Session Types for Object-Oriented Languages

joint work with Mariangiola Dezani (Torino), Nobuko Yoshida (Imperial), and Sophia Drossopoulou (Imperial)

Wednesday, 11 October 2006
Outline

- Background
- Example
- Syntax and Types
- Typing System
- Properties
- Future Work
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.


We integrated sessions in a small object calculus

Scenario: Item Purchasing  (Typical W3C example)

Exchange of data/objects between three participants (i.e., three “processes” running in parallel).
Class C {
    void m() {
        connect c s {
            c.send(5);
            new B.h();
            bool x := c.receive;
        }
    }
}
Class C {
    void m() {
        connect c s {
            c.send(5);
            new B.h();
            bool x := c.receive;
        }
    }
}
Class C {
    void m() {
        connect c s {
            c.send(5);
            new B.h();
            bool x := c.receive;
        }
    }
}
Basics of MOOSE

Class C {
    void m() {
        connect c s {
            c.send(5);
            new B.h();
            bool x := c.receive;
        }
    }
}
Class C {
    void m() {
        connect c s {
            c.send(5);
            new B.h();
            bool x := c.receive;
        }
    }
}
Class C {
    void m() {
        connect c s {
            c.send(5);
            new B.h();
            bool x := c.receive;
        }
    }
}
Basics of MOOSE

Class C {
    void m() {
        connect c s {
            c.send(5);
            new B.h();
            bool x := c.receive;
        }
    }
}
Class C {
    void m() {
        connect c s {
            c.send(5);
            new B.h();
            bool x := c.receive;
        }
    }
}
Class C {
    void m() {
        connect c s {
            c.send(5);
            new B.h();
            bool x := c.receive;
        }
    }
}

Notation: s = begin.!int.?bool.end
Class C {
    void m () {
        connect c s {
            c.send(5);
            new B.h();
            bool x := c.receive;
        }
    }
}

Notation: s = begin!.int.?bool.end
Class C {
    void m() {
        connect c s {
            c.send(5);
            new B.h();
            bool x := c.receive;
        }
    }
}

Notation: s = begin !int.?bool.end
Basics of MOOSE

Class C {
  void m() {
    connect c s {
      c.send(5);
      new B.h();
      bool x := c.receive;
    }
  }
}

Class D {
  void g() {
    connect c s' {
      // ... local computation
      this.f(c.receive);
      c.send(true);
    }
  }
}

Notation: s = begin.!int.?bool.end

suppose we are executing:  new C.m();  |  new D.g();
Class C {
    void m() {
        connect c s {
            c.send(5);
            new B.h();
            bool x := c.receive;
        }
    }
}

Class D {
    void g() {
        connect c s' {
            // ... local computation
            this.f(c.receive);
            c.send(true);
        }
    }
}

Notation: $s = \text{begin}!\text{int}.?\text{bool}\text{end}$

communication over the same channel
**Basics of MOOSE**

Class C {
    void m() {
        connect c s {
            c.send(5);
            new B.h();
            bool x := c.receive;
        }
    }
}

Class D {
    void g() {
        connect c s' {
            // ... local computation
            this.f(c.receive);
            c.send(true);
        }
    }
}

Notation: s = begin.!int.?bool.end

session types s and s' must ‘agree’
Basics of MOOSE

Class C {
    void m() {
        connect c s {
            c.send(5);
            new B.h();
            bool x := c.receive;
        }
    }
}

Class D {
    void g() {
        connect c s' {
            // ... local computation
            this.f(c.receive);
            c.send(true);
        }
    }
}

Notation: s = begin.!int.?bool.end

whenever one sends, the other must receive
Class C {
    void m() {
        connect c s {
            c.send(5);
            new B.h();
            bool x := c.receive;
        }
    }
}

Class D {
    void g() {
        connect c s' {
            c.send(true);
            // ... local computation
            this.f(c.receive);
            c.send(true);
        }
    }
}

Notation: $s = \text{begin}.!\text{int}.?\text{bool}.\text{end}$

types of exchanged data must always be as expected
Basics of MOOSE

Class C {
  void m() {
    connect c s {
      c.send(5);
      new B.h();
      bool x := c.receive;
    }
  }
}

Class D {
  void g() {
    connect c s' {
      // ... local computation
      this.f(c.receive);
      c.send(true);
    }
  }
}

Notation: s = begin.int.?bool.end  s' = begin.?int.!bool.end
Basics of MOOSE

Class C {
    void m() {
        connect c s {
            c.send(5);
            new B.h();
            bool x := c.receive;
        }
    }
}

Class D {
    void g() {
        connect c s' {
            // ... local computation
            this.f(c.receive);
            c.send(true);
        }
    }
}

Notation: s = begin.!int.?bool.end s' = begin.?int.!bool.end

s is dual to s' (the dual of s is written \(\overline{s}\))
Conditional Sessions

...  

\[
\text{connect } c \ s \ \{ \\
  c.\text{sendIf}\left(x > 5\right) \ \{ \\
    \text{int } n := c.\text{receive}; \ c.\text{send}(\text{true}); \\
  \} \\
  c.\text{send}(\text{false}); \\
\}\ \\
\]

...  

Note: \textit{sendIf}(e) matches with \textit{receiveIf}
... 

connect c s 
  
  c.sendIf(x > 5) 
  
  int n := c.receive; c.send(true);

  }

  c.send(false);

  }

  }...
Conditional Sessions

... 

connect c s { 
  c.sendIf(x > 5) { 
    int n := c.receive; c.send(true); 
  } 
  c.send(false); 
} 

...
...  

`connect c s {`

`  c.sendIf(x > 5) {`

`    int n := c.receive; c.send(true);`

`  }`

`  c.send(false);`

`  }

`  }

`  ...

`s =
begin`

`  !\langle`

`  ?int!.bool,`
Conditional Sessions

... connect c s {  
  c.sendIf(x > 5) {  
    int n := c.receive; c.send(true);  
  }  
  c.send(false);  
}  
...

s = begin  
!⟨  
?int.!bool,  
!bool  
}
Conditional Sessions

...

connect c s {
  c.sendIf(x > 5) {
    int n := c.receive; c.send(true);
  }
  c.send(false);
}

Notation: \( s = \text{begin}.!\langle?\text{int}!.\text{bool}, !\text{bool}\rangle.\text{end} \)
Conditional Sessions

... connect c s { c.receiveIf { int n := c.receive; c.send(true); }{ c.send(false); } }

Notation: s = begin.?⟨?int.!bool, !bool⟩.end
Conditional Sessions

...  
connect c s {  
  c.receiveIf {  
    int n := c.receive; c.send(true);  
  }  
  c.send(false);  
}  
...

Notation: s = begin.⟨⟨?int.!bool, !bool⟩⟩.end

We also have sendWhile(e) and receiveWhile for iteration
Session over Session

```c
connect c1 begin.!bool.!int.end {
    c1.send(true);
}

connect c1 begin.?bool.?int.end { c1.receive; c1.receive }
```
Session over Session

connect c1 begin.!bool.!int.end { 
    c1.send(true);

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

connect c1 begin.?bool.?int.end { c1.receive; c1.receive}
Session over Session

```plaintext
connect c1 begin.!bool.!int.end {
    c1.send(true);
    connect c2 begin.!(!int.end).end {
        c2.sendS(c1);
    }
}

connect c1 begin.?bool.?int.end {
    c1.receive; c1.receive
}

connect c2 begin.?(!int.end).end {
    c2.receiveS(x) {
        x.send(5);
    }
}
```
Methods with Session Parameters

\[
\text{connect } c1 \begin{array}{c}
\text{begin.} \text{!bool.} \text{!int.} \text{end} \\
\quad \text{c1.send(true);} \\
\end{array}
\]

\[
\text{connect } c1 \begin{array}{c}
\text{begin.} \text{?bool.} \text{?int.} \text{end} \\
\quad \text{c1.receive; c1.receive} \\
\end{array}
\]
Methods with Session Parameters

```plaintext
connect c1 begin.!bool.!int.end { 
    c1.send(true);
    new C.m(c1);
}

connect c1 begin.?bool.?int.end { c1.receive; c1.receive
```
Methods with Session Parameters

```java
connect c1 begin.!bool.!int.end { 
    c1.send(true);
    new C.m(c1);
} 

connect c1 begin.?bool.?int.end { c1.receive; c1.receive }

Class C {
    void m(!int.end x) { x.send(5); }
}
```
Concurrent programming support

Class D {
    void m ( x ) {
        spawn {
            ... 
            connect x s { x.send( .. ) .. }
        }
        connect x s { .. x.receive; .. }
    }
}
Class D {
    void m((s, \bar{s}) x) {
        spawn {
            ...
            connect x s { x.send(..) .. }
        }
        connect x \bar{s} { ..x.receive; .. }
    }
}
Concurrent programming support

Class D {
    void m((s, ñ) x) {
        spawn {
            ...
            connect x s { x.send(..) .. }
        }

        connect x ñ { ..x.receive; .. }
    }
}

Most direct way to call it is using new D.m(new (s, ñ));
Syntax of types

\[ 
\begin{align*}
\dagger &::= ! | ? & \text{direction} \\
\pi &::= \varepsilon | \pi.\pi | \dagger t | \dagger \langle \pi, \pi \rangle & \text{partial session type} \\
& | \dagger \langle \pi \rangle^* | \dagger (\rho) \\
\rho &::= \pi.\text{end} | \pi.\dagger \langle \rho, \rho \rangle & \text{ended session type} \\
\eta &::= \pi | \rho & \text{running session type} \\
s &::= \text{begin} \cdot \rho & \text{shared session type} \\
t &::= C | \text{bool} | s | (s, \overline{s}) & \text{standard type}
\end{align*}
\]
Starting a Session

\[ \ldots \text{connect } c s \{ e_1 \} \ldots | \ldots \text{connect } c s \{ e_2 \} \ldots, h \]
Starting a Session

\[
\ldots \text{connect } c \ s \{e_1\} \ldots \mid \ldots \text{connect } c \ \bar{s} \{e_2\} \ldots, \ h \\
\downarrow \\
\ldots e_1[c'/c] \ldots \mid \ldots e_2[c'/c] \ldots, \ h \cdot c' \\
\]

\(c'\) is fresh (i.e., not in the heap \(h\))
we call these live channels
Starting a Session

... connect \( c \leftarrow \{ e_1 \} \) ... | ... connect \( c \leftarrow \{ e_2 \} \) ..., \( h \)

\[ \downarrow \]

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

\( c' \) is fresh (i.e., not in the heap \( h \) )
we call these live channels

recall that \( c \) is shared

freshness of \( c' \) guarantees that \( e_1 \) and \( e_2 \)
only interact with each other
... $c . \text{send}(v) \ldots \mid \ldots c . \text{receive} \ldots$, $h$
Interactions

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

\[ \downarrow \]

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

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

\[ \downarrow \]

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

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

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

\[ \downarrow \]

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

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

\[ \downarrow \]

\[ \ldots e_1 | e_3 \ldots, h \]
session BuyProduct =
  begin.!String.?double.!⟨Address.?DeliveryDetails.end, end⟩

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

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

Seller’s viewpoint of the Seller-Shipper interaction
Implementation fits in one A4 page…
Typing

Γ; Σ; 𝒦 ⊢ e : t
Typing

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

standard environment
Typing

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

expression
Typing

\( \Gamma; \Sigma; S \vdash e : t \)
Typing

Γ; Σ; 𝕊 ⊢ e : t

session environment
records which channels are used in e, and in what way
Typing

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

hot set

identifies the current/active session channel
used to prevent interleaving of sessions
Typing

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

\[ \text{SEQ} \]

\[
\begin{align*}
\Gamma; \Sigma; S & \vdash e : t \\
\Gamma; \Sigma'; S & \vdash e' : t' \\
\hline
\Gamma; \Sigma \cdot \Sigma'; S & \vdash e; e' : t'
\end{align*}
\]
Typing

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

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

\[ \Gamma; \mathbf{x} : \text{!nat}; \mathbf{x} \vdash \mathbf{x}.\text{send}(5) : \text{nat} \]

\[ \Gamma; \mathbf{x} : \text{?bool}; \text{!nat}; \mathbf{x} \vdash \mathbf{x}.\text{receive}; \mathbf{x}.\text{send}(5) : \text{nat} \]
We can also infer session types, avoiding type annotations in "connect"
Properties

P0  Subject Reduction
Properties

P0  Subject Reduction

P1  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;
Properties

**P0**  Subject Reduction

**P1**  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**  typable threads can always *progress* unless one of the following situations occurs:
Properties

P0 Subject Reduction

P1 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 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.
Properties

P0 Subject Reduction

P1 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 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 after a session has started the required communications are always executed in the expected order.
Properties (cont.)

**P0** Subject Reduction for Threads

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

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

**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 \( P_0, \emptyset \rightarrow P, h \). 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., \( P \equiv E[\text{connect } c s \{ e \}] | Q \).
Future Work

- exception handling and propagation
- timeout in connect
- prototype implementation

for a longer version with proofs:

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