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

Introduction

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

Kohei Honda. *Types for Dyadic Interaction*.
CONCUR'93, LNCS 715, pages 509–523, Springer-Verlag, 1993

Introduction

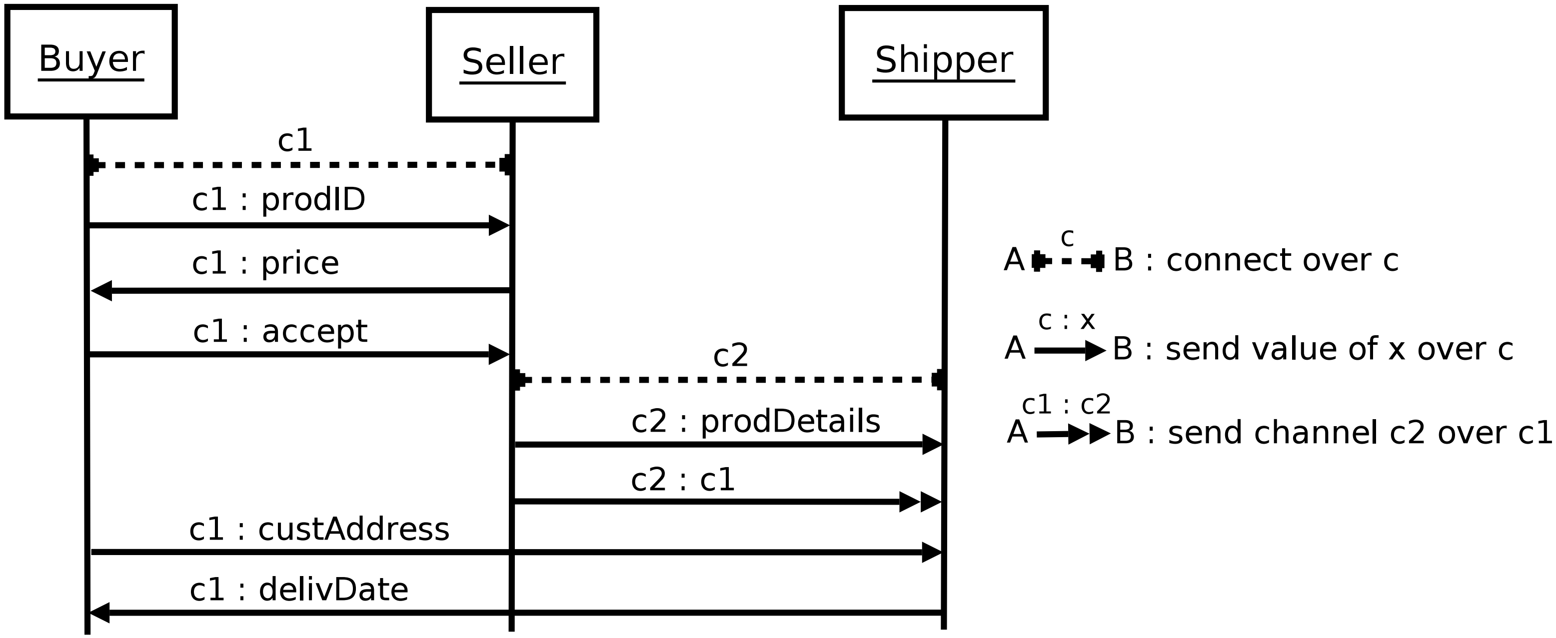
A session describes a communication protocol between two parties, that takes place over a single connection

Kohei Honda. *Types for Dyadic Interaction*. CONCUR'93, LNCS 715, pages 509–523, Springer-Verlag, 1993

We integrated sessions in a small object calculus

Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. *Session Types for Object-Oriented Languages*. ECOOP 2006.

Protocol Example



Scenario: Item Purchasing (Typical W3C example)

Exchange of data/objects between three participants (i.e., three "processes" running in parallel).

Basics of MOOSE

```
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;
    }
  }
}
```

c is a channel

it represents the connection

send and receive are the basic primitives for communication

Basics of MOOSE

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

s abbreviates a session type
(that we will see next)

the session body must
'agree' with s

Basics of MOOSE

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

$\mathbf{s} =$

Basics of MOOSE

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

$\mathbf{s} =$

begin

Basics of MOOSE

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

$\mathbf{s}$ =

begin

!int

Basics of MOOSE

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

Basics of MOOSE

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

$$\mathbf{s} = \mathsf{begin}\ !\mathsf{int}\ ?\mathsf{bool}\ \mathsf{end}$$

Basics of MOOSE

Code	Session type
Class C {	
void m () {	$\mathbf{s} \ =$
connect c s {	begin
c.**send**(5);	!int
new B.h ();	
bool x := c.**receive**;	?bool
}	end
}	
}	

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;
    }
  }
}
```

when can this connect block execute?

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

Basics of MOOSE

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

when can this connect block execute?

who can it communicate with?

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

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

suppose we are executing: **new** C.m(); | **new** D.g();

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

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

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

whenever one sends, the other must receive

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

types of exchanged data must always be as expected

Basics of MOOSE

```
Class C {                              Class D {
  void m () {                            void g () {
    connect c s {                          connect c s' {
      c.send(5);                             // ... local computation
      new B.h ();                            this.f( c.receive );
      bool x := c.receive;                   c.send(true);
    }                                      }
  }                                      }
}                                      }
```

Notation: $s = \mathsf{begin.!int.?bool.end}$ $\quad s' = \mathsf{begin.?int.!bool.end}$

Basics of MOOSE

```
Class C {                          Class D {
  void m () {                        void g () {
    connect c s {                      connect c s' {
      c.send(5);                         // ... local computation
      new B.h ();                        this.f( c.receive );
      bool x := c.receive;               c.send(true);
    }                                  }
  }                                  }
}                                  }
```

Notation: $s = \mathsf{begin.!int.?bool.end}$ $\quad s' = \mathsf{begin.?int.!bool.end}$

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

Conditional Sessions

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

$\mathbf{s} =$

Note: **sendIf** (**e**) matches with **receiveIf**

Conditional Sessions

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

s =

begin

Conditional Sessions

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

$\mathbf{s} =$

begin

$!\langle$

Conditional Sessions

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

$\mathbf{s} =$

begin

$!\langle$

$?\mathsf{int}.!\mathsf{bool},$

Conditional Sessions

$\ldots$

$\textbf{connect}\ \mathsf{c}\ \mathsf{s}\ \{$

$\quad \mathsf{c}.\textbf{sendIf}(\mathsf{x} > 5)\ \{$

$\quad\quad \mathsf{int\ n}\ := \mathsf{c}.\textbf{receive};\ \mathsf{c}.\textbf{send}(\mathsf{true});$

$\quad \}\{$

$\quad\quad \mathsf{c}.\textbf{send}(\mathsf{false});$

$\quad \}$

$\}$

$\ldots$

$\textbf{s} =$

begin

$!\langle$

$\quad ?\mathsf{int}.!\mathsf{bool},$

$\quad !\mathsf{bool}$

Conditional Sessions

$$
\begin{array}{l}
\ldots \\
\textbf{connect}\ \mathsf{c}\ \mathsf{s}\ \{ \\
\quad \mathsf{c}.\textbf{sendIf}(\mathsf{x} > 5)\ \{ \\
\qquad \mathsf{int}\ \mathsf{n}\ := \mathsf{c}.\textbf{receive};\ \mathsf{c}.\textbf{send}(\mathsf{true}); \\
\quad \}\{ \\
\qquad \mathsf{c}.\textbf{send}(\mathsf{false}); \\
\quad \} \\
\} \\
\ldots
\end{array}
\qquad
\begin{array}{l}
\mathbf{s}\ = \\
\mathsf{begin} \\
!\langle \\
\quad ?\mathsf{int}.!\mathsf{bool}, \\
\\
\quad !\mathsf{bool} \\
\ \rangle \\
\mathsf{end}
\end{array}
$$

Notation: $\mathsf{s} = \mathsf{begin}.!\langle ?\mathsf{int}.!\mathsf{bool}, !\mathsf{bool}\rangle.\mathsf{end}$

Conditional Sessions

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

$\mathbf{s} =$

begin

$?\langle$

$?\mathsf{int}.!\mathsf{bool},$

$!\mathsf{bool}$

$\rangle$

end

Notation: $\mathsf{s} = \mathsf{begin}.?\langle ?\mathsf{int}.!\mathsf{bool}, !\mathsf{bool}\rangle.\mathsf{end}$

Conditional Sessions

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

$$
\mathbf{s} = \mathsf{begin}\; ?\langle\; ?\mathsf{int}.!\mathsf{bool},\;\; !\mathsf{bool}\; \rangle\; \mathsf{end}
$$

Notation: $\mathsf{s} = \mathsf{begin}.?\langle ?\mathsf{int}.!\mathsf{bool}, !\mathsf{bool}\rangle.\mathsf{end}$

We also have **sendWhile** (**e**) and **receiveWhile** for iteration

Session over Session

```
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

```
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

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

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

Methods with Session Parameters

```
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

```
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  s̄  {..x.receive;..}
  }
}
```

Concurrent programming support

```
Class D {
    void m( (s, s̄) x ) {
        spawn {
            ...
            connect  x  s  { x.send(..).. }
        }
        connect  x  s̄  { ..x.receive; .. }
    }
}
```

Concurrent programming support

```
Class D {
    void m( (s, s̄) x ) {
        spawn {
            . . .
            connect  x  s  { x.send(..).. }
        }
        connect  x  s̄  { ..x.receive; .. }
    }
}
```

Most direct way to call it is using new D.m(new (s, $\bar{s}$));

Syntax of types

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

Starting a Session

$$\ldots\textbf{connect}\,\mathsf{c}\ \mathsf{s}\{\mathsf{e}_1\}\ldots \mid \ldots\textbf{connect}\,\mathsf{c}\ \overline{\mathsf{s}}\{\mathsf{e}_2\}\ldots,\ h$$

Starting a Session

$$\ldots \textbf{connect}\ \mathsf{c}\ \mathsf{s}\{\mathsf{e}_1\}\ldots \mid \ldots \textbf{connect}\ \mathsf{c}\ \overline{\mathsf{s}}\{\mathsf{e}_2\}\ldots,\ h$$

$$\downarrow$$

$$\ldots \mathsf{e}_1[\mathsf{c}'/\mathsf{c}]\ldots \mid \ldots \mathsf{e}_2[\mathsf{c}'/\mathsf{c}]\ldots,\ h \cdot \mathsf{c}'$$

c' is fresh (i.e., not in the heap h)

we call these live channels

Starting a Session

$$\ldots \textbf{connect}\; c\; s\{e_1\}\ldots \mid \ldots \textbf{connect}\; c\; \bar{s}\{e_2\}\ldots,\; h$$

$$\downarrow$$

$$\ldots e_1[c'/c]\ldots \mid \ldots e_2[c'/c]\ldots,\; h \cdot c'$$

c' is fresh (i.e., not in the heap h)

we call these live channels

recall that c is shared

freshness of c' guarantees that e_1 and e_2

only interact with each other

Interactions

$$\ldots \, \mathsf{c}\,.\mathbf{send}(\mathsf{v}) \ldots \mid \ldots \, \mathsf{c}\,.\mathbf{receive} \ldots, \; h$$

Interactions

$$\ldots \mathsf{c}\,.\mathbf{send}(\mathsf{v}) \ldots \;|\; \ldots \mathsf{c}\,.\mathbf{receive} \ldots, h$$

$$\downarrow$$

$$\ldots \mathsf{null} \ldots \;|\; \ldots \mathsf{v} \ldots, h$$

Interactions

$$\ldots \texttt{c}.\mathbf{send}(\texttt{v}) \ldots \mid \ldots \texttt{c}.\mathbf{receive} \ldots, h$$

$$\downarrow$$

$$\ldots \texttt{null} \ldots \mid \ldots \texttt{v} \ldots, h$$

$$\ldots \texttt{c}.\mathbf{sendIf}(\texttt{true})\{\texttt{e}_1\}\{\texttt{e}_2\} \mid \texttt{c}.\mathbf{receiveIf}\{\texttt{e}_3\}\{\texttt{e}_4\} \ldots, h$$

Interactions

$$\ldots \mathsf{c}.\mathbf{send}(\mathsf{v}) \ldots \mid \ldots \mathsf{c}.\mathbf{receive} \ldots, h$$

$$\downarrow$$

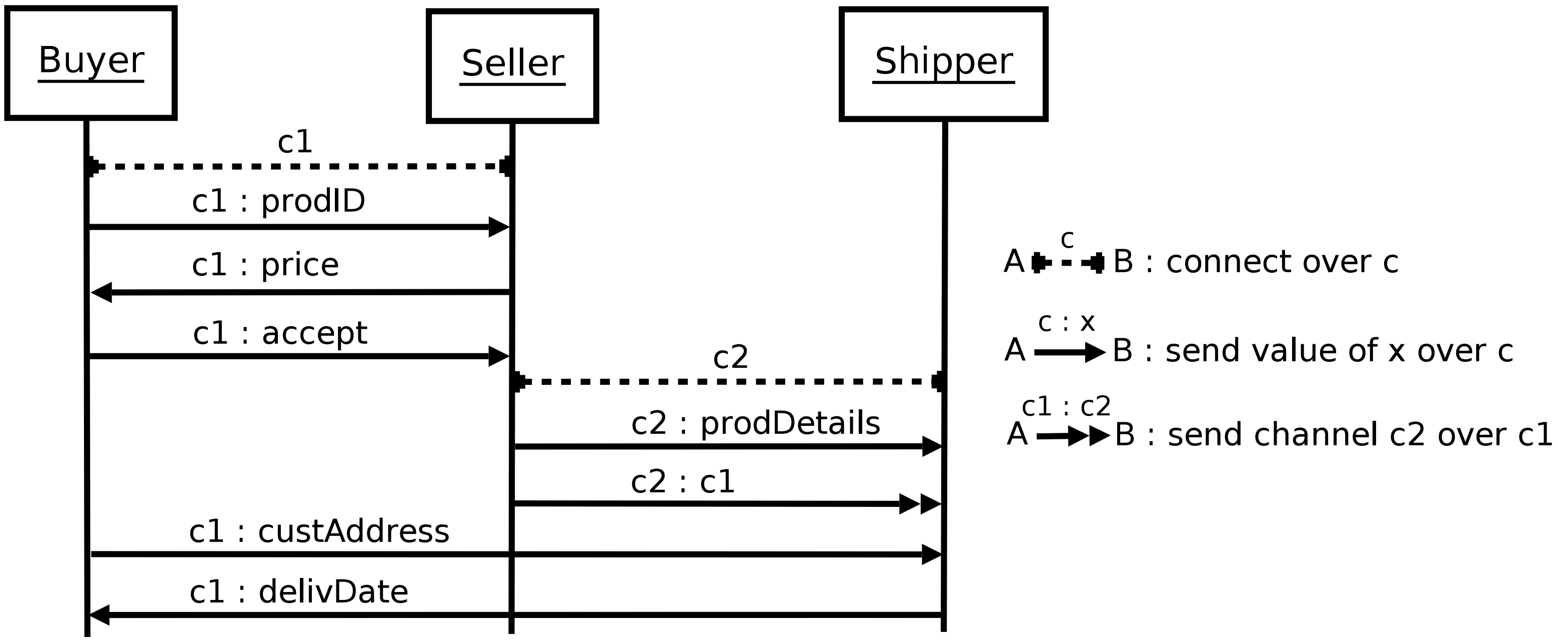
$$\ldots \mathsf{null} \ldots \mid \ldots \mathsf{v} \ldots, h$$

$$\ldots \mathsf{c}.\mathbf{sendIf}(\mathsf{true})\{\mathsf{e}_1\}\{\mathsf{e}_2\} \mid \mathsf{c}.\mathbf{receiveIf}\{\mathsf{e}_3\}\{\mathsf{e}_4\} \ldots, h$$

$$\downarrow$$

$$\ldots \mathsf{e}_1 \mid \mathsf{e}_3 \ldots, h$$

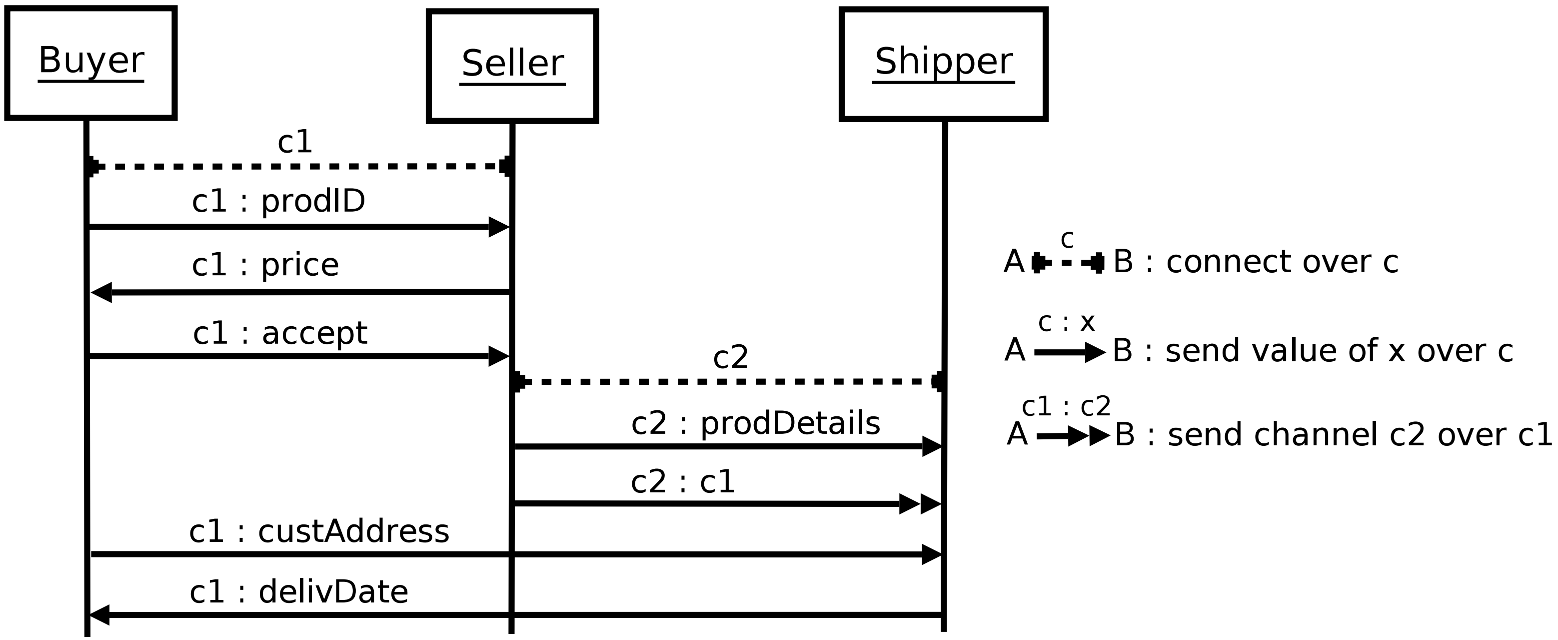
Types for Web Service Example



$$\text{session} \quad \text{BuyProduct} = \text{begin}.!\text{String}.?\text{double}.!\langle !\text{Address}.?\text{DeliveryDetails}.\text{end}, \text{end}\rangle$$

Buyer's viewpoint of the Buyer-Seller interaction

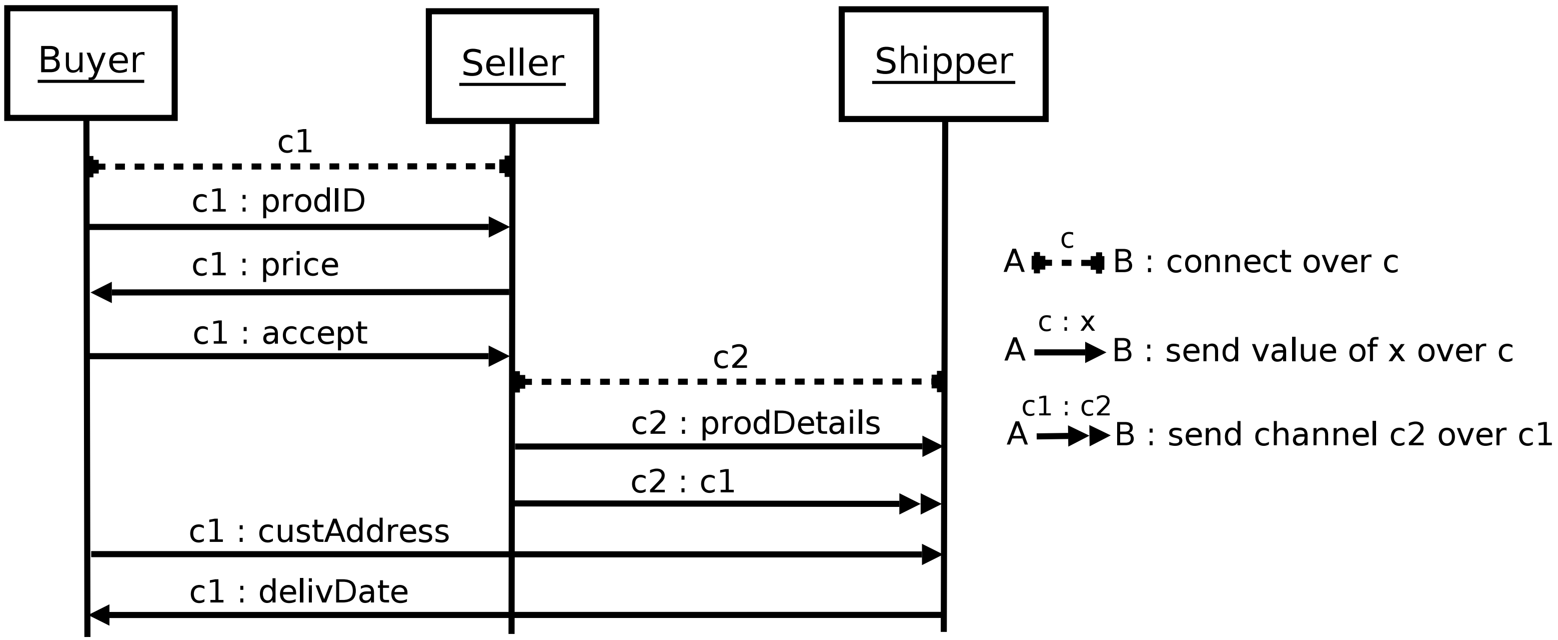
Types for Web Service Example



$$\begin{array}{l}\mathsf{session}\quad \mathsf{RequestDelivery} = \\ \quad \mathsf{begin.!ProductDetails.!(?Address.!DeliveryDetails.end).end}\end{array}$$

Seller's viewpoint of the Seller-Shipper interaction

Types for Web Service Example



Implementation fits in one A4 page…

Typing

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

Typing

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

standard environment

Typing

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

expression

Typing

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

type

Typing

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

session environment

records which channels are used in e , and in what way

Typing

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

hot set

identifies the current/active session channel

used to prevent interleaving of sessions

Typing

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

SEQ

$$\frac{\Gamma;\Sigma;\mathcal{S} \vdash \mathsf{e} : \mathsf{t} \qquad \Gamma;\Sigma';\mathcal{S} \vdash \mathsf{e}' : \mathsf{t}'}{\Gamma;\Sigma\cdot\Sigma';\mathcal{S} \vdash \mathsf{e};\mathsf{e}' : \mathsf{t}'}$$

Typing

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

$$\frac{\Gamma; \mathsf{x} : ?\mathsf{bool}; \mathsf{x} \vdash \mathsf{x}.\mathbf{receive} : \mathsf{bool} \qquad \Gamma; \mathsf{x} : !\mathsf{nat}; \mathsf{x} \vdash \mathsf{x}.\mathbf{send}(5) : \mathsf{nat}}{\Gamma; \underline{\mathsf{x} : ?\mathsf{bool}.!\mathsf{nat}}; \mathsf{x} \vdash \mathsf{x}.\mathbf{receive}; \mathsf{x}.\mathbf{send}(5) : \mathsf{nat}}$$

Typing

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

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

$\Gamma;\Sigma \vdash P;h$ and $P,h \longrightarrow P',h'$ imply $\Gamma';\Sigma' \vdash P';h'$ with $\Gamma \subseteq \Gamma'$ and $\Sigma \subseteq \Sigma'$.

Properties (cont.)

P0 Subject Reduction for Threads

$\Gamma;\Sigma \vdash P;h$ and $P,h \longrightarrow P',h'$ imply $\Gamma';\Sigma' \vdash P';h'$ with $\Gamma \subseteq \Gamma'$ and $\Sigma \subseteq \Sigma'$.

P2 Progress

If $P_0,\emptyset \longrightarrow\!\!\!\!\rightarrow P,h$. Then one of the following holds:

- $P,h \longrightarrow P',h'$;
- 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.*, $P \equiv E[\textbf{connect}\ \mathsf{c}\ \mathsf{s}\{\mathsf{e}\}] \,|\, Q$.

Future Work

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

for a longer version with proofs:

http://www.doc.ic.ac.uk/∼dm04