formalisation in terms of operational semantics and type system.

We developed our formalism building upon previous experience with $\mathcal{L}_{doos}$ [10], a distributed object-oriented language with basic session capabilities. In the present work we have chosen to simplify the substrate to that of a concurrent calculus, and focus on the integration of advanced session types. In [10], shared channels could only be associated with a single session type each, and therefore runtime checks were not required for connections; however, this assumption is not necessary, and we preferred to compromise such superficial type-checking — the essence of our system is in typing a session body against the session type.

In our new formulation we chose not to model RMI, and in fact, an interesting question is whether we can encode RMI as a form of degenerate session in the spirit of [27]. Also, we have now introduced more powerful primitives for thread and (shared) channel creation, along with the ability to delegate live sessions via method invocation and higher-order sessions. None of these features are considered in [10]. We discovered a flaw in the progress theorem in $\mathcal{L}_{doos}$ [10], and developed the new type system with hot sets in order to guard against the offending configurations.

Subject Reduction and Progress In all previously mentioned papers on session types, typability guarantees absence of run-time communication errors. However, not all of them have the subject reduction property: the problem emerges when sending a live channel to a thread which already uses this channel to communicate, as in Example 4.1. This example can be translated into the calculi studied in [6, 14, 20, 29], and this issue has been discussed with some of their authors [2].

MOOSE has been inspired by the previously mentioned papers, however, we believe that it is the only calculus which guarantees absence of starvation on live channels. For example, we can encode the counterpart of Example 5.3 in the calculi of [6, 14, 20, 29]. More details on these two issues can be found in [1].

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

9 Conclusion and Future Work

This paper proposes the language MOOSE, a simple multi-threaded object-oriented language augmented with session communication primitives and types. MOOSE provides a clean object-oriented programming style for structural interaction protocols by prescribing channel usages as session types. We develop a typing system for MOOSE and prove type safety with respect to the operational semantics. We also show that in a well-typed MOOSE program, there will never be a communication error, starvation on live channels, nor an incorrect completion between two party interactions. These results demonstrate that a consistent integration of object-oriented language features and session types is possible where well-typedness can guarantee the consistent composition of communication protocols. To our best knowledge, MOOSE is the first application of session types to a concurrent object-oriented class-based programming language. Furthermore, type inference on session environments (Proposition 6.1), and the progress property on live channels (Theorem 7.5) have never been proved in any work on session types including those in the π-calculus.

Advanced session types An issue that arises with the use of sessions is how to group and distinguish different behaviours within a program or protocol. In [20] and subsequently in [29] the authors utilise labelled *branching* and *selection*; the first enables a process to offer alternative session paths indexed by labels, and the second is used dually to choose a path by selecting one of the available labels. In [13, 17, 20, 28], branching and selection are considered as an effective way to simulate methods of objects. Several advancements have been made, ranging from simple session subtyping [13] to more complex bounded session polymorphism [17], which corresponds to parametric polymorphism within session types. Our conditional constructs are a simplification of branching and selection, therefore the same behaviour realised by branching types can also be expressed using our types.

As another study on the enrichment of basic session types, in [6] the authors integrate the *correspondence assertions* of [16] with standard session types to reason about multi-party protocols comprising of standard interleaved sessions.

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

We plan to study transformations from methods with more than one live channel parameters to methods with only one live channel parameter; and from interleaved sessions to non-interleaved ones for investigating expressiveness of our type system.

Exceptions, timeout and implementation Another feature not considered in our system, although important in practice, is exceptions; in particular, we did not provide any way for a session type to declare that it may throw a *checked* exception, so that when this occurs both communicating processes can execute predefined error-handling code. One obvious way to encode an exception would be to use a branch as in [27]. In addition, when a thread becomes blocked waiting for a session to commence, in our operational semantics, it will never escape the waiting state unless a connection occurs. In practice, this is unrealistic, but it could have been ameliorated by introducing a 'timeout' version of our basic connection primitive such as `connect(timeout)u s {e}`. However, controlling exceptions during session communication and realising timeout would be non-trivial if we wish to preserve the progress property on live channels.

Finally, we are considering a prototype implementation using source to source translation from MOOSE to Java code. Firstly, the current notation for session types is convenient for our calculus, but sessions can be long and complex in large programs, making the types difficult to understand. We are developing an equivalent but more scalable way to describe sessions, using an alternative notation in which sessions are declared as nominal interface-like types. Other interesting issues are the choice of a suitable runtime representation for both shared and linear channels, the ability to detect and control implicit multi-threading, and the efficient implementation of higher-order sessions.

Acknowlegments Eduardo Bonelli, Adriana Compagnoni, Kohei Honda, Simon Gay, Pablo Garralda, Elsa Gunter, Antonio Ravara and Vasco Vasconcelos, discussed with us subject reduction and progress for systems with sessions types. Vasco Vasconcelos and the ECOOP reviewers gave useful suggestions. Discussions with Marco Carbone, Kohei Honda, Robin Milner and the members of W3C Web Services Choreography Working Group for their collaboration on [8, 30] motivated the example in Section 2.

References

1. A full version of this paper. http://www.doc.ic.ac.uk/~dm04.
2. Personal communication by e-mails between the authors of [6, 7, 13, 15, 18, 20, 26, 27, 29].
3. Conversation with Steve Ross-Talbot. *ACM Queue*, 4(2), 2006.
4. A. Ahern and N. Yoshida. Formalising Java RMI with Explicit Code Mobility. In *OOPSLA '05*, pages 403–422. ACM Press, 2005.
5. 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.
6. E. Bonelli, A. Compagnoni, and E. Gunter. Correspondence Assertions for Process Synchronization in Concurrent Communications. *J. of Funct. Progr.*, 15(2):219–248, 2005.
7. E. Bonelli, A. Compagnoni, and E. Gunter. Typechecking Safe Process Synchronization. In *FGUC 2004*, volume 138 of *ENTCS*, pages 3–22. Elsevier, 2005.

8. 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.
9. R. DeLine and M. Fahndrich. Enforcing High-Level Protocols in Low-Level Software. In *PLDI '01*, volume 36(5) of *SIGPLAN Notices*, pages 59–69. ACM Press, 2001.
10. M. Dezani-Ciancaglini, N. Yoshida, A. Ahern, and S. Drossopoulou. A Distributed Object Oriented Language with Session Types. In *TGC'05*, volume 3705 of *LNCS*, pages 299–318. Springer-Verlag, 2005.
11. S. Drossopoulou. Advanced issues in object oriented languages course notes. http://www.doc.ic.ac.uk/˜scd/Teaching/AdvOO.html.
12. M. Fahndrich and R. DeLine. Adoption and focus: practical linear types for imperative programming. In *PLDI '02*, pages 13–24. ACM Press, 2002.
13. S. Gay and M. Hole. Types and Subtypes for Client-Server Interactions. In *ESOP'99*, volume 1576 of *LNCS*, pages 74–90. Springer-Verlag, 1999.
14. S. Gay and M. Hole. Subtyping for Session Types in the Pi-Calculus. *Acta Informatica*, 42(2/3):191–225, 2005.
15. S. Gay, V. T. Vasconcelos, and A. Ravara. Session Types for Inter-Process Communication. TR 2003–133, Department of Computing, University of Glasgow, 2003.
16. A. D. Gordon and A. Jeffrey. Typing Correspondence Assertions for Communication Protocols. In *MFPS'01*, volume 45 of *ENTCS*, pages 379–409. Elsevier, 2001.
17. M. Hole and S. J. Gay. Bounded Polymorphism in Session Types. Technical Report TR-2003-132, Department of Computing Science, University of Glasgow, 2003.
18. K. Honda. Types for Dyadic Interaction. In *CONCUR'93*, volume 715 of *LNCS*, pages 509–523. Springer-Verlag, 1993.
19. K. Honda. Composing Processes. In *POPL'96*, pages 344–357. ACM Press, 1996.
20. K. Honda, V. T. Vasconcelos, and M. Kubo. Language Primitives and Type Disciplines for Structured Communication-based Programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer-Verlag, 1998.
21. 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.
22. N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the Pi-Calculus. *ACM TOPLAS*, 21(5):914–947, Sept. 1999.
23. R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, Parts I and II. *Information and Computation*, 100(1), 1992.
24. D. Mostrous. Moose: a Minimal Object Oriented Language with Session Types. Master's thesis, Imperial College London, 2005.
25. B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
26. K. Takeuchi, K. Honda, and M. Kubo. An Interaction-based Language and its Typing System. In *PARLE'94*, volume 817 of *LNCS*, pages 398–413. Springer-Verlag, 1994.
27. A. Vallecillo, V. T. Vasconcelos, and A. Ravara. Typing the Behavior of Objects and Components using Session Types. In *FOCLASA'02*, volume 68(3) of *ENTCS*. Elsevier, 2002.
28. V. Vasconcelos. Typed Concurrent Objects. In *ECOOP'94*, volume 821 of *LNCS*, pages 100–117. Springer-Verlag, 1994.
29. V. T. Vasconcelos, A. Ravara, and S. Gay. Session Types for Functional Multithreading. In *CONCUR'04*, volume 3170 of *LNCS*, pages 497–511. Springer-Verlag, 2004.
30. Web Services Choreography Working Group. Web Services Choreography Description Language. http://www.w3.org/2002/ws/chor/.