Session Types for Object-Oriented Languages and the price of progress

joint work with Mariangiola Dezani (Torino), Nobuko Yoshida (Imperial), and Dimitris Mostrous (Imperial)

Sophia Drossopoulou

1Department of Computing, Imperial College London, work partially supported by the EU project Mobius

FMCO 2006, 7th November 2006
Outline

1. Sessions - background
   - Sessions Example: A purchasing protocol (Typical W3C example)
2. Moose primitives
3. Operational Semantics
   - Connect
   - Interactions
4. Moose types
   - Types for Web Service Example
5. The price of soundness
6. The price of progress
7. Type System
8. Properties
A session describes a communication protocol between two parties, that takes place over a single connection.

A session describes a communication protocol between two parties, that takes place over a single connection.


Inspired by sockets, we integrated sessions in a small object calculus.

Sessions - background
Moose primitives
Operational Semantics
Moose types
The price of soundness
The price of progress
Type System
Properties

Sessions Example: A purchasing protocol (Typical W3C example)

Exchange of data/objects between three participants, i.e., three “processes” running in parallel.
Outline

1. Sessions - background
   - Sessions Example: A purchasing protocol (Typical W3C example)

2. Moose primitives

3. Operational Semantics
   - Connect
   - Interactions

4. Moose types
   - Types for Web Service Example

5. The price of soundness

6. The price of progress

7. Type System

8. Properties
We chose primitives to represent

- basic object oriented and multithreading features,
Moose primitives

We chose primitives to represent

- basic object oriented and multithreading features,
- connection establishment, send and receive values.
Moose primitives

We chose primitives to represent

- basic object oriented and multithreading features,
- connection establishment, send and receive values.
- conditional flow of protocols,
Moose primitives

We chose primitives to represent

- basic object oriented and multithreading features,
- connection establishment, send and receive values,
- conditional flow of protocols,
- repetition of part of a protocol,
Moose primitives

We chose primitives to represent

- basic object oriented and multithreading features,
- connection establishment, send and receive values,
- conditional flow of protocols,
- repetition of part of a protocol,
- delegate a session.
In Moose we have

- classes, subclasses and inheritance,
- fields, methods, dynamic method dispatch,
In Moose we have

- classes, subclasses and inheritance,
- fields, methods, dynamic method dispatch,
- spawning of threads,
Moose primitives - basic object oriented and multithreading

In Moose we have

- classes, subclasses and inheritance,
- fields, methods, dynamic method dispatch,
- spawning of threads,
- shared heap of execution
Moose primitives - connect, send, receive

- We need a primitive to establish a connection, eg when a buyer wants to contact a seller. This is done through `connect`.
Moose primitives - connect, send, receive

- We need a primitive to establish a connection, eg when a buyer wants to contact a seller. This is done through `connect`.
- We need a primitive to send a value on a channel, eg when the buyer sends his address to the seller. This is done through `send`.
Moose primitives - connect, send, receive

- We need a primitive to establish a connection, eg when a buyer wants to contact a seller. This is done through `connect`.

- We need a primitive to send a value on a channel, eg when the buyer sends his address to the seller. This is done through `send`.

- We need a primitive to receive a value on a channel, eg when the seller receives the buyer’s address.
Moose primitives - connect, send, receive

- We need a primitive to establish a connection, eg when a buyer wants to contact a seller. This is done through `connect`.
- We need a primitive to send a value on a channel, eg when the buyer sends his address to the seller. This is done through `send`.
- We need a primitive to receive a value on a channel, eg when the seller receives the buyer’s address. This is done through `receive`.
Moose primitives - connect, send, receive

- We need a primitive to establish a connection, eg when a buyer wants to contact a seller. This is done through `connect`.
- We need a primitive to send a value on a channel, eg when the buyer sends his address to the seller. This is done through `send`.
- We need a primitive to receive a value on a channel, eg when the seller receives the buyer’s address. This is done through `receive`.

... in more detail ...
class C
{
    void m()
    {
        connect c s {
            c.send(5);
            new B.h();
            bool b = c.receive;
        }
    }
}
Moose primitives - connect, send, receive

class C

void m()
{
    connect c s {
        c.send(5);
        new B.h();
        bool b = c.receive;
    }
    s = begin.
    !int.
    end
}

Sophia Drossopoulou
Moose primitives - connect, send, receive

The connect-block executes when another thread reaches a connect statement with same session identifier \( c \), and dual session type \( s' \).

```java
class C {
    void m() {
        connect c s {
            c.send(5);
            new B.h();
            bool b = c.receive;
        }
    }
}
```

Sophia Drossopoulou
Session Types for Object-Oriented Languages and the price of progress
Moose primitives - connect, send, receive

The connect-block executes when another thread reaches a connect statement with same session identifier \( c \), and *dual* session type \( s' \).

```c
class C
    void m()
    {
        connect c s {
            c.send(5);
            new B.h();
            bool b = c.receive;
        }
    }
```
The connect-block executes when another thread reaches a connect statement with same session identifier c, and dual session type s'. For example, `new C.m() || new D.g() \leadsto *...`

```java
class C {
    void m() {
        connect c s {
            c. send(5);
            new B.h();
            bool b = c. receive;
        }
    }
}

class D {
    void g() {
        connect c s' {
            ... some local computation
            c. receive;
            c. send(true);
        }
    }
}
```
Moose primitives - - conditional flow of protocols

We need primitives to continue execution depending on values sent or received. For example, if the buyer finds the asking price acceptable, they continue the negotiation by giving their details, otherwise they suggest a slightly lower price.
Moose primitives - - conditional flow of protocols

We need primitives to continue execution depending on values sent or received. For example, if the buyer finds the asking price acceptable, they continue the negotiation by giving their details, otherwise they suggest a slightly lower price.

Such conditionals affect the session types, therefore we introduced sendAndIf and receiveAndIf expressions.
Moose primitives - conditional flow of protocols

We need primitives to continue execution depending on values sent or received. For example, if the buyer finds the asking price acceptable, they continue the negotiation by giving their details, otherwise they suggest a slightly lower price.

Such conditionals affect the session types, therefore we introduced `sendAndIf` and `receiveAndIf` expressions.
Moose primitives - - conditional flow of protocols

We need primitives to continue execution depending on values sent or received. For example, if the buyer finds the asking price acceptable, they continue the negotiation by giving their details, otherwise they suggest a slightly lower price.

Such conditionals affect the session types, therefore we introduced sendAndIf and receiveAndIf expressions.
Moose primitives - conditional flow of protocols

```plaintext
\texttt{c.sendAndIf(e)\{e1\}\{e2\}} evaluates \texttt{e} and sends it value along \texttt{c}; it continues with \texttt{e1} if the former was \texttt{true}, and with \texttt{e2} otherwise.

\begin{verbatim}
c.sendAndIf(x > 55) {
    x = c.receive; c.send(true);
}
\end{verbatim}
```

Sophia Drossopoulou
Moose primitives - conditional flow of protocols

c.sendAndIf(e){e1}{e2} evaluates e and sends it value along c; it continues with e1 if the former was true, and with e2 otherwise.

c. sendAndIf(x > 55) {
    x = c.receive; c.send(true);
} {
    x := x + 1
}

Notation: …! <?int.!bool >< ε > ...
receiveAndIf is the dual of sendAndIf; Thus, c.receiveAndIf {e1}\{e2} receives a boolean along c; it continues with e1 if the former was true, and with e2 otherwise.
Moose primitives - conditional flow of protocols

`receiveAndIf` is the dual of `sendAndIf` ; Thus, `c.receiveAndIf {e1}{e2}` receives a boolean along c ; it continues with e1 if the former was true , and with e2 otherwise.

```plaintext
  c. receiveAndIf {
    c.send(3); b = c.receive;
  } {
    b := not b
  }
```
receiveAndIf is the dual of sendAndIf; Thus, c.receiveAndIf {e1}{e2} receives a boolean along c; it continues with e1 if the former was true, and with e2 otherwise.

```plaintext
c. receiveAndIf {
    c.send(3); b = c.receive;  !int.?bool
    }
    b := not b  >>
}

Notation: ...? <!int.!bool >! ε > ...
We also need primitives to express repetition of part of a protocol. For example, the seller repeats the authentication check until the buyer has given them the correct password.

As for conditionals, these iterations affect session types, and therefore we have primitives \( \texttt{c.sendWhile}(e) \{ e1 \} \{ e2 \} \) and \( \texttt{c.receiveWhile} \{ e1 \} \{ e2 \} \), with the obvious semantics.
Moose primitives - delegate a session

We need primitives to express the delegation of a running session to a further thread. For example, once the buyer has agreed the price with the seller, he passes the running session to the shipper, so that the shipper may take the buyer’s details and organize delivery of the goods.

For that reason we introduced the primitive sendSess.
Moose primitives - delegate a session

Assume execution without delegation.

1st thread

\[
\text{connect } c \ s_1 \ \{ \\
\quad c.\text{send}(3); \\
\quad c.\text{send}(5); \\
\}\n\]

where \( s_1 = \text{begin} \cdot \text{!int} \cdot \text{!int} \cdot \text{end} \), ,
Assume execution without delegation.

\begin{itemize}
\item \textit{1st thread}
\begin{verbatim}
connect c s_1 \{ 
  c.send(3);
  c.send(5);
\}
\end{verbatim}
\item \textit{2nd thread}
\begin{verbatim}
connect c s_2 \{ 
  c.receive ;
  c.receive ;
\}
\end{verbatim}
\end{itemize}

where $s_1 =$ begin.!int.!int.end , and $s_2 =$ begin.?int.?int.end,
Moose primitives - delegate a session

Now thread 1 *delegates* part of its session to a further thread.

1st thread

```plaintext
connect c s1 {
    c.send(3);
    connect c' s3 {
        c'.sendSess(c);
    }
}
```

2nd thread

```plaintext
connect c s2 {
    c.receive;
}
```

3rd thread

```plaintext
connect c' s4 {
    c'.receiveSess(x)
    x.send(5);
}
```

where $s_1 = \text{begin.}!\text{int.}!\text{int.}\text{end}$, and $s_2 = \text{begin.?}\text{int.?}\text{int.}\text{end}$, and where $s_3 = \text{begin.}!(\text{!int.end})\text{.end}$, and $s_4 = \text{begin.?}(\text{!int.end})\text{.end}$
Moose primitives - delegate a session

Now thread 1 *delegates* part of its session to a further thread.

1st thread

\[
\text{connect } c \quad s_1 \quad \{
\quad \text{c.send}(3);
\quad \text{connect } c' \quad s_3 \quad \{
\quad \text{c'.sendSess}(c);
\quad \}
\quad \}
\]

2nd thread

\[
\text{connect } c \quad s_2 \quad \{
\quad \text{c.receive};
\quad \}
\]

3rd thread

\[
\text{connect } c' \quad s_4 \quad \{
\quad \text{c'.receiveSess}(x) \quad \}
\quad \}
\]

\[
x\text{.send}(5);\]

where \(s_1 = \text{begin.}!\text{int.}!\text{int.end}\), and \(s_2 = \text{begin.}?\text{int.}?\text{int.end}\), and

where \(s_3 = \text{begin.}!(\text{!int.end}).\text{end}\), and \(s_4 = \text{begin.}?(!\text{int.end}).\text{end}\)

Notice that thread 2 is *unaware* of the change!
Outline

1. Sessions - background
   - Sessions Example: A purchasing protocol (Typical W3C example)

2. Moose primitives

3. Operational Semantics
   - Connect
   - Interactions

4. Moose types
   - Types for Web Service Example

5. The price of soundness

6. The price of progress

7. Type System

8. Properties
Operational Semantics - connect

\[ \ldots \text{connect } c \ s \{ e_1 \} \ldots \ | \ldots \text{connect } c \ \overline{s} \{ e_2 \} \ldots, \ h \]
...\texttt{connect } c \ s \{ e_1 \} ... | ... \texttt{connect } c \ s \{ e_2 \} ... , h

\[ \cdots e_1[c'/c] ... | ... e_2[c'/c] ... , h \cdot c' \]

\( \overline{s} \) is the dual of a session type.

\( c' \) is fresh i.e., not in the heap \( h \); we call these \textit{live channels}
...\texttt{connect } c \ s \{e_1\} \ldots | \ldots \texttt{connect } c \ \overline{s}\{e_2\} \ldots \), \ h

\[\downarrow\]

\ldots e_1[c'/c] \ldots | \ldots e_2[c'/c] \ldots \), \ h \cdot c'

\(\overline{s}\) is the dual of a session type.
\(c'\) is fresh \ i.e., \ not in the heap \(h\); \ we call these live channels

Freshness of \(c'\) guarantees that \(e_1\) and \(e_2\) only interact with each other
...c.send(v) ... | ...... c.receive ..., h
Operational Semantics - interactions

\[ \ldots c . \text{send}(v) \ldots | \ldots c . \text{receive} \ldots, h \]

\[ \downarrow \]

\[ \ldots \text{null} \ldots | \ldots v \ldots, h \]
Operational Semantics - interactions

\[ \cdots c.\text{send}(v) \cdots | \cdots c.\text{receive} \cdots, h \]

\[ \downarrow \]

\[ \cdots \text{null} \cdots | \cdots v \cdots, h \]

\[ \cdots c.\text{sendIf (true)} \{ e_1 \} \{ e_2 \} | c.\text{receiveIf} \{ e_3 \} \{ e_4 \} \cdots, h \]
Operational Semantics - interactions

\[ ... c . \text{send}(v) ... | \ldots \ldots c . \text{receive} \ldots, h \]

\[ \downarrow \]

\[ ... \text{null} \ldots | ... v \ldots, h \]

\[ ... c . \text{sendIf}(\text{true}) \{ e_1 \}\{ e_2 \} | c . \text{receiveIf} \{ e_3 \}\{ e_4 \} \ldots, h \]

\[ \downarrow \]

\[ ... e_1 | e_3 \ldots, h \]
Outline

1. Sessions - background
   - Sessions Example: A purchasing protocol (Typical W3C example)
2. Moose primitives
3. Operational Semantics
   - Connect
   - Interactions
4. Moose types
   - Types for Web Service Example
5. The price of soundness
6. The price of progress
7. Type System
8. Properties

Sophia Drossopoulou
Session Types for Object-Oriented Languages and the price of progress
### Moose Types

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>†</td>
<td>::= !</td>
<td>?</td>
</tr>
<tr>
<td>π</td>
<td>::= ε</td>
<td>π.π</td>
</tr>
<tr>
<td>ρ</td>
<td>::= π.end</td>
<td>π.†⟨ρ,ρ⟩</td>
</tr>
<tr>
<td>s</td>
<td>::= begin.ρ</td>
<td>shared session type</td>
</tr>
<tr>
<td>t</td>
<td>::= C</td>
<td>bool</td>
</tr>
</tbody>
</table>
session BuyProduct =
begin !String ?double !⟨!Address ?DeliveryDetails .end, end⟩

Buyer’s viewpoint of the Buyer-Seller interaction

Sophia Drossopoulou
session RequestDelivery =
begin !ProductDetails !(?Address !DeliveryDetails.end).end

Seller’s viewpoint of the Seller-Shipper interaction
Types for Web Service Example

![Diagram showing sessions and types for a web service example.]

Implementation fits in one A4 page...
Outline

1. Sessions - background
   - Sessions Example: A purchasing protocol (Typical W3C example)
2. Moose primitives
3. Operational Semantics
   - Connect
   - Interactions
4. Moose types
   - Types for Web Service Example
5. The price of soundness
6. The price of progress
7. Type System
8. Properties
The price of soundness

For soundness we need to guarantee *bilinearity* of channels, i.e., any "active" channel is accessible from exactly two threads.
For soundness we need to guarantee \textit{bilinearity} of channels, i.e., any "active" channel is accessible from exactly two threads. This requires that active channels cannot be shared.
Assume, e.g. \( c \) an active channel, with type \(!int!bool\) , or its dual, and that \( o1.f1 \), and \( o2.f2 \) and \( o3.f3 \) were pointing to \( c \).

Wrong Communication

\[
\begin{align*}
o1.f1.\text{send} (3); o1.f1.\text{send} (true) & \quad | \quad o3.f3.\text{receive} ; o3.f3.\text{receive} \\
o2.f2.\text{send} (3); o2.f2.\text{send} (true) & \\
\end{align*}
\]
The price of soundness

Assume, e.g. $c$ an active channel, with type $\texttt{!int.!bool}$, or its dual, and that $o1.f1$, and $o2.f2$ and $o3.f3$ were pointing to $c$.

Wrong Communication

```plaintext
o1.f1.send(3); o1.f1.send(true) | o2.f2.send(3); o2.f2.send(true) | o3.f3.receive; o3.f3.receive
```

Therefore, active channels, i.e., those of type $!...$, or $?...$, or $\texttt{end}$, or $\varepsilon$,

- may not be stored in fields,
- no more than one active channel may be passed as parameter.
The price of soundness

Assume, e.g. \( c \) an active channel, with type \(!\text{int}!\text{bool}\) , or its dual, and that \( o1.f1 \), and \( o2.f2 \) and \( o3.f3 \) were pointing to \( c \).

**Wrong Communication**

\[
\begin{align*}
& o1.f1.\text{send}(3); o1.f1.\text{send}(\text{true}) & | & \quad o3.f3.\text{receive}; o3.f3.\text{receive} \\
& o2.f2.\text{send}(3); o2.f2.\text{send}(\text{true}) & |
\end{align*}
\]

Therefore, active channels, i.e., those of type \(!\ldots\), or \(?\ldots\), or end, or \(\varepsilon\),

- may *not* be stored in fields,
- *no more than one* active channel may be passed as parameter.

However, shared sessions, i.e., those with type \begin{ldots\},

- may be stored in fields.
Outline

1. Sessions - background
   - Sessions Example: A purchasing protocol (Typical W3C example)
2. Moose primitives
3. Operational Semantics
   - Connect
   - Interactions
4. Moose types
   - Types for Web Service Example
5. The price of soundness
6. The price of progress
7. Type System
8. Properties
We designed Moose with the desire that it should guarantee progress, in the sense that once a session has started, it cannot deadlock at the points of interaction. Other session type systems do not guarantee that property.
Is progress desirable?

After all, multithreaded programs may deadlock. Also, Moose programs might wait indefinitely at `connect` points.
Is progress desirable?

After all, multithreaded programs may deadlock. Also, Moose programs might wait indefinitely at `connect` points. We think it *is* desirable.
Is progress desirable?

After all, multithreaded programs may deadlock. Also, Moose programs might wait indefinitely at \texttt{connect} points. We think it \textit{is} desirable. Waiting indefinitely at \texttt{connect} is like not receiving an answer on the phone when calling somebody - i.e., acceptable.
Is progress desirable?

After all, multithreaded programs may deadlock. Also, Moose programs might wait indefinitely at **connect** points. We think it *is* desirable.

Waiting indefinitely at **connect** is like not receiving an answer on the phone when calling somebody - i.e., acceptable.

Waiting indefinitely at a interaction point is like not receiving an answer *after* having started the telephone conversation - i.e., not acceptable.
Assume that the following were a possible intermediate configuration:

```
Deadlock
```

\[ c_1.\text{send}(3); c_2.\text{send}(5); \quad | \quad c_2.\text{receive}; c_1.\text{receive}; \]

Above leads to deadlock, after session had started.
Assume that the following were a possible intermediate configuration:

**Deadlock**

```
c1.send(3); c2.send(5); \mid c2.receive; c1.receive;
```

Above leads to deadlock, after session had started. In order to prevent such situations, type system forbids any kind of *interleaving*:

**Type Incorrect**

```
connect c1{
    connect c2
    c1.send(2);
    c2.send(2);
}
```
Progress affects operational semantics of `sendSess` and `receiveSess`.
Obvious, but wrong Semantics

\[
E_1[\text{receiveSess}(x)\{e\}] \mid E_2[\text{sendSess}\{c\}], \quad h \rightarrow
\]

\[
E_1[e[c'/x]] \mid E_2[\text{null}], \quad h
\]
The price of progress - 3

Obvious, but wrong Semantics

\[ E_1[\text{receiveSess}(x)\{e\}] \parallel E_2[\text{sendSess}\{c\}], \quad h \rightarrow \]

\[ E_1[e[c'/x]] \parallel E_2[\text{null}], \quad h \]

and then
The price of progress - 3

Obvious, but wrong Semantics

\[ E_1[\text{receiveSess} (x) \{ e \}] \ | \ E_2[\text{sendSess} \{ c \}], \ h \rightarrow \]
\[ E_1[e[c'/x]] \ | \ E_2[\text{null}], \ h \]

and then

connect c1 s1 {
  connect c2 s2{
    c2.receiveSess(x)\{ x.send(5) \};
    c1.receive
  }
}

would break the bilinearity condition, and rewrite to

Deadlock

\[ c'.send(5); c'.receive \ | \ \text{null}, \ h \]
Instead, receipt of a channel spawns a new thread
\[ \text{receiveSess} \,(x)\{e\} \mid \text{sendSess} \,\{c\}, \ h \rightarrow \null \mid e[c'/x] \mid \null, \ h \]
and then
\[ \text{connect} \ c1 \ s1 \{ \text{connect} \ c2 \ s2\{c2\text{receiveSess}(x)\{x\text{send}(5)\}; \text{c1.receive}\} \} \]

\[ \text{connect} \ c1 \ s1 \{ \text{connect} \ c2 \ s2 \{ \text{sendSesssend} \,(c1); \} \} \]
Instead, receipt of a channel spawns a new thread

\[
\text{receiveSess} (x) \{ e \} \mid \text{sendSess} \{ c \}, \ h \rightarrow \\
\text{null} \mid e[c'/x] \mid \text{null}, \ h
\]

and then

\[
\text{connect} \ c1 \ s1 \ \{ \\
\text{connect} \ c2 \ s2 \{ \\
\quad c2.\text{receiveSess}(x) \{ x.\text{send}(5) \} ; \\
\quad c1.\text{receive} \}
\}
\]

\[
\text{connect} \ c1 \ s1 \ \{ \\
\text{connect} \ c2 \ s2 \ { \\
\quad \text{sendSess} \text{send} (c1) ; \\
\quad } \}
\]

does \textit{not} break the bilinearity condition; it rewrites to

\[
c' . \text{send}(5) \mid c'.\text{receive} \mid \text{null}, \ h
\]
Outline

1. Sessions - background
   - Sessions Example: A purchasing protocol (Typical W3C example)
2. Moose primitives
3. Operational Semantics
   - Connect
   - Interactions
4. Moose types
   - Types for Web Service Example
5. The price of soundness
6. The price of progress
7. Type System
8. Properties
Typing

Typing has the shape

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

where
Typing has the shape

$$\Gamma; \Sigma; S \vdash e : t$$

where

- $\Gamma$ standard environment
Typing

Typing has the shape

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

where

1. $\Gamma$ standard environment
2. $\Sigma$ is the sessions environment, holding the kinds of communications of the expressions $e$
Typing has the shape

\[ \Gamma; \Sigma; S \vdash e : t \]

where

1. \( \Gamma \) standard environment
2. \( \Sigma \) is the sessions environment, holding the kinds of communications of the expressions \( e \)
3. \( S \) is the hot set, either empty or singleton, holding the currently active session
Typing has the shape

\[ \Gamma; \Sigma; S \vdash e : t \]

where

1. \( \Gamma \) standard environment
2. \( \Sigma \) is the sessions environment, holding the kinds of communications of the expressions \( e \)
3. \( S \) is the hot set, either empty or singleton, holding the currently active session
4. \( e \) is the expression
5. \( t \) is the type.
Type Rules

**Receive**

\[ \frac{}{\Gamma; \{ u : ?t \}; \{ u \} \vdash u.\text{receive} : t} \]

**Send**

\[ \frac{}{\Gamma; \Sigma; \{ u \} \vdash e : t} \]

\[ \frac{}{\Gamma; \Sigma; \{ u : !t \}; \{ u \} \vdash u.\text{send}( e ) : \text{Object}} \]
Type Rules

**Receive**

\[ \Gamma; \{ u : ?t \}; \{ u \} \vdash u.\text{receive} : t \]

**Send**

\[ \Gamma; \Sigma; \{ u \} \vdash e : t \]

\[ \Gamma; \Sigma.\{ u : !t \}; \{ u \} \vdash u.\text{send}(e) : \text{Object} \]

**Sequence**

\[ \Gamma; \Sigma S \vdash e : t \quad \Gamma; \Sigma' S \vdash e' : t \]

\[ \Gamma; \Sigma \circ \Sigma'; S \vdash e; e' : t \]
**Type Rules**

**RECEIVE**

\[ \Gamma; \{ u : ?t \}; \{ u \} \vdash u . \text{receive} : t \]

**SEND**

\[ \Gamma; \Sigma; \{ u \} \vdash e : t \]

\[ \Gamma; \Sigma; \{ u : !t \}; \{ u \} \vdash u . \text{send} ( e ) : \text{Object} \]

**SEQUENCE**

\[ \Gamma; \Sigma S \vdash e : t \quad \Gamma; \Sigma' S \vdash e' : t', \]

\[ \Gamma; \Sigma \circ \Sigma'; S \vdash e; e' : t', \]

Thus,

\[ \Gamma; x : ?\text{bool}; x \vdash x . \text{receive} : \text{bool} \quad \Gamma; x : !\text{int}; x \vdash x . \text{send} (5) : \text{int} \]

\[ \Gamma; x : ?\text{bool} . !\text{int}; x \vdash x . \text{receive} ; x . \text{send} (5) : \text{int} \]
Type Rules

**RECEIVE**

\[ \Gamma; \{u : ?t\}; \{u\} \vdash u \cdot \text{receive} : t \]

**SEND**

\[ \Gamma; \Sigma; \{u\} \vdash e : t \]

\[ \Gamma; \Sigma \cdot \{u : !t\}; \{u\} \vdash u \cdot \text{send}(\ e\ ) : \text{Object} \]

**SEQUENCE**

\[ \Gamma; \Sigma S \vdash e : t \quad \Gamma; \Sigma' S \vdash e' : t', \]

\[ \Gamma; \Sigma \circ \Sigma'; S \vdash e; e' : t', \]

Thus,

\[ \Gamma; x : ?\text{bool}; x \vdash x \cdot \text{receive} : \text{bool} \]

\[ \Gamma; x : !\text{int}; x \vdash x \cdot \text{send}(5) : \text{int} \]

\[ \Gamma; x : ?\text{bool} . !\text{int}; x \vdash x \cdot \text{receive}; x \cdot \text{send}(5) : \text{int} \]

On the other hand, \( x \cdot \text{receive}; y \cdot \text{send}(5) \) will *not* typecheck.
Type Rules - Intricacies

We use the hot set to forbid the interleaving of sessions.

Thus,
We use the hot set to forbid the interleaving of sessions.

Thus,

```java
connect c1 s1 { 
  connect c2 s2{ 
    c2.send(3) 
    c1.send(5) 
  }
}
```

type incorrect
We use the hot set to forbid the interleaving of sessions.

Thus,

```latex
connect c1 s1 { 
  connect c2 s2{
    c2.send(3)
    c1.send(5) 
  };
}
```

type incorrect

but

```latex
connect c1 s1 { 
  connect c2 s1 {
    c2.send(3) 
    c1.send(5) 
  };
}
```

type correct
We use the hot set to forbid the interleaving of sessions.

Thus,

```java
connect c1 s1 {
    connect c2 s2{
        c2.send(3)
        c1.send(5)
    };
    c1.send(5)
};;
```

but

```java
connect c1 s1 {
    connect c2 s1 {
        c2.send(3)
        c1.send(5)
    };
    c1.send(5)
};;
```

and also

```java
connect c1 s1 {
    c2.send(3);
    c1.send(5)
};;
```

type incorrect
type correct
type correct

if c2 fresher than c1
In order to handle intricacies such as in previous slide, we have six non-structural rules.
In order to handle intricacies such as in previous slide, we have six non-structural rules.

*If you want the world to hear,*
*write your papers crystal clear,*
*then, add some ingenuities,*
*to show the world how difficult to do it is.*

PietHein
We can also infer session types, avoiding type annotations in “connect”
Outline

1. Sessions - background
   - Sessions Example: A purchasing protocol (Typical W3C example)
2. Moose primitives
3. Operational Semantics
   - Connect
   - Interactions
4. Moose types
   - Types for Web Service Example
5. The price of soundness
6. The price of progress
7. Type System
8. Properties
Properties

P0  Subject Reduction
Properties

P0 Subject Reduction

P1 communication errors cannot occur, i.e., there cannot be two sends or two receives on the same channel in parallel in two different threads;
Properties

P0  Subject Reduction
P1  communication errors cannot occur, i.e., there cannot be two sends or two receives on the same channel in parallel in two different threads;
P2  typeable 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.
Properties

P0  Subject Reduction

P1  communication errors cannot occur, i.e., there cannot be two sends or two receives on the same channel in parallel in two different threads;

P2  typeable 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  after a session has started, the required communications are always executed in the expected order.
Properties

P0  Subject Reduction for Threads

\[ \Gamma; \Sigma \vdash P; h \text{ and } P, h \rightarrow P', h' \text{ imply } \Gamma'; \Sigma' \vdash P'; h' \]

with \( \Gamma \subseteq \Gamma' \) and \( \Sigma \subseteq \Sigma' \).
Properties

**P0**  Subject Reduction for Threads

\[ \Gamma; \Sigma \vdash P; h \text{ and } P, h \rightarrow P', h' \text{ imply } \Gamma'; \Sigma' \vdash P'; h' \]

with \( \Gamma \subseteq \Gamma' \) and \( \Sigma \subseteq \Sigma' \).

**P2**  Progress

If \( P_0, \emptyset \rightarrow\rightarrow P, h \). Then one of the following holds:

- \( P, h \rightarrow P', h' \);
- In \( P \), all expressions are values;
- \( P \) throws a null pointer exception; or
- \( P \) stops with a connect waiting for its dual instruction, i.e., \( P \equiv E[\text{connect } c \hspace{1pt} s\{e\}] | Q. \)
Progress Proof

Key points:

- the correspondence of session primitives and session types;
- information on the current live channel of expressions;
- the active channels in a thread are visited in a scoped manner, where the most recent is visited first,
- knowing the active channel of a thread, we show that some “dual” expression will appear on a parallel thread, modulo specific exceptions;
- the two expressions can react, hence we get progress;
- the non-session cases are more straightforward.
Conclusions and Future Work

- is progress essential?
Conclusions and Future Work

- Is progress essential?
- The combination of higher order sessions, soundness, and progress is intricate,
Conclusions and Future Work

- is progress essential?
- the combination of higher order sessions, soundness, and progress is intricate,
- more work on combination of oo and sessions,
Conclusions and Future Work

- is progress essential?
- the combination of higher order sessions, soundness, and progress is intricate,
- more work on combination of oo and sessions,
- more work on timeout in connect,
Conclusions and Future Work

- is progress essential?
- the combination of higher order sessions, soundness, and progress is intricate,
- more work on combination of oo and sessions,
- more work on timeout in connect,
- prototype implementation (in progress).
Conclusions and Future Work

- is progress essential?
- the combination of higher order sessions, soundness, and progress is intricate,
- more work on combination of oo and sessions,
- more work on timeout in connect,
- prototype implementation (in progress).

for a longer version with proofs:

http://www.doc.ic.ac.uk/~dm04
Conclusions and Future Work

- is progress essential?
- the combination of higher order sessions, soundness, and progress is intricate,
- more work on combination of oo and sessions,
- more work on timeout in connect,
- prototype implementation (in progress).

for a longer version with proofs:

http://www.doc.ic.ac.uk/~dm04

Thank you!

Sophia Drossopoulou