FUNDAMENTALS OF SESSION TYPES

Vasco T. Vasconcelos
University of Lisbon

Formal Methods for the Design of
Computer, Communication and Software Systems
Bertinoro, June 4, 2009

CHALLENGE

- Among the many problems faced in developing web-service based applications, there is a little one that this talk addresses:
  - Formally describing the protocol between a service provider a client
  - Making sure, at compile time, that a program follows the protocol

BINARY PROTOCOLS ONLY

- We concentrate on binary protocols, involving exactly one service provider and one client (at a time)
- Protocols demanding three or more partners are the topic of the next talk

OUTLINE

Part I _ The practice

Where, based on an example, we describe a simple service and program it

Part 2 _ The theory

Where we discusss the technology behind a compiler that makes sure programs conform to service descriptions

PART I
THE PRACTICE

I.

PROTOCOL DESCRIPTION

OUR RUNNING EXAMPLE

- A simplified distributed auction system with three kinds of players:
  - Sellers that want to sell items
  - Auctioneers that sell items on their behalf
  - Bidders that bid for an item being auctioned

THE SELLER'S PROTOCOL

THE SELLER'S PROTOCOL

⊕{*selling*:

THE SELLER'S PROTOCOL

Select option *selling* on the auctioneer

⊕{*selling*:

THE SELLER'S PROTOCOL

⊕{*selling*:

THE SELLER'S PROTOCOL

$\oplus\{$*selling*: !Item

THE SELLER'S PROTOCOL

Send the item to be sold

⊕{*selling*: !Item

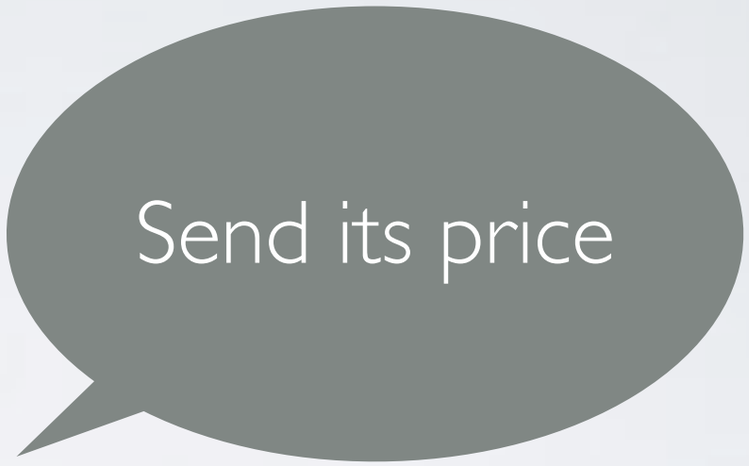
THE SELLER'S PROTOCOL

$\oplus\{$*selling*: !Item

THE SELLER'S PROTOCOL

⊕{*selling*: !Item.!Price.

THE SELLER'S PROTOCOL



⊕{*selling*: !Item.!Price.

THE SELLER'S PROTOCOL

⊕{*selling*: !Item.!Price.

THE SELLER'S PROTOCOL

$\oplus\{$*selling*: !Item.!Price.
$\&\{$*sold*:

THE SELLER'S PROTOCOL

$\oplus\{selling$: !Item.!Price.
$\&\{sold$:

If the item was sold...

THE SELLER'S PROTOCOL

$\oplus\{$*selling*: !Item.!Price.
$\&\{$*sold*:

THE SELLER'S PROTOCOL

$\oplus\{$*selling*: !Item.!Price.
$\&\{$*sold*: ?Price

THE SELLER'S PROTOCOL

$\oplus\{$*selling*: !Item.!Price.
$\&\{$*sold*: ?Price

Read the price

THE SELLER'S PROTOCOL

$\oplus\{$*selling*: !Item.!Price.
$\&\{$*sold*: ?Price

THE SELLER'S PROTOCOL

$$\oplus\{\mathit{selling}\colon !\mathrm{Item}.!\mathrm{Price}.\ \&\{\mathit{sold}\colon ?\mathrm{Price}.\mathbf{end},$$

THE SELLER'S PROTOCOL

$\oplus\{$*selling*: !Item.!Price.
$\&\{$*sold*: ?Price.**end**,

... and terminate

THE SELLER'S PROTOCOL

$$\oplus\{\mathit{selling}\colon !\mathrm{Item}.!\mathrm{Price}.\ \&\{\mathit{sold}\colon ?\mathrm{Price}.\mathbf{end},$$

THE SELLER'S PROTOCOL

$$\oplus\{\mathit{selling}\colon !\mathrm{Item}.!\mathrm{Price}.\ \&\{\mathit{sold}\colon ?\mathrm{Price}.\mathbf{end},\ \mathit{notSold}\colon \mathbf{end}\}\}$$

THE SELLER'S PROTOCOL

$$\oplus\{\mathit{selling}\colon !\mathrm{Item}.!\mathrm{Price}.\ \&\{\mathit{sold}\colon ?\mathrm{Price}.\mathbf{end},\ \mathit{notSold}\colon \mathbf{end}\}\}$$

If not sold terminate

THE SELLER'S PROTOCOL

$$\oplus\{\mathit{selling}\colon !\mathrm{Item}.!\mathrm{Price}.\ \&\{\mathit{sold}\colon ?\mathrm{Price}.\mathbf{end},\ \mathit{notSold}\colon \mathbf{end}\}\}$$

THE TYPE CONSTRUCTORS

!Item	Send a value
?Price	Receive a value
⊕{*selling*: ...}	Select an option
&{*sold*: ..., *notSold*: ...}	Offer a set of options
end	Terminate

THE AUCTIONEER INTERACTING WITH A SELLER

THE AUCTIONEER INTERACTING WITH A SELLER

&{*selling*:

THE AUCTIONEER INTERACTING WITH A SELLER

Offer option *selling* to sellers

&{*selling*:

THE AUCTIONEER INTERACTING WITH A SELLER

&{*selling*:

THE AUCTIONEER INTERACTING WITH A SELLER

&{*selling*: ?Item.

THE AUCTIONEER INTERACTING WITH A SELLER

Receive the item to be sold

&{*selling*: ?Item.

THE AUCTIONEER INTERACTING WITH A SELLER

&{*selling*: ?Item.

THE AUCTIONEER INTERACTING WITH A SELLER

&{*selling*: ?Item.?Price.

THE AUCTIONEER INTERACTING WITH A SELLER

Receive its price

&{*selling*: ?Item.?Price.

THE AUCTIONEER INTERACTING WITH A SELLER

&{*selling*: ?Item.?Price.

THE AUCTIONEER INTERACTING WITH A SELLER

&{*selling*: ?Item.?Price.
⊕{*sold*:

THE AUCTIONEER INTERACTING WITH A SELLER

&{*selling*: ?Item.?Price.
⊕{*sold*:

Inform that the item was sold...

THE AUCTIONEER INTERACTING WITH A SELLER

&{*selling*: ?Item.?Price.
⊕{*sold*:

THE AUCTIONEER INTERACTING WITH A SELLER

&{*selling*: ?Item.?Price.
⊕{*sold*: !Price

THE AUCTIONEER INTERACTING WITH A SELLER

$\&\{selling\colon ?\mathrm{Item}.?\mathrm{Price}.$
$\quad\oplus\{sold\colon !\mathrm{Price}$

Send the selling price

THE AUCTIONEER INTERACTING WITH A SELLER

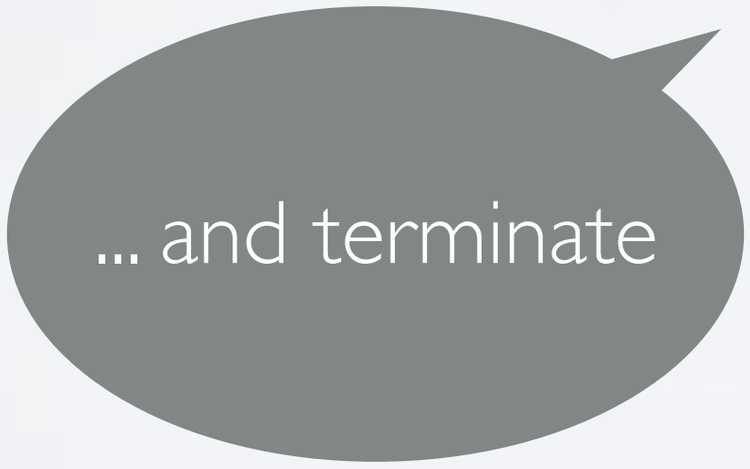
&{*selling*: ?Item.?Price.
⊕{*sold*: !Price

THE AUCTIONEER INTERACTING WITH A SELLER

&{*selling*: ?Item.?Price.
⊕{*sold*: !Price.**end**,

THE AUCTIONEER INTERACTING WITH A SELLER

&{*selling*: ?Item.?Price.
⊕{*sold*: !Price.**end**,



THE AUCTIONEER INTERACTING WITH A SELLER

&{*selling*: ?Item.?Price.
⊕{*sold*: !Price.**end**,

THE AUCTIONEER INTERACTING WITH A SELLER

&{*selling*: ?Item.?Price.
⊕{*sold*: !Price.**end**,
notSold: **end**}}

THE AUCTIONEER INTERACTING WITH A SELLER

$$\&\{\mathit{selling}\colon ?\mathrm{Item}.?\mathrm{Price}.\ \oplus\{\mathit{sold}\colon !\mathrm{Price}.\mathbf{end},\ \mathit{notSold}\colon \mathbf{end}\}\}$$

Or inform the item was not sold and terminate

THE AUCTIONEER INTERACTING WITH A SELLER

&{*selling*: ?Item.?Price.
⊕{*sold*: !Price.**end**,
notSold: **end**}}

SELLING AND BUYING...

- ... are complementary activities
- And so are the types that govern them:
- We call them dual

SELLING AND BUYING...

- ... are complementary activities
- And so are the types that govern them:

$$\oplus\{\mathit{selling}\colon\ !\mathrm{Item}.!\mathrm{Price}.\&\{\mathit{sold}\colon\ ?\mathrm{Price}.\mathbf{end},\ \mathit{notSold}\colon\ \mathbf{end}\}\}$$

- We call them dual

SELLING AND BUYING...

- ... are complementary activities
- And so are the types that govern them:

The protocol for the seller

⊕{*selling*: !Item.!Price.&{*sold*: ?Price.**end**, *notSold*: **end**}}

- We call them dual

SELLING AND BUYING...

- ... are complementary activities
- And so are the types that govern them:

The protocol for the seller

⊕{*selling*: !Item.!Price.&{*sold*: ?Price.**end**, *notSold*: **end**}}

&{*selling*: ?Item.?Price.⊕{*sold*: !Price.**end**, *notSold*: **end**}}

- We call them dual

SELLING AND BUYING...

- ... are complementary activities
- And so are the types that govern them:

The protocol for the seller

⊕{*selling*: !Item.!Price.&{*sold*: ?Price.**end**, *notSold*: **end**}}

&{*selling*: ?Item.?Price.⊕{*sold*: !Price.**end**, *notSold*: **end**}}

- We call them dual

The protocol for the auctioneer

THE BIDDERS PROTOCOL

THE BIDDERS PROTOCOL

⊕{*register*:

THE BIDDERS PROTOCOL

Select option *register* on the auctioneer

⊕{*register*:

THE BIDDERS PROTOCOL

⊕{*register*:

THE BIDDERS PROTOCOL

$\oplus\{register\colon !\mathrm{Name}.$

THE BIDDERS PROTOCOL

Send buyer's name

$\oplus\{register\colon !\mathrm{Name}.$

THE BIDDERS PROTOCOL

$\oplus\{register\colon !\mathrm{Name}.$

THE BIDDERS PROTOCOL

⊕{*register*: !Name. ?Item.

THE BIDDERS PROTOCOL

Receive an item on sale

$\oplus\{\mathit{register}: !\mathrm{Name}.\ ?\mathrm{Item}.$

THE BIDDERS PROTOCOL

⊕{*register*: !Name. ?Item.

THE BIDDERS PROTOCOL

$\oplus\{$*register*: !Name. ?Item.?Price.

THE BIDDERS PROTOCOL

Receive its price

$\oplus\{$*register*: !Name. ?Item.?Price.

THE BIDDERS PROTOCOL

$\oplus\{$*register*: !Name. ?Item.?Price.

THE BIDDERS PROTOCOL

⊕{*register*: !Name. ?Item.?Price.

⊕{*buy*:

THE BIDDERS PROTOCOL

$\oplus\{$*register*: !Name. ?Item.?Price.

$\oplus\{$*buy*:

Decided to buy...

THE BIDDERS PROTOCOL

⊕{*register*: !Name. ?Item.?Price.

⊕{*buy*:

THE BIDDERS PROTOCOL

$\oplus\{$*register*: !Name. ?Item.?Price.

$\oplus\{$*buy*: **end**,

THE BIDDERS PROTOCOL

$\oplus\{$*register*: !Name. ?Item.?Price.

$\oplus\{$*buy*: **end**,

... and terminate

THE BIDDERS PROTOCOL

$\oplus\{$*register*: !Name. ?Item.?Price.

$\oplus\{$*buy*: **end**,

THE BIDDERS PROTOCOL

$\oplus\{$*register*: !Name. ?Item.?Price.

$\oplus\{$*buy*: **end**,

notInterested: **end**$\}\}$

THE BIDDERS PROTOCOL

$\oplus\{$*register*: !Name. ?Item.?Price.

$\oplus\{$*buy*: **end**,

notInterested: **end**$\}\}$

Decided not to buy; terminate

THE BIDDERS PROTOCOL

$\oplus\{$*register*: !Name. ?Item.?Price.

$\oplus\{$*buy*: **end**,

notInterested: **end**$\}\}$

THE AUCTIONEER WHILE INTERACTING WITH BIDDERS

$$\&\{\mathit{register}\colon ?\mathrm{Name}.!\mathrm{Item}.!\mathrm{Price}.\ \&\{\mathit{buy}\colon \mathbf{end},\ \mathit{notInterested}\colon \mathbf{end}\}\}$$

- Recall the type when interacting with sellers:

$$\&\{\mathit{selling}\colon ?\mathrm{Item}.?\mathrm{Price}.\ \oplus\{\mathit{sold}\colon !\mathrm{Price}.\mathbf{end},\ \mathit{notSold}\colon \mathbf{end}\}\}$$

- Two types? one for interacting with sellers, the other for bidders?

THE AUCTIONEER ITSELF

$$\&\{\mathit{register}\colon ?\mathrm{Name}.\ !\mathrm{Item}.\ !\mathrm{Price}.\ \&\{\mathit{buy}\colon \mathbf{end},\ \mathit{notInterested}\colon \mathbf{end}\},\ \mathit{selling}\colon ?\mathrm{Item}.?\mathrm{Price}.\ \oplus\{\mathit{sold}\colon !\mathrm{Price}.\ \mathbf{end},\ \mathit{notSold}\colon \mathbf{end}\}\}$$

- The particular types used to interact with sellers and with bidders are subtypes of this type
- Advantages:
  - Bidders do not need to know the protocol for sellers
  - The code for bidders may be developed before the introduction of (online) sellers in the auction system

COMPATIBILITY

- The bidder-auctioneer compatibility rest assured. The actual type for the auctioneer

$$\&\{\mathit{register}\colon ?\mathrm{Name}.\ !\mathrm{Item}.\ !\mathrm{Price}.\ \&\{\mathit{buy}\colon \mathbf{end},\ \mathit{notInterested}\colon \mathbf{end}\},\ \mathit{selling}\colon \ldots\}$$

- is a supertype of

$$\&\{\mathit{register}\colon ?\mathrm{Name}.\ !\mathrm{Item}.\ !\mathrm{Price}.\ \&\{\mathit{buy}\colon \mathbf{end},\ \mathit{notInterested}\colon \mathbf{end}\}\}$$

- which is dual of that for the bidder

$$\oplus\{\mathit{register}\colon !\mathrm{Name}.\ ?\mathrm{Item}.\ ?\mathrm{Price}.\ \oplus\{\mathit{buy}\colon \mathbf{end},\ \mathit{notInterested}\colon \mathbf{end}\}\}$$

SYSTEM EVOLUTION

- By far the most common sellers' complaint is the inability of lowering the initial price after an unsuccessful auction
- The new auctioneer now provides a 3rd choice

$$\&\{\mathit{selling}\text{: ?Item.?Price.Selling, } \mathit{register}\text{: ...}\}$$

$$\text{Selling} = \oplus\{\mathit{sold}\text{: !Price. } \mathbf{end}\text{,}$$
$$\mathit{notSold}\text{: } \mathbf{end}\text{,}$$
$$\mathit{lowerYourPrice}\text{: } \&\{\mathit{ok}\text{: ?Price. Selling,}$$
$$\mathit{noWay}\text{: } \mathbf{end}\}\}$$

"We are very excited about your item; would you consider lowering the price?"

COMPATIBILITY ASSURED?

- The old seller still works, it just does not use the new functionality
- The new type is far more complex than the original: additional recursion and one more ⊕ choice. Expanding recursion we see that all there remains is one more choice

&{*selling*: ?Item. ?Price.
⊕{*sold*: !Price. **end**, *notSold*: **end**,
lowerYourPrice&{*ok*: ?Price. Selling,
noWay: **end**}}}

SUBTYPING

	Subtype	Supertype	Variancy
Branch &	Less options offered	More options offered	Covariant
Selection ⊕	More options taken	Less options taken	Contravariant
Input ?	Input value is subtype	Input value is supertype	Covariant
Output !	Output value is supertype	Output value is supertype	Contravariant

- In all cases continuation are covariant
- Recursion "unfolded away" _ co-inductive definition

SESSIONS

- Protocols such as the seller-auctioneer-bidder run between exactly two partners at a time:
  - seller-auctioneer, or
  - auctioneer-bidder
- Each such run is called a session

CHANNELS

- An auctioneer must be able to conduct multiple sessions in parallel, with different sellers, with different bidders
- And must not mix the sessions. E.g.,
  - Announcing *sold* to bidder A
  - Sending the corresponding **Price** to bidder B
- Each session is conducted on a different bi-directional communication medium called channel

ESTABLISHING SESSIONS

- How are sessions created?
- On channels known to all participants potentially interested on online auctions, e.g., distributed on the www
- We could distinguish
  - linear channels - known by one partner
  - shared channels - known by any number of partners...

CLASSIFYING OPERATIONS

- ...but we prefer to work with a single kind of channel and distinguish **lin**ear from **un**restricted (shared) operations

- This gives us greater flexibility and a simplified theory

- All the operations we have seen so far are linear

lin⊕{*selling*: lin!Item. lin!Price.
lin&{*sold*: lin?Price. un **end**,
notSold: un **end**}}

The interesting end is unrestricted

BACK TO SESSION ESTABLISHMENT

- The common knowledge between the three kinds of partners is a shared channel, used to establish linear sessions

- Recall the type of the auctioneer's session

$$\mathrm{T} = \mathrm{lin}\&\{\mathit{register}\colon \ldots, \mathit{selling}\colon \ldots\}$$

- The type of the shared channel is

BACK TO SESSION ESTABLISHMENT

- The common knowledge between the three kinds of partners is a shared channel, used to establish linear sessions

- Recall the type of the auctioneer's session

$$\mathrm{T} = \mathrm{lin}\&\{\mathit{register}\colon \ldots, \mathit{selling}\colon \ldots\}$$

- The type of the shared channel is

$$\mathrm{S} = \mathrm{un}?\mathrm{T}.\mathrm{S}$$

BACK TO SESSION ESTABLISHMENT

- The common knowledge between the three kinds of partners is a shared channel, used to establish linear sessions

- Recall the type of the auctioneer's session

$$\mathrm{T} = \mathrm{lin}\&\{\mathit{register}\colon \ldots, \mathit{selling}\colon \ldots\}$$

- The type of the shared channel is

$$\mathrm{S} = \mathrm{un}?\mathrm{T}.\mathrm{S}$$

Establish a session

BACK TO SESSION ESTABLISHMENT

- The common knowledge between the three kinds of partners is a shared channel, used to establish linear sessions

- Recall the type of the auctioneer's session

$$\mathrm{T} = \mathrm{lin}\&\{\mathit{register}\colon \ldots, \mathit{selling}\colon \ldots\}$$

- The type of the shared channel is

$$\mathrm{S} = \mathrm{un}?\mathrm{T}.\mathrm{S}$$

BACK TO SESSION ESTABLISHMENT

- The common knowledge between the three kinds of partners is a shared channel, used to establish linear sessions

- Recall the type of the auctioneer's session

$$\mathrm{T} = \mathrm{lin}\&\{\mathit{register}\colon \ldots, \mathit{selling}\colon \ldots\}$$

- The type of the shared channel is

$$\mathrm{S} = \mathrm{un}?\mathrm{T}.\mathrm{S}$$

Establish more sessions

BACK TO SESSION ESTABLISHMENT

- The common knowledge between the three kinds of partners is a shared channel, used to establish linear sessions

- Recall the type of the auctioneer's session

$$\mathrm{T} = \mathrm{lin}\&\{\mathit{register}\colon \ldots, \mathit{selling}\colon \ldots\}$$

- The type of the shared channel is

$$\mathrm{S} = \mathrm{un}?\mathrm{T}.\mathrm{S}$$

2.
PROGRAMMING

WHICH PROGRAMMING LANGUAGE?

- In which language shall we program the protocol?
  - Functional?
  - Imperative?
  - Object-oriented?
- You'll find all flavours in the literature
- It must incorporate the notion of channels; we shall use a pi-calculus

PROGRAMMING THE SELLER

c ◃ *selling*.
c ! “psp”.
c ! 100.
c ▹{
  sold ⇒ c?(x).print!(“made “ ^ x ^ “euros!”)
  notSold ⇒ print!(“next time I’ll ask 99.9!”)
}

PROGRAMMING THE SELLER

c is the name of a channel

c ◃ *selling*.

c ! “psp”.

c ! 100.

c ▹{

  sold ⇒ c?(x).print!(“made “ ^ x ^ “euros!”)

  notSold ⇒ print!(“next time I’ll ask 99.9!”)

}

PROGRAMMING THE SELLER

c ◃ *selling*.
c ! “psp”.
c ! 100.
c ▹{
  sold ⇒ c?(x).print!(“made “ ^ x ^ “euros!”)
  notSold ⇒ print!(“next time I’ll ask 99.9!”)
}

PROGRAMMING THE SELLER

Select option *selling* on the auctioneer

```
c ◁ selling.
c ! “psp”.
c ! 100.
c ▷{
  sold ⇒ c?(x).print!(“made “ ^ x ^ “euros!”)
  notSold ⇒ print!(“next time I’ll ask 99.9!”)
}
```

PROGRAMMING THE SELLER

c ◃ *selling*.
c ! “psp”.
c ! 100.
c ▹{
  sold ⇒ c?(x).print!(“made “ ^ x ^ “euros!”)
  notSold ⇒ print!(“next time I’ll ask 99.9!”)
}

PROGRAMMING THE SELLER

Send the item to be sold

```
c ◁ selling.
c ! “psp”.
c ! 100.
c ▷{
  sold ⇒ c?(x).print!(“made “ ^ x ^ “euros!”)
  notSold ⇒ print!(“next time I’ll ask 99.9!”)
}
```

PROGRAMMING THE SELLER

c ◃ *selling*.
c ! “psp”.
c ! 100.
c ▹{
  sold ⇒ c?(x).print!(“made “ ^ x ^ “euros!”)
  notSold ⇒ print!(“next time I’ll ask 99.9!”)
}

PROGRAMMING THE SELLER

c ◁ *selli*

c ! “psp”

c ! 100.

Send its price

c ▷{

sold ⇒ c?(x).print!(“made “ ^ x ^ “euros!”)

notSold ⇒ print!(“next time I’ll ask 99.9!”)

}

PROGRAMMING THE SELLER

c ◃ *selling*.
c ! “psp”.
c ! 100.
c ▹{
  sold ⇒ c?(x).print!(“made “ ^ x ^ “euros!”)
  notSold ⇒ print!(“next time I’ll ask 99.9!”)
}

PROGRAMMING THE SELLER

c ◁ s

c !

c !

Wait for an option

c ▷{

sold ⇒ c?(x).print!(“made “ ^ x ^ “euros!”)

notSold ⇒ print!(“next time I’ll ask 99.9!”)

}

PROGRAMMING THE SELLER

c ◃ *selling*.
c ! “psp”.
c ! 100.
c ▹{
  sold ⇒ c?(x).print!(“made “ ^ x ^ “euros!”)
  notSold ⇒ print!(“next time I’ll ask 99.9!”)
}

PROGRAMMING THE SELLER

```
c ◃ selling.
c ! "ps
c ! 10
c ▹{
  sold ⇒ c?(x).print!("made " ^ x ^ "euros!")
  notSold ⇒ print!("next time I'll ask 99.9!")
}
```

If the item was sold...

PROGRAMMING THE SELLER

c ◃ *selling*.
c ! “psp”.
c ! 100.
c ▹{
  sold ⇒ c?(x).print!(“made “ ^ x ^ “euros!”)
  notSold ⇒ print!(“next time I’ll ask 99.9!”)
}

PROGRAMMING THE SELLER

c ◃ *selling*.
c ! “psp”.
c ! 100.
c ▹{
  sold ⇒ c?(x).print!(“made “ ^ x ^ “euros!”)
  notSold ⇒ print!(“next time I’ll ask 99.9!”)
}

Read the price

PROGRAMMING THE SELLER

c ◃ *selling*.
c ! “psp”.
c ! 100.
c ▹{
  sold ⇒ c?(x).print!(“made “ ^ x ^ “euros!”)
  notSold ⇒ print!(“next time I’ll ask 99.9!”)
}

PROGRAMMING THE SELLER

```
c ◃ selling.
c ! "psp".
c ! 100.
c ▹{
  sold ⇒ c?(x).print!("made " ^ x ^ "euros!")
  notSold ⇒ print!("next time I'll ask 99.9!")
}
```

If not sold

PROGRAMMING THE SELLER

c ◃ *selling*.
c ! “psp”.
c ! 100.
c ▹{
  sold ⇒ c?(x).print!(“made “ ^ x ^ “euros!”)
  notSold ⇒ print!(“next time I’ll ask 99.9!”)
}

CHANNEL OPERATIONS

c ! “psp”	Send value
c?(x)	Receive value
c ◁ *selling*	Select option
c ▷{*sold*:..., *notSold*:...}	Branch on option

CHANNEL OPERATIONS AND TYPES

```
c ◁ selling.
c ! “psp”.
c ! 100.
c ▷{
   sold ⇒ c ? (x)...
   notSold ⇒ ...}
```

- Collect all operations on channel **c**
- **un** qualifier omitted on type **end**; **lin** omitted on all other type constructors

CHANNEL OPERATIONS AND TYPES

```
c ◁ selling.
c ! “psp”.
c ! 100.
c ▷{
  sold ⇒ c ? (x)...
  notSold ⇒ ...}
```

```
⊕{selling:
  !Item.
  !Price.
  &{
    sold: ?Price. end,
    notSold: end}}
```

- Collect all operations on channel **c**
- **un** qualifier omitted on type **end**; **lin** omitted on all other type constructors

PROGRAMMING THE BIDDER

c ◁ *register.*
c ! “Vasco”.
c ? (item).
c ? (price).
if (item = “psp” **and** price < 100)
then c ◁ *buy*
else c ◁ *notInterested*

THE CODE AND THE TYPE FOR THE BIDDER

```
c ◁ register.
c ! “Vasco”.
c ? (item).
c ? (price).
if (...)
then c ◁ buy
else c ◁ notInterested
```

- Collect all operations on channel **c**
- Qualifiers omitted on all type constructors as before

THE CODE AND THE TYPE FOR THE BIDDER

```
c ◁ register.
c ! “Vasco”.
c ? (item).
c ? (price).
if (...)
then c ◁ buy
else c ◁ notInterested
```

```
⊕{register:
    !Name.
    ?Item.
    ?Price.

    ⊕{buy: end,
      notInterested: end}}
```

- Collect all operations on channel **c**
- Qualifiers omitted on all type constructors as before

THE AUCTIONEER

- The most sophisticated piece of code

```
c ▷ {
  selling ⇒ -- handle sellers' requests
  register ⇒ -- handle bidders' requests
}
```

- More later...

BOOTSTRAPPING

- How do sellers and bidders start sessions?
- By requesting such a session on **a**, the auctioneer's public, shared, name:
- The auctioneer's public name is a shared channel of type

BOOTSTRAPPING

- How do sellers and bidders start sessions?

- By requesting such a session on **a**, the auctioneer's public, shared, name:

$$\text{Seller} = \text{a?(c).}\ \text{c} \triangleleft \mathit{selling}.\text{c}\ !\ \text{“psp”}\ldots$$

- The auctioneer's public name is a shared channel of type

BOOTSTRAPPING

- How do sellers and bidders start sessions?

- By requesting such a session on **a**, the auctioneer's public, shared, name:

$$\text{Seller} = a?(c).\ c \triangleleft \mathit{selling}.c\,!\,\text{“psp”}\ldots$$

$$\text{Bidder} = a?(c).\ c \triangleleft \mathit{register}.\ c\,!\,\text{“Vasco”}\ldots$$

- The auctioneer's public name is a shared channel of type

BOOTSTRAPPING

- How do sellers and bidders start sessions?

- By requesting such a session on **a**, the auctioneer's public, shared, name:

$$\text{Seller} = a?(c).\ c \triangleleft \mathit{selling}.c\,!\,\text{“psp”}...$$

$$\text{Bidder} = a?(c).\ c \triangleleft \mathit{register}.\ c\,!\,\text{“Vasco”}...$$

- The auctioneer's public name is a shared channel of type

$$S = \mathrm{un}?T.S$$

BOOTSTRAPPING

- How do auctioneers start sessions?
- By creating a fresh channel and sending it to clients
- We shall distinguish the two ends of a channel

$$
\begin{aligned}
\text{Auctioneer} = (\boldsymbol{\nu} cc')(\\
&a!c' \mid \\
&c \triangleright \{\mathit{selling} \Rightarrow \ldots\ \mathit{register} \Rightarrow \ldots\} \mid \\
&\text{Auctioneer})
\end{aligned}
$$

BOOTSTRAPPING

- How do auctioneers start sessions?
- By creating a fresh channel and sending it to clients
- We shall distinguish the two e[illegible]l

$$\begin{aligned}\text{Auctioneer} = (\boldsymbol{\nu}\, cc')(\;& \\ & a!c' \mid \\ & c \triangleright \{\mathit{selling} \Rightarrow \ldots\ \mathit{register} \Rightarrow \ldots\} \mid \\ & \text{Auctioneer})\end{aligned}$$

Create the two ends of a channel

BOOTSTRAPPING

- How do auctioneers start sessions?
- By creating a fresh channel and sending it to clients
- We shall distinguish the two ends of a channel

$$
\begin{aligned}
\text{Auctioneer} = (\boldsymbol{\nu} cc')(\\
&a!c' \mid \\
&c \triangleright \{\mathit{selling} \Rightarrow \ldots\ \mathit{register} \Rightarrow \ldots\} \mid \\
&\text{Auctioneer})
\end{aligned}
$$

BOOTSTRAPPING

- How do auctioneers start sessions?
- By creating a fresh channel and sending it to clients
- We shall distinguish the two ends of a channel

$$\begin{aligned}\text{Auctioneer} = (\nu cc')(&\\ &a!c' \mid\\ &c \triangleright \{\mathit{selling} \Rightarrow \ldots\ \mathit{register} \Rightarrow \ldots\} \mid\\ &\text{Auctioneer})\end{aligned}$$

Send one end to the client

BOOTSTRAPPING

- How do auctioneers start sessions?
- By creating a fresh channel and sending it to clients
- We shall distinguish the two ends of a channel

$$
\begin{aligned}
\text{Auctioneer} = (\boldsymbol{\nu} cc')(\\
&a!c' \mid \\
&c \triangleright \{\mathit{selling} \Rightarrow \ldots\ \mathit{register} \Rightarrow \ldots\} \mid \\
&\text{Auctioneer})
\end{aligned}
$$

BOOTSTRAPPING

- How do auctioneers start sessions?
- By creating a fresh channel and sending it to clients
- We shall distinguish the two ends of a channel

$$\begin{aligned}\mathrm{Auctioneer} = (\boldsymbol{\nu} cc')(\\ & a!c' \mid \\ & c \triangleright \{\mathit{selling} \Rightarrow \ldots\ \mathit{register} \Rightarrow \ldots\} \mid \\ & \mathrm{Auctioneer})\end{aligned}$$

Interact on the other end

BOOTSTRAPPING

- How do auctioneers start sessions?
- By creating a fresh channel and sending it to clients
- We shall distinguish the two ends of a channel

$$
\begin{aligned}
\text{Auctioneer} = (\boldsymbol{\nu} cc')(\\
&a!c' \mid \\
&c \triangleright \{\mathit{selling} \Rightarrow \ldots\ \mathit{register} \Rightarrow \ldots\} \mid \\
&\text{Auctioneer})
\end{aligned}
$$

MORE PROCESS CONSTRUCTORS

$P \mid Q$	Parallel composition
$(\nu c c')$	Channel creation
if v **then** P **else** Q	Conditional

REFINING THE AUCTIONEER

- Concentrate on the selling option; some pseudo-code first

```
c?(item). c?(price). put(item, price)
if sold(item)
then c ◁ sold. c!price(item)
else c ◁ notSold
```

- Assume a shared repository with operations put, sold, price

THE SHARED AUCTION DATA REPOSITORY

- If shared, it must be accessed by a protocol
- Operation put(item, price) becomes

$$\mathrm{r?(d).\ d} \triangleleft \mathit{put}.\ \mathrm{d!item.\ d!price}$$

- where **r** is the shared name for the repository

THE SHARED AUCTION DATA REPOSITORY

- Operations sold(item)/price(item) must be dealt together

- Sessions **c** and **e** are now mixed...
- ... but the types remain apart. The type of **e** is

THE SHARED AUCTION DATA REPOSITORY

- Operations sold(item)/price(item) must be dealt together

$$\mathrm{r?(e).\ e} \triangleleft \mathit{wasItSold}.\ \mathrm{e!item.}$$
$$\mathrm{e} \triangleright \{\mathit{sold} \Rightarrow \mathrm{c} \triangleleft \mathit{sold}.\ \mathrm{e?(price).\ c!price},$$
$$\mathit{notSold} \Rightarrow \mathrm{c} \triangleleft \mathit{notSold}\}$$

- Sessions **c** and **e** are now mixed...
- ... but the types remain apart. The type of **e** is

THE SHARED AUCTION DATA REPOSITORY

- Operations sold(item)/price(item) must be dealt together

$$\mathrm{r?(e).\ e} \triangleleft \mathit{wasItSold}.\ \mathrm{e!item.}$$
$$\mathrm{e} \triangleright \{\mathit{sold} \Rightarrow \mathrm{c} \triangleleft \mathit{sold}.\ \mathrm{e?(price).\ c!price},$$
$$\mathit{notSold} \Rightarrow \mathrm{c} \triangleleft \mathit{notSold}\}$$

- Sessions **c** and **e** are now mixed...
- ... but the types remain apart. The type of **e** is

$$\oplus\{\mathit{wasItSold}\colon !\mathrm{Item}.\ \&\{\mathit{sold}\colon ?\mathrm{Price}.\mathbf{end},\ \mathit{notSold}\colon \mathbf{end}\}\}$$

SESSION DELEGATION

- Noticed the copy-cat?

$$\mathrm{e} \triangleright \{sold \Rightarrow \mathrm{c} \triangleleft sold.\ \mathrm{e?(price).\ c!price}$$
$$notSold \Rightarrow \mathrm{c} \triangleleft notSold\}$$

- Why not trust the seller's channel to the repository? The repository takes care of replying directly to the client

SESSION DELEGATION

- Noticed the copy-cat?

$$\mathrm{e} \triangleright \{sold \Rightarrow \mathrm{c} \triangleleft sold.\ \mathrm{e?(price).\ c!price}$$
$$notSold \Rightarrow \mathrm{c} \triangleleft notSold\}$$

- Why not trust the seller's channel to the repository? The repository takes care of replying directly to the client

$$\mathrm{r?(e).\ e} \triangleleft wasItSold.\ \mathrm{e?(item).\ e!c}$$

SESSION DELEGATION

- Noticed the copy-cat?

$$\mathrm{e} \triangleright \{\mathit{sold} \Rightarrow \mathrm{c} \triangleleft \mathit{sold}.\ \mathrm{e?(price)}.\ \mathrm{c!price}$$
$$\mathit{notSold} \Rightarrow \mathrm{c} \triangleleft \mathit{notSold}\}$$

- Why not trust the seller's channel to the repository? The repository takes care of replying directly to the client

$$\mathrm{r?(e)}.\ \mathrm{e} \triangleleft \mathit{wasItSold}.\ \mathrm{e?(item)}.\ \mathrm{e!c}$$

Sending a
channel on a channel

A TYPE FOR THE REPOSITORY

- The type for the shared channel...

- ... and that for the session it establishes

- The seller is not aware of the delegation; it needs not change its type or its code

A TYPE FOR THE REPOSITORY

- The type for the shared channel...

$$S = \mathbf{un}\ ?T.S$$

- ... and that for the session it establishes

- The seller is not aware of the delegation; it needs not change its type or its code

A TYPE FOR THE REPOSITORY

- The type for the shared channel...

$$S = \mathbf{un}\ ?T.S$$

- ... and that for the session it establishes

$$T = \oplus\{put\colon !\mathrm{Item}.\ !\mathrm{Price}.\ \mathbf{end},\ wasItSold\colon !\mathrm{Item}.\ !U.\ \mathbf{end}\}$$

$$U = \&\{sold\colon ?\mathrm{Price}.\ \mathbf{end},\ notSold\colon \mathbf{end}\}$$

- The seller is not aware of the delegation; it needs not change its type or its code

A TYPE FOR THE REPOSITORY

- The type for the shared channel...

$$S = \mathbf{un}\ ?T.S$$

- ... and that for the session it establishes

$$T = \oplus\{put\text{: } !\mathrm{Item}.\ !\mathrm{Price}.\ \mathbf{end},\ wasItSold\text{: } !\mathrm{Item}.\ !U.\ \mathbf{end}\}$$

$$U = \&\{sold\text{: } ?\mathrm{Price}.\ \mathbf{end},\ notSold\text{: } \mathbf{end}\}$$

The type of the delegated session

- The seller is not aware of the delegation; it needs not change its type or its code

PART II
THE THEORY

THE PROGRAMME

- The language
- Its typing system and
- Its operational semantics
- What is an error?
- Main result: Typable processes do not reduce to errors

THE PROGRAMME

- The language
- Its typing system and
- Its operational semantics
- What is an error?
- Main result: Typable processes do not reduce to errors

Very much standard!

I.

THE LANGUAGE

CHANNEL CREATION

- Variables come in pairs, called co-variables
- Each represents one end of a communication channel
- Interacting threads do not share variables for communication; instead, each thread owns its variable
- This mechanism allows a precise control of resources via a linear type system

THE SYNTAX OF PROCESSES, TO START WITH

P ::=		Processes:
	$\overline{x}\,v.P$	output
	$x(x).P$	input
	$P \mid P$	parallel composition
	if v then P else P	conditional
	$\mathbf{0}$	inaction
	$(\nu x x)P$	scope restriction
v ::=		Values:
	x	variable
	true $\mid$ false	boolean values

2.
TYPING SYSTEM

LINEAR AND UNRESTRICTED TYPES

- **Lin** (linear) qualified types describe variables that occur in exactly one thread

- **Un** (unrestricted, shared) qualifier indicates a value that can occur in multiple threads

- Type **lin ! (lin bool). un end** represents a channel-end that can be used once to output a boolean value (that can be used once) and then behaves as shared channel on which no further operation is possible

THE SYNTAX OF TYPES

q ::=		Qualifiers:
	lin	linear
	un	unrestricted
p ::=		Pretypes:
	bool	booleans
	end	termination
	$?T.T$	receive
	$!T.T$	send
T ::=		Types:
	$q\ p$	qualified pretype
Γ ::=		Contexts:
	$\emptyset$	empty context
	$\Gamma, x : T$	assumption

EXAMPLE: VALID AND INVALID PROCESSES

- x is a variable of an arbitrarily qualified type
- a is a variable of an unrestricted type, and
- c a variable of a linear type

$$\overline{x}\,\mathsf{true}.x(y)$$

$$\overline{a}\,\mathsf{true} \mid \overline{a}\,\mathsf{true} \mid \overline{a}\,\mathsf{false}$$

$$\overline{c}\,\mathsf{true} \mid \overline{c}\,\mathsf{false}$$

Omit the trailing 0

DUALITY

No rule for bool

$$\overline{q\,?T.U} = q\,!T.\overline{U} \qquad \overline{q\,!T.U} = q\,?T.\overline{U} \qquad \overline{q\,\mathsf{end}} = q\,\mathsf{end}$$

x_1 and x_2 are two co-variables

$$\overline{x_1}\ \mathsf{true} \mid x_2(z)$$

$$\overline{x_1}\ \mathsf{true}.x_1(w) \mid x_2(z).\overline{x_2}\ \mathsf{false}$$

$$\overline{x_1}\ \mathsf{true} \mid \overline{x_2}\ \mathsf{false}$$

$$\overline{x_1}\ \mathsf{true}.x_1(w) \mid x_2(z).x_2(t)$$

CONTEXT SPLITTING

- An operation central to linear typing systems

$$\emptyset \cdot \emptyset = \emptyset$$

$$\frac{\Gamma = \Gamma_1 \cdot \Gamma_2 \qquad \mathsf{un}(T)}{\Gamma, x\colon T = (\Gamma_1, x\colon T) \cdot (\Gamma_2, x\colon T)}$$

$$\frac{\Gamma = \Gamma_1 \cdot \Gamma_2 \qquad \mathsf{lin}(T)}{\Gamma, x\colon T = (\Gamma_1, x\colon T) \cdot \Gamma_2}$$

$$\frac{\Gamma = \Gamma_1 \cdot \Gamma_2 \qquad \mathsf{lin}(T)}{\Gamma, x\colon T = \Gamma_1 \cdot (\Gamma_2, x\colon T)}$$

- When type checking processes with two sub-processes we pass the unrestricted part of the context to both processes, while splitting the linear part in two and passing a different part to each process

TYPE SYSTEM: INVARIANTS

- Linear channels occur in exactly one thread
- Co-variables have dual types

TYPING RULES FOR VALUES

$$\frac{\mathsf{un}(\Gamma)}{\Gamma \vdash \mathsf{false} : \mathsf{bool}} \qquad \frac{\mathsf{un}(\Gamma_1, \Gamma_2)}{\Gamma_1, x : T, \Gamma_2 \vdash x : T}$$

- The rules make sure that linear variables are not discarded without being used
- The base cases of the type system check that there is no linear variable in the context

TYPING RULES FOR PROCESSES (1/3)

$$\frac{\mathsf{un}(\Gamma)}{\Gamma \vdash \mathbf{0}} \qquad \frac{\Gamma_1 \vdash P \qquad \Gamma_2 \vdash Q}{\Gamma_1 \cdot \Gamma_2 \vdash P \mid Q}$$

- The base case checks that there is no linear variable in the context
- Parallel composition crucially takes advantage of context splitting

TYPING RULES FOR PROCESSES (2/3)

$$\frac{\Gamma_1 \vdash v\colon q\,\mathsf{bool} \qquad \Gamma_2 \vdash P \qquad \Gamma_2 \vdash Q}{\Gamma_1 \cdot \Gamma_2 \vdash \mathsf{if}\ v\ \mathsf{then}\ P\ \mathsf{else}\ Q} \qquad \frac{\Gamma, x\colon T, y\colon \overline{T} \vdash P}{\Gamma \vdash (\nu xy)P}$$

- No context splitting for the two branches in the conditional
- The rule for scope restriction captures the essence of co-variables: they must have dual types

TYPING RULES FOR PROCESSES (3/3)

$$\frac{\Gamma_1 \vdash x : q?T.U \qquad (\Gamma_2, y : T) \cdot x : U \vdash P}{\Gamma_1 \cdot \Gamma_2 \vdash x(y).P}$$

$$\frac{\Gamma_1 \vdash x : q!T.U \qquad \Gamma_2 \vdash v : T \qquad \Gamma_3 \cdot x : U \vdash P}{\Gamma_1 \cdot \Gamma_2 \cdot \Gamma_3 \vdash \overline{x}\, v.P}$$

- The rule for input splits the context into two parts: one to type check x, the other to type check continuation P
- If $x: q?T.U$ in $x(y).P$ then we use $y:T$ to type check P
- $x(y).P$ uses x at type $q?T.U$, whereas P may use the same variable this time at type U

TYPING RULES FOR PROCESSES (SUMMARY)

$$\frac{\mathsf{un}(\Gamma)}{\Gamma \vdash \mathbf{0}} \qquad \frac{\Gamma_1 \vdash P \qquad \Gamma_2 \vdash Q}{\Gamma_1 \cdot \Gamma_2 \vdash P \mid Q}$$

$$\frac{\Gamma_1 \vdash v\colon q\,\mathsf{bool} \qquad \Gamma_2 \vdash P \qquad \Gamma_2 \vdash Q}{\Gamma_1 \cdot \Gamma_2 \vdash \mathsf{if}\ v\ \mathsf{then}\ P\ \mathsf{else}\ Q} \qquad \frac{\Gamma, x_1\colon T, x_2\colon \overline{T} \vdash P}{\Gamma \vdash (\nu x_1 x_2)P}$$

$$\frac{\Gamma_1 \vdash x\colon q\,?T.U \qquad (\Gamma_2, y\colon T) \cdot x\colon U \vdash P}{\Gamma_1 \cdot \Gamma_2 \vdash x(y).P}$$

$$\frac{\Gamma_1 \vdash x\colon q\,!T.U \qquad \Gamma_2 \vdash v\colon T \qquad \Gamma_3 \cdot x\colon U \vdash P}{\Gamma_1 \cdot \Gamma_2 \cdot \Gamma_3 \vdash \overline{x}\,v.P}$$

A SIMPLE SESSION

Unrestricted environment

$$\dfrac{x\colon T' \vdash x\colon T' \quad x\colon \mathsf{end}, y\colon \mathsf{bool} \vdash \mathbf{0}}{\dfrac{x\colon T \vdash x\colon T \quad \emptyset \vdash \mathsf{true}\colon \mathsf{bool} \qquad x\colon T' = ?\mathsf{bool}.\mathsf{end} \vdash x(y).\mathbf{0}}{x\colon T = \mathsf{lin}!\mathsf{bool}.?\mathsf{bool}.\mathsf{end} \vdash \overline{x}\,\mathsf{true}.x(y).\mathbf{0}}}$$

Unrestricted qualifiers omitted

LINEAR VALUES ARE CONSUMED

Not unrestricted

$$\dfrac{\dfrac{}{x\colon T' \vdash x\colon T'} \quad \dfrac{}{x\colon \mathsf{end}, y\colon \mathsf{lin\ bool} \vdash \mathbf{0}}}{x\colon T' = ?(\mathsf{lin\ bool}).\mathsf{end} \vdash x(y).\mathbf{0}} \qquad \dfrac{}{x\colon T \vdash x\colon T} \quad \dfrac{}{\emptyset \vdash \mathsf{true}\colon \mathsf{bool}}}{x\colon T = \mathsf{lin!bool}.?(\mathsf{lin\ bool}).\mathsf{end} \vdash \overline{x}\,\mathsf{true}.x(y).\mathbf{0}}$$

Process not typable

A LANGUAGE OF LINEAR CHANNELS ONLY

Context splitting not defined

$$\frac{\Gamma \vdash x : !(\mathsf{bool}).\mathsf{end} \quad \Gamma \vdash \mathsf{true} : \mathsf{bool} \quad \Gamma \cdot (x : \mathsf{end}) \vdash \mathbf{0}}{\Gamma \vdash \overline{x}\,\mathsf{true}} \qquad \Gamma \vdash \overline{x}\,\mathsf{true}$$

$$\Gamma = x : !(\mathsf{bool}).\mathsf{end} \vdash \overline{x}\,\mathsf{true} \mid \overline{x}\,\mathsf{true}$$

Process not typable; to be fixed!

Parallel composition

3.

OPERATION SEMANTICS

STRUCTURAL CONGRUNECE

- Factor out a on processes allowing the syntactic rearrangement of these
- Contribute for a more concise presentation of the reduction relation

$$P \mid Q \equiv Q \mid P \qquad (P \mid Q) \mid R \equiv P \mid (Q \mid R) \qquad P \mid \mathbf{0} \equiv P$$

$$(\nu xy)P \mid Q \equiv (\nu xy)(P \mid Q) \qquad (\nu xy)\mathbf{0} \equiv \mathbf{0} \qquad (\nu wz)(\nu xy)P \equiv (\nu xy)(\nu wz)P$$

REDUCTION

$$(\nu xy)(\overline{x}\,v.P \mid y(z).Q \mid R) \;\rightarrow\; (\nu xy)(P \mid Q[v/z] \mid R)$$

$$\text{if true then } P \text{ else } Q \;\rightarrow\; P$$

$$\text{if false then } P \text{ else } Q \;\rightarrow\; Q$$

$$\frac{P \;\rightarrow\; Q}{(\nu xy)P \;\rightarrow\; (\nu xy)Q}$$

$$\frac{P \;\rightarrow\; Q}{P \mid R \;\rightarrow\; Q \mid R}$$

$$\frac{P \equiv P' \qquad P' \;\rightarrow\; Q' \qquad Q' \equiv Q}{P \;\rightarrow\; Q}$$

WHAT CAN GO WRONG?

- An obvious case: in a conditional process the value in the condition is neither true nor false

- More interesting:

$$\overline{a}\,\mathsf{true} \mid a(z)$$

$$(\nu x_1 x_2)(\overline{x_1}\,\mathsf{true} \mid \overline{x_2}\,\mathsf{true})$$

$$(\nu x_1 x_2)(x_1(z) \mid x_2(w))$$

- Notice: no mention of types or lin/un channel nature

WHAT CAN GO WRONG?

- An obvious case: in a conditional proce
  condition is neither true nor false

Different communication patterns on the same channel end

- More interesting:

$$\overline{a}\,\mathsf{true} \mid a(z)$$

$$(\nu x_1 x_2)(\overline{x_1}\,\mathsf{true} \mid \overline{x_2}\,\mathsf{true})$$

$$(\nu x_1 x_2)(x_1(z) \mid x_2(w))$$

- Notice: no mention of types or lin/un channel nature

WHAT CAN GO WRONG?

- An obvious case: in a conditional process the value in the condition is neither true nor false

- More interesting:

$$\overline{a}\,\mathsf{true} \mid a(z)$$

$$(\nu x_1 x_2)(\overline{x_1}\,\mathsf{true} \mid \overline{x_2}\,\mathsf{true})$$

$$(\nu x_1 x_2)(x_1(z) \mid x_2(w))$$

- Notice: no mention of types or lin/un channel nature

WHAT CAN GO WRONG?

- An obvious case: in a conditional process the value in the condition is neither true nor false

- Mor[illegible]ting:

Channels ends with incompatible communication patterns

$$\overline{a}\,\mathsf{true} \mid a(z)$$

$$(\nu x_1 x_2)(\overline{x_1}\,\mathsf{true} \mid \overline{x_2}\,\mathsf{true})$$

$$(\nu x_1 x_2)(x_1(z) \mid x_2(w))$$

- Notice: no mention of types or lin/un channel nature

WHAT CAN GO WRONG?

- An obvious case: in a conditional process the value in the condition is neither true nor false

- More interesting:

$$\overline{a}\,\mathsf{true} \mid a(z)$$

$$(\nu x_1 x_2)(\overline{x_1}\,\mathsf{true} \mid \overline{x_2}\,\mathsf{true})$$

$$(\nu x_1 x_2)(x_1(z) \mid x_2(w))$$

- Notice: no mention of types or lin/un channel nature

WELL TYPED PROGRAMS DO NOT GO WRONG

- Outline of the proof

1. Type preservation

$$\text{If } \Gamma \vdash P \text{ and } P \rightarrow Q \text{ then } \Gamma \vdash Q$$

2. Type safety

$$\text{If } \vdash P \text{ then } P \text{ is well formed}$$

3. Compose and done! (details in the book)

4.
RECURSIVE TYPES

RECURSIVE TYPES

- Context splitting not defined...

$$\frac{x : !\mathsf{bool}.T \vdash x : !\mathsf{bool}.T \qquad x : !\mathsf{bool}.T \vdash \mathsf{true} : \mathsf{bool} \qquad (x : !\mathsf{bool}.T) \cdot (x : \mathsf{end}) \vdash \mathbf{0}}{x : !\mathsf{bool}.T \vdash \overline{x}\,\mathsf{true}}$$

- ... unless there is type T such that

$$!\mathsf{bool}.T = T$$

- Use a finite notation for the solution of the equation

$$\mu a.!\mathsf{bool}.a$$

TYPES FOR SHARED CHANNELS

- Shared output channels are of types $\mu a.!T.a$ which we abbreviate to $*!T$

$$*!\mathsf{bool} = !\mathsf{bool}.*!\mathsf{bool}$$

$$\frac{\Gamma \vdash x : *!\mathsf{bool} \quad \Gamma \vdash \mathsf{true} : \mathsf{bool} \quad \Gamma \cdot (x : *!\mathsf{bool}) \vdash \mathbf{0}}{\Gamma = x : !\mathsf{bool}.*!\mathsf{bool} \vdash \overline{x}\,\mathsf{true}}$$

- Unfold/fold recursive types as needed: equi-recursive notion of types

EXAMPLE _ TUPLE PASSING ON SHARED CHANNELS

- If x is is shared then there is a risk of interference

$$\overline{x}\,\langle u, v\rangle.P = \overline{x}\,u.\overline{x}\,v.P$$

ok if x is linear

- Use the standard encoding...

$$\overline{x_1}\,\langle u, v\rangle.P = (\nu y_1 y_2)\overline{x_1}\,y_2.\overline{y_1}\,u.\overline{y_1}\,v.P$$

$$x_2(w, t).P = x_2(z).z(w).z(t).P$$

- ... which is typable

$$x_1 : \ *!(\mathsf{lin}?T.\mathsf{lin}?U)$$

$$x_2 : \ *?(\mathsf{lin}!T.\mathsf{lin}!U)$$

LINEAR CHANNELS THAT BECOME UNRESTRICTED

- Suppose that

$$x_1 : \quad \mathsf{lin!bool}.*?\mathsf{bool}$$

$$x_2 : \quad \mathsf{lin?bool}.*!\mathsf{bool}$$

- Then

$$\overline{x_1}\,\mathsf{true}.(x_1(y) \mid x_1(z)) \mid x_2(x).(\overline{x_2}\,\mathsf{true} \mid \overline{x_2}\,\mathsf{false} \mid \overline{x_2}\,\mathsf{true})$$

5.
REPLICATION

UNBOUNDED BEHAVIOR

- Up until now our language is strongly normalizing
- Providing for unbounded behavior
  - Recursion (as in the auctioneer example)
  - Replication
- Replication is conceptually simpler; we go for it

SYNTAX AND REDUCTION

- New syntax

$$P ::= \ldots \qquad \text{Processes:}$$

$$* x(x).P \qquad \text{replication}$$

- New reduction rule

$$(\nu xy)(\overline{x}\, v.P \mid *y(z).Q \mid R) \rightarrow (\nu xy)(P \mid Q[v/z] \mid *y(z).Q \mid R)$$

$$(\nu xy)(\overline{x}\, v.P \mid y(z).Q \mid R) \rightarrow (\nu xy)(P \mid Q[v/z] \mid R)$$

The replicated process survives reduction

TYPING

$$\frac{\Gamma \vdash P \qquad \mathsf{un}(\Gamma)}{\Gamma \vdash *P}$$

TYPING

P can be used multiple times

$$\frac{\Gamma \vdash P \qquad \mathsf{un}(\Gamma)}{\Gamma \vdash *P}$$

TYPING

$$\frac{\Gamma \vdash P \qquad \mathsf{un}(\Gamma)}{\Gamma \vdash *P}$$

"PROCEDURES"

- A procedure that sends a boolean value on a fixed channel c

$$p_2 : *?\mathsf{end}, c : \mathsf{lin}!\mathsf{bool} \not\vdash * p_2(z).\overline{c}\,\mathsf{true}$$

- c cannot be linear for the body of the procedure must be un

$$*p_2(z).\overline{c}\,\mathsf{true} \mid \overline{p_1}\,x \mid \overline{p_1}\,x \;\longrightarrow\longrightarrow\; *p_2(z).\overline{c}\,\mathsf{true} \mid \overline{c}\,\mathsf{true} \mid \overline{c}\,\mathsf{true}$$

- What if we pass c as parameter?

$$p_2 : ?(\mathsf{lin}!\mathsf{bool}) \vdash * p_2(z).\overline{z}\,\mathsf{true}$$

- The type system controls the linearity of the arguments

$$\ldots, p_2 : ?(\mathsf{lin}!\mathsf{bool}) \not\vdash * p_2(z).\overline{z}\,\mathsf{true} \mid \overline{p_1}\,c \mid \overline{p_1}\,c$$

- Linear values in procedures must be passed as parameters

6.
CHOICE

CHOICE _ SYNTAX

$$
\begin{array}{rll}
P & ::= & \ldots \qquad \text{Processes:} \\
 & & x \triangleleft l.P \qquad \text{selection} \\
 & & x \triangleright \{l_i \colon P_i\}_{i \in I} \qquad \text{branching} \\
p & ::= & \ldots \qquad \text{Pretypes:} \\
 & & \oplus \{l_i \colon T_i\}_{i \in I} \qquad \text{select} \\
 & & \& \{l_i \colon T_i\}_{i \in I} \qquad \text{branch}
\end{array}
$$

CHOICE _ REDUCTION

$$\frac{j \in I}{(\nu xy)(x \lhd l_j.P \mid y \rhd \{l_i : Q_i\}_{i \in I} \mid R) \;\rightarrow\; (\nu xy)(P \mid Q_j \mid R)}$$

CHOICE _TYPING

- Duality

$$\overline{q \oplus \{l_i : T_i\}_{i \in I}} = q \,\&\, \{l_i : \overline{T_i}\}_{i \in I} \qquad \overline{q \,\&\, \{l_i : T_i\}_{i \in I}} = q \oplus \{l_i : \overline{T_i}\}_{i \in I}$$

- Typing

$$\frac{\Gamma_2 \vdash x : q \oplus \{l_i : T_i\}_{i \in I} \qquad \Gamma_2 \cdot x : T_j \vdash P \qquad j \in I}{\Gamma_1 \cdot \Gamma_2 \vdash x \lhd l_j.P}$$

$$\frac{\Gamma_1 \vdash x : q \,\&\, \{l_i : T_i\}_{i \in I} \qquad \Gamma_2 \cdot x : T_i \vdash P_i \qquad \forall i \in I}{\Gamma_1 \cdot \Gamma_2 \vdash x \rhd \{l_i : P_i\}_{i \in I}}$$

7.

SUBTYPING

MOTIVATION

- Subtyping brings extra flexibility to our type system

- The insistence that arguments in output processes exactly match input parameters in corresponding receivers leads to the rejection of programs that will never go wrong when executed

- We have seen examples in the first part

INGREDIENTS

- We need
  - A notion of subtyping. $T <: U$ means any value of type T can be safely used in a context where U is expected
  - A rule to incorporate subtyping in the type system

$$\frac{\Gamma \vdash v : T \qquad T <: U}{\Gamma \vdash v : U}$$

FINITE SUBTYPING _ CHOICE

- The auctioneer can forget options in order to address sellers or bidders

$$\frac{I \subseteq J \qquad T_i <: U_i \qquad \forall i \in I}{\&\{l_i : T_i\}_{i \in I} <: \&\{l_j : U_j\}_{j \in J}}$$

- Conversely sellers may call more options on the auctioneer (so as to behave as bidders as well)

$$\frac{I \supseteq J \qquad T_j <: U_j \qquad \forall j \in J}{\oplus\{l_i : T_i\}_{i \in I} <: \oplus\{l_j : U_j\}_{j \in J}}$$

FINITE SUBTYPING _ I/O

- Input is co-variant; output is contra-variant

$$\frac{T' <: T \qquad U <: U'}{!T.U <: !T'.U'} \qquad \frac{T <: T' \qquad U <: U'}{?T.U <: ?T'.U'}$$

- In summary:
  - Input operations (?, &) are co-variant; output operations (!, ⊕) contra-variant
  - Continuations are always co-variant
- See book for subtyping recursive types

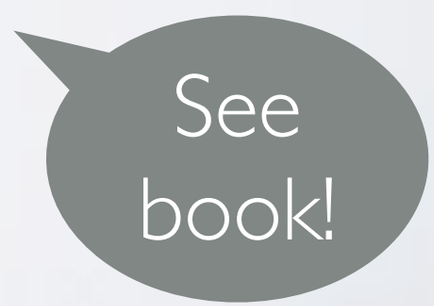
8.
CONCLUSION

ALGORITHMIC TYPE CHECKING

- The typing rules cannot be implemented directly for two main reasons. Difficulties

  - Implementing the non-deterministic splitting operation

  - Guessing the types for nu-bound variables



WE INTRODUCED...

- A (type) language to describe (the protocol part of) services
- A (programming) language to program protocols
- A decidable type checking system that makes sure that "well typed programs do not go wrong"

LOOKING FOR RESEARCH TOPICS?

- Programs can easily deadlock

$$\overline{x_1}\,\mathsf{true}.\overline{y_1}\,\mathsf{false} \mid y_2(x).x_2(w)$$

  - Identify conditions / refine the type system so as to ensure progress

- Context splitting for both parallel and sequential composition

$$\frac{\Gamma_1 \vdash x\colon q\,?T.U \qquad (\Gamma_2, y\colon T) \cdot x\colon U \vdash P}{\Gamma_1 \cdot \Gamma_2 \vdash x(y).P}$$

Parallel

Sequential