

Session Types for Object-Oriented Languages

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

Wednesday, 7 June 2006



EPSRC Engineering and Physical Sciences
Research Council

Introduction

Session types are **process** types

Introduction

Session types are **process** types

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

Introduction

Session types are **process** types

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

Session types are **process** types

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**

Introduction

Session types are **process** types

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

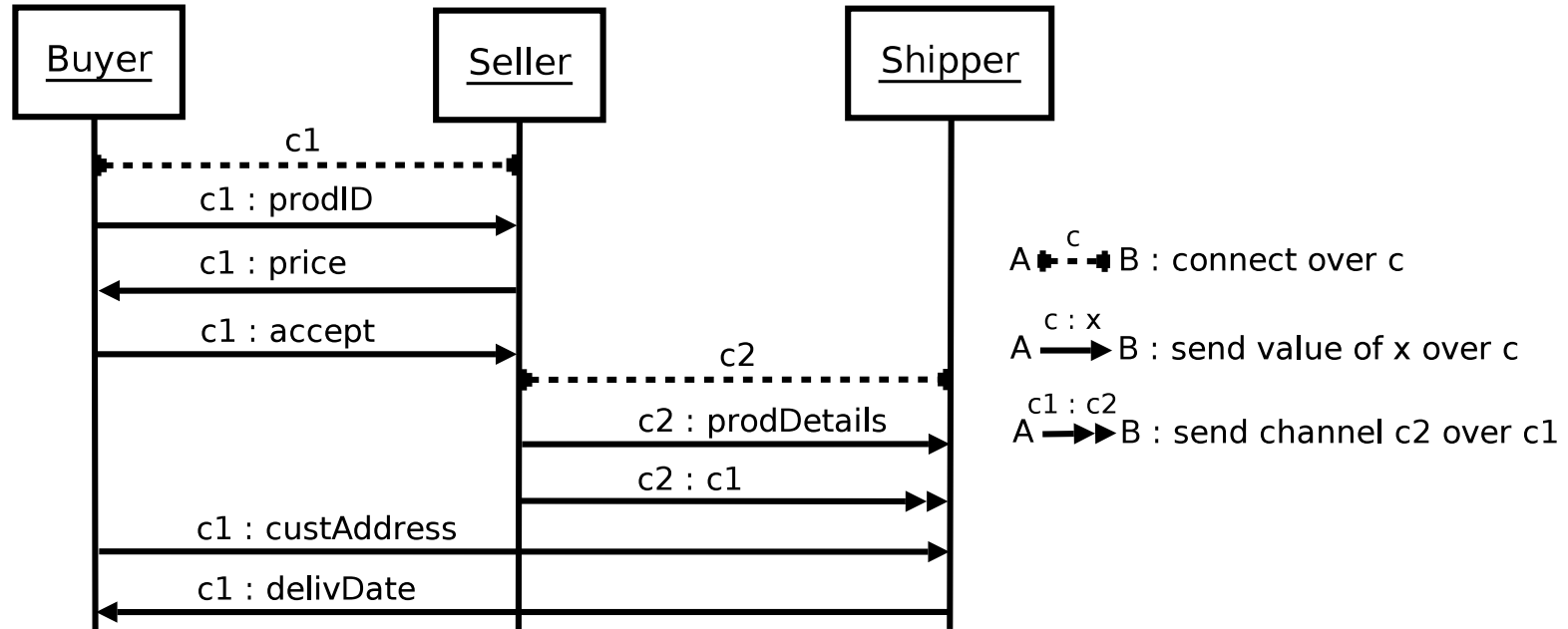
Kohei Honda. *Types for Dyadic Interaction*.

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*. To appear at 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;  
    }  
  }  
}
```

s =

Basics of MOOSE

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

```
s =  
begin
```

Basics of MOOSE

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

```
s =  
begin  
!int
```

Basics of MOOSE

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

```
s =  
begin  
!int  
?bool
```

Basics of MOOSE

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

```
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;  
    }  
  }  
}
```

```
s =  
begin  
!int  
?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: **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?

who can it communicate with?

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

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

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

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

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

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

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

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

Conditional Sessions

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

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);  
}  
}  
...
```

```
s =  
begin  
!⟨
```

Conditional Sessions

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

```
s =  
begin  
!⟨  
  ?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);
}
}
```

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

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);  
}  
}
```

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

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);
}
}
```

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

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

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  $\bar{s}$  {..x.receive;..}  
  }  
}
```

Concurrent programming support

```
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,  $\bar{s}$ ) x ) {  
    spawn {  
      ...  
      connect x s { x.send(..).. }  
    }  
    connect x  $\bar{s}$  { ..x.receive;.. }  
  }  
}
```

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

Starting a Session

... **connect** c s $\{e_1\}$... | ... **connect** c \bar{s} $\{e_2\}$..., h

Starting a Session

... **connect** c $s\{e_1\}$... | ... **connect** c $\bar{s}\{e_2\}$..., h



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

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

Starting a Session

... **connect** c $s\{e_1\}$... | ... **connect** c $\bar{s}\{e_2\}$..., h



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

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

Interactions

... **c** .send(*v*) ... | ... **c** .receive ..., *h*

Interactions

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



$\dots \mathbf{null} \dots \mid \dots v \dots, h$

Interactions

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



$\dots \mathbf{null} \dots \mid \dots v \dots, h$

$\dots c.\mathbf{sendIf}(\mathbf{true})\{e_1\}\{e_2\} \mid c.\mathbf{receiveIf}\{e_3\}\{e_4\} \dots, h$

Interactions

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



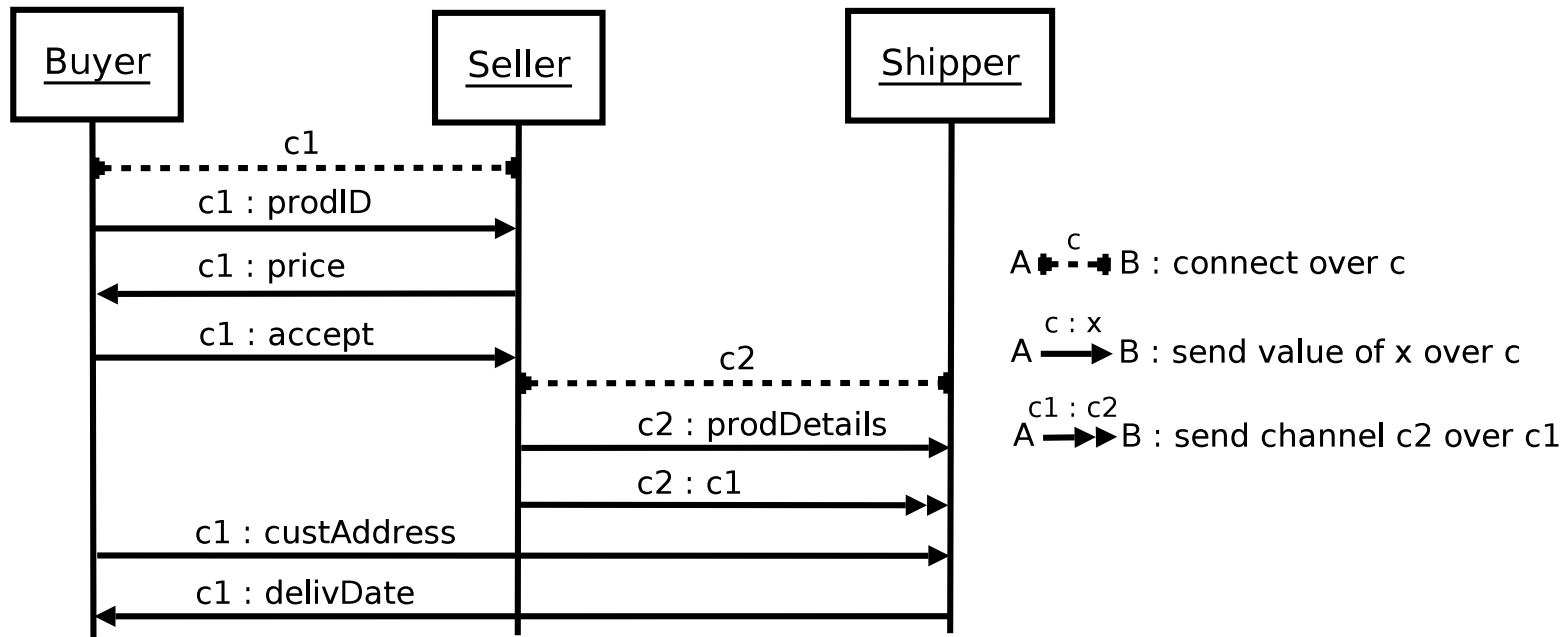
$\dots \mathbf{null} \dots \mid \dots v \dots, h$

$\dots c.\mathbf{sendIf}(\mathbf{true})\{e_1\}\{e_2\} \mid c.\mathbf{receiveIf}\{e_3\}\{e_4\} \dots, h$



$\dots e_1 \mid e_3 \dots, h$

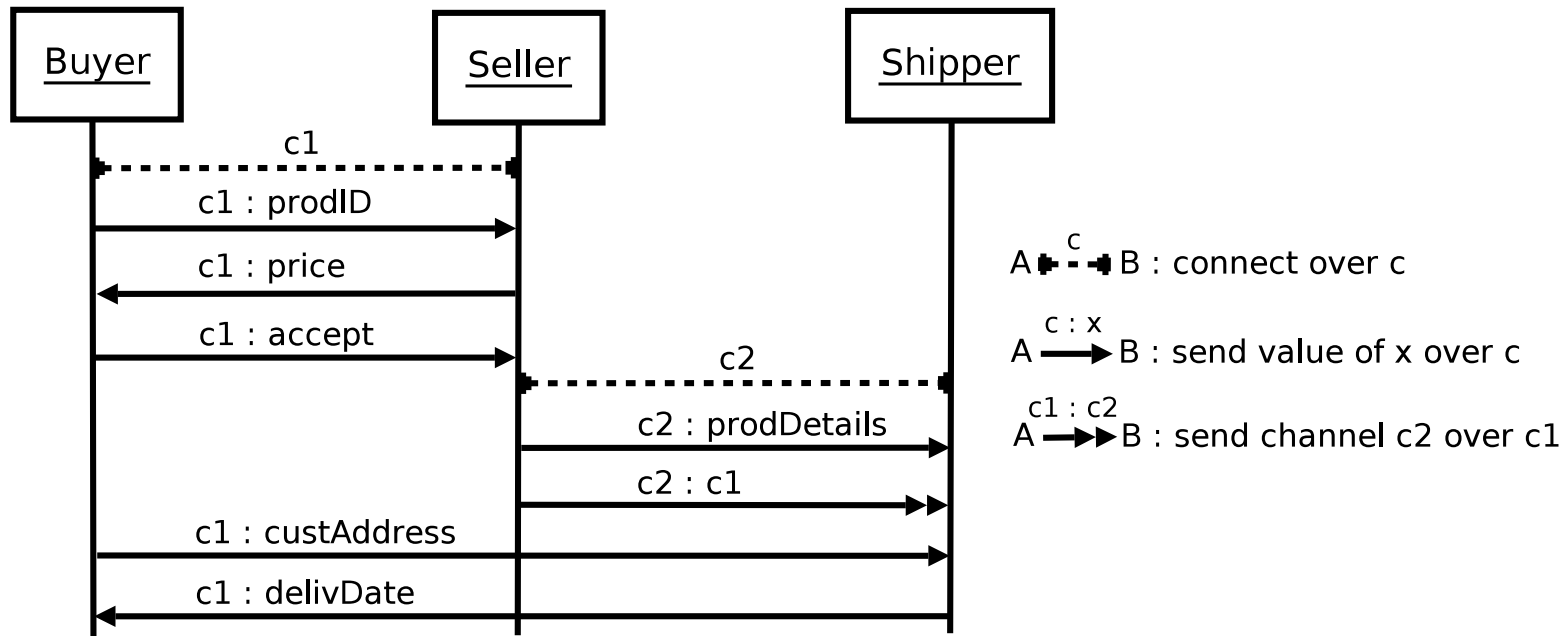
Types for Web Service Example



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

Buyer's viewpoint of the Buyer-Seller interaction

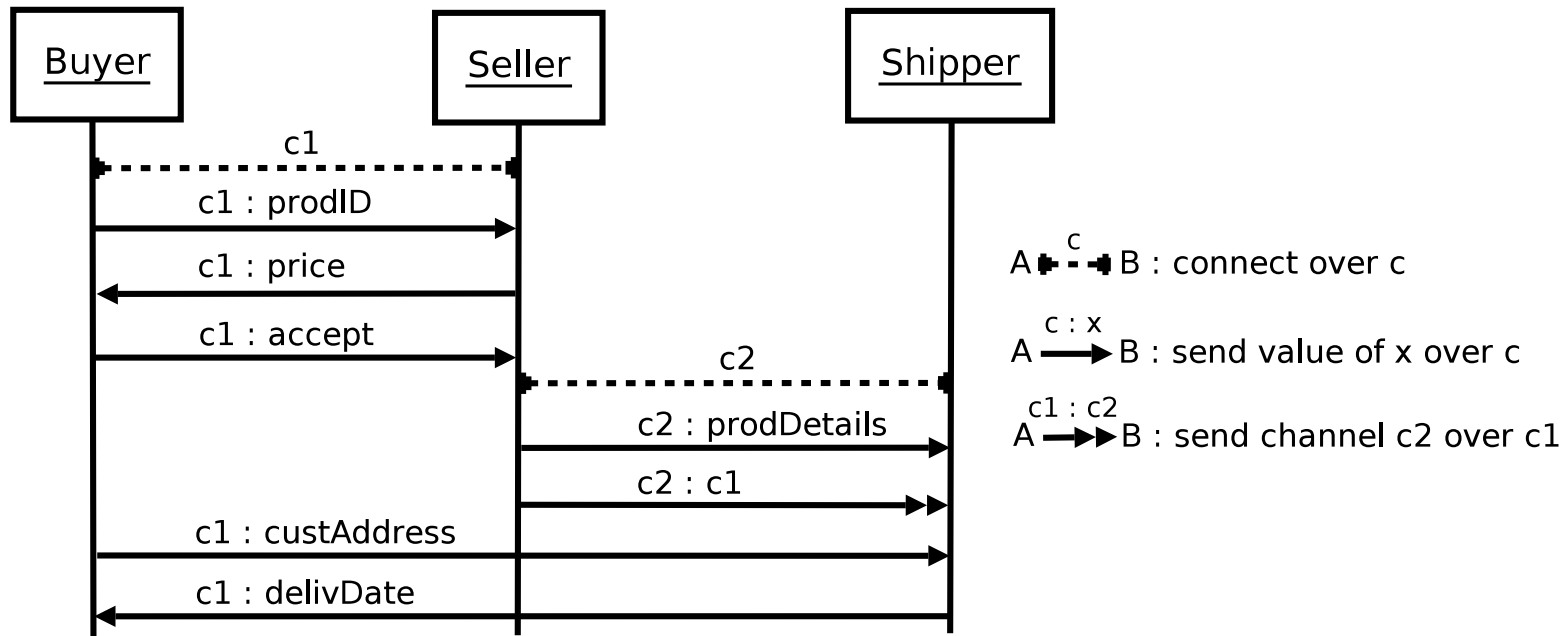
Types for Web Service Example



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

Seller's viewpoint of the Seller-Shipper interaction

Types for Web Service Example



Implementation fits in one A4 page...

Typing

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

Typing

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

standard environment



Typing

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

expression



Typing

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

type



Typing

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


session consumption environment

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

Typing

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

hot set

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

Typing

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

RECEIVE

$$\Gamma; \{ u : ?t \}; \{ u \} \vdash \mathbf{u.receive} : t$$

Typing

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

CONN

$$\Gamma; \emptyset; \emptyset \vdash u : \text{begin}.\rho \quad \Gamma \setminus u ; \Sigma, u : \rho; \{u\} \vdash e : t$$

$$\Gamma; \Sigma; \emptyset \vdash \mathbf{connect} u \text{ begin}.\rho \{e\} : t$$

Typing

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

CONN

$$\Gamma; \emptyset; \emptyset \vdash u : \text{begin}.\rho \quad \Gamma \setminus u ; \Sigma, u : \rho; \{u\} \vdash e : t$$

$$\Gamma; \Sigma; \emptyset \vdash \mathbf{connect} u \text{ begin}.\rho \{e\} : t$$

We can also infer session types, avoiding type annotations

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**.

Future Work

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

Limitations

- only nested sessions allowed, no interleaving
- only one session channel parameter in methods

Checking Existing Code

- Inference, checking, and extraction of protocols by analysing standard C/Java source code that uses sockets and threads, also in the presence of aliasing;
- Checking of restrictions in C/Java, e.g. of interleaving, and transformation when possible;
- Dependency analysis may also allow lifting restrictions on the number of linear method parameters.

Questions & Further Information

Questions ?

for more information and downloads:

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

Questions & Further Information

Questions ?

for more information and downloads:

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

Questions & Further Information

Questions ?

for more information and downloads:

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