A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming

Alceste Scalas[1] **Ornela Dardha**[2] Raymond Hu[1] Nobuko Yoshida[1]

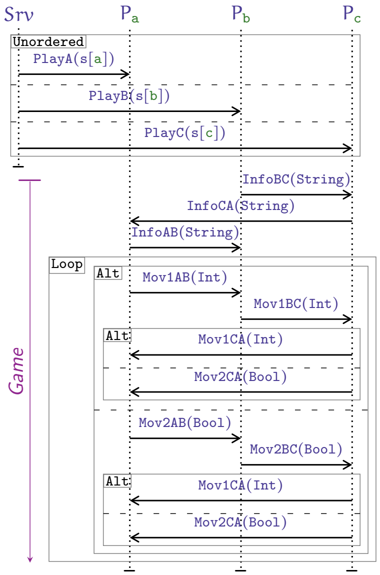
Imperial College London (1)

University of Glasgow (2)

Open Problems in Concurrency Theory — Vien, 27 June 2017

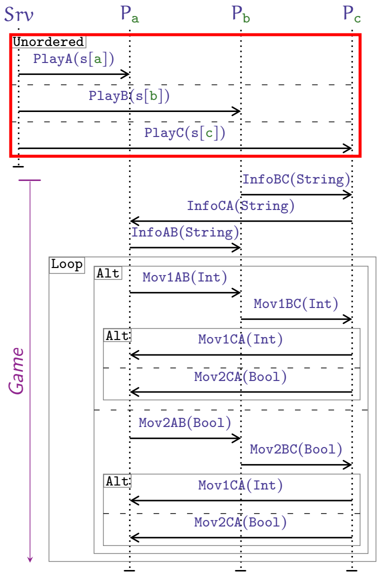
Supported by the UK EPSRC grant EP/K034413/1, *From Data Types to Session Types: A Basis for Concurrency and Distribution* (ABCD)

A Motivating Example: Peer-to-Peer Game



Clients P_a, P_b, P_c want to play a game as roles a, b, c via a **matchmaking server** *Srv*

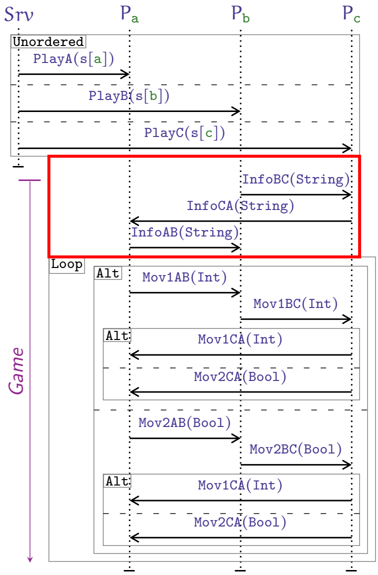
A Motivating Example: Peer-to-Peer Game



Clients P_a, P_b, P_c want to play a game as roles a, b, c via a **matchmaking server** $\mathcal{S}rv$

The server $\mathcal{S}rv$ sends some networking data to the clients, so they **"know each other"**

A Motivating Example: Peer-to-Peer Game

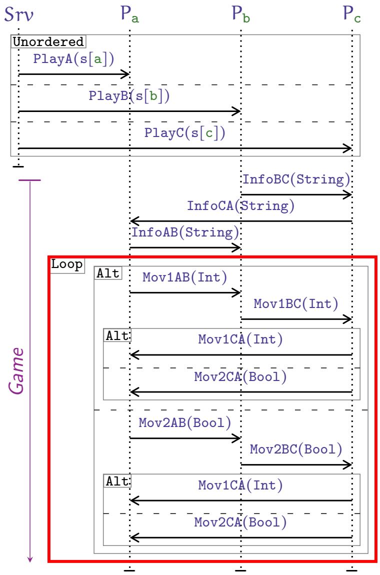


Clients P_a, P_b, P_c want to play a game as roles a, b, c via a **matchmaking server** *Srv*

The server *Srv* sends some networking data to the clients, so they **"know each other"**

The clients can now interact directly in a **multiparty session**: they first exchange some information...

A Motivating Example: Peer-to-Peer Game



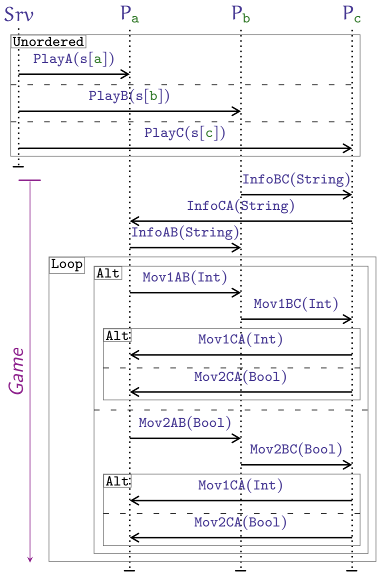
Clients P_a, P_b, P_c want to play a game as roles a, b, c via a **matchmaking server** *Srv*

The server *Srv* sends some networking data to the clients, so they **"know each other"**

The clients can now interact directly in a **multiparty session**: they first exchange some information. . .

. . . and then begin the main *Game* loop

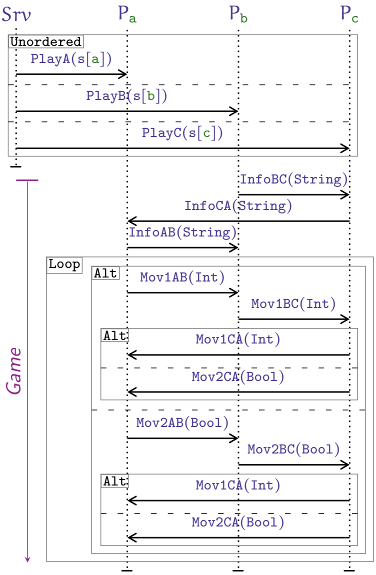
A Motivating Example: Peer-to-Peer Game



Implementing this specification is **challenging**:

- **structured protocol**
  - **choices**
  - inter-role **message dependencies**
  - **recursion**
- **non-fixed communication topology**
  - initially **client-to-server**
  - later becoming **peer-to-peer**
- risks: **protocol violations**, **deadlocks**

A Motivating Example: Peer-to-Peer Game



Implementing this specification is **challenging**:

- **structured protocol**
  - **choices**
  - inter-role **message dependencies**
  - **recursion**
- **non-fixed communication topology**
  - initially **client-to-server**
  - later becoming **peer-to-peer**
- risks: **protocol violations**, **deadlocks**

Can we provide a **formally grounded** way to address these challenges?

Our Contribution

We leverage the **multiparty session types (MPST) theory** to turn **multiparty protocol specifications** into **Scala APIs**

Our Contribution

We leverage the **multiparty session types (MPST) theory** to turn **multiparty protocol specifications** into **Scala APIs**

1. we **encode** the **full MPST calculus** into **linear π-calculus**
2. we develop an **encoding-based multiparty API generation**

Our Contribution

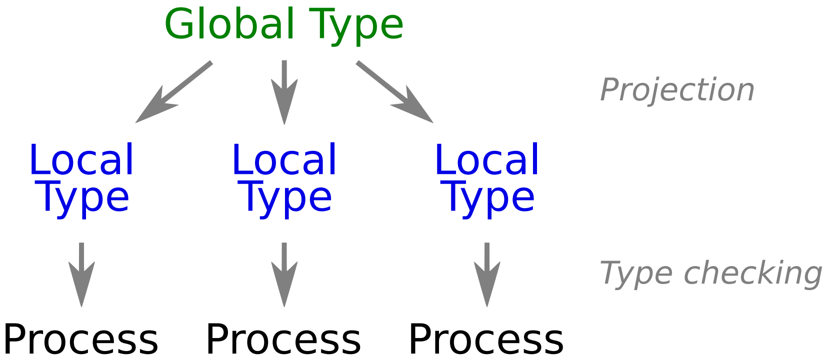
We leverage the **multiparty session types (MPST) theory** to turn **multiparty protocol specifications** into **Scala APIs**

1. we **encode** the **full MPST calculus** into **linear π-calculus**
2. we develop an **encoding-based multiparty API generation**

With this approach, the resulting Scala APIs:

- are **formally grounded** (exploit formal correctness properties)
- are **type-safe** (many protocol errors detected **at compile time**)
- are **choreographic** (no centralised orchestration middleware)
- **reuse existing libraries** for type-safe binary channels
- support **distributed multiparty session delegation** (first time!)

MPST Theory: Overview



(Honda *et al.*, POPL'08/JACM'16; Bettini *et al.*, CONCUR'08; Coppo *et al.*, MSCS'16)

MPST Theory: Protocols as Types

The **global type** G is the **game protocol** with **3 players** a, b, c:

$$G = b \to c{:}\mathtt{InfoBC}(\mathtt{String})\,.\,c \to a{:}\mathtt{InfoCA}(\mathtt{String})\,.\,a \to b{:}\mathtt{InfoAB}(\mathtt{String})\,.$$

$$\mu t.a \to b{:}\left\{ \begin{array}{l} \mathtt{Mov1AB}(\mathtt{Int}).b \to c{:}\mathtt{Mov1BC}(\mathtt{Int}).c \to a{:}\left\{ \begin{array}{l} \mathtt{Mov1CA}(\mathtt{Int}).t\,, \\ \mathtt{Mov2CA}(\mathtt{Bool}).t \end{array} \right\}, \\ \mathtt{Mov2AB}(\mathtt{Bool}).b \to c{:}\mathtt{Mov2BC}(\mathtt{Bool}).c \to a{:}\left\{ \begin{array}{l} \mathtt{Mov1CA}(\mathtt{Int}).t\,, \\ \mathtt{Mov2CA}(\mathtt{Bool}).t \end{array} \right\} \end{array} \right\}$$

MPST Theory: Protocols as Types

The **global type** G is the **game protocol** with **3 players** a, b, c:

$$G = b \to c : \mathtt{InfoBC}(\mathtt{String}) \,.\, c \to a : \mathtt{InfoCA}(\mathtt{String}) \,.\, a \to b : \mathtt{InfoAB}(\mathtt{String}) \,.$$

$$\mu t. a \to b : \left\{ \begin{array}{l} \mathtt{Mov1AB}(\mathtt{Int}).b \to c : \mathtt{Mov1BC}(\mathtt{Int}).c \to a : \left\{ \begin{array}{l} \mathtt{Mov1CA}(\mathtt{Int}).t \,, \\ \mathtt{Mov2CA}(\mathtt{Bool}).t \end{array} \right\} , \\ \mathtt{Mov2AB}(\mathtt{Bool}).b \to c : \mathtt{Mov2BC}(\mathtt{Bool}).c \to a : \left\{ \begin{array}{l} \mathtt{Mov1CA}(\mathtt{Int}).t \,, \\ \mathtt{Mov2CA}(\mathtt{Bool}).t \end{array} \right\} \end{array} \right\}$$

The **projection** $G \upharpoonright b$ yields the **(local) session type** describing how a **communication channel** should be used to play as b:

$$T_b = c!\mathtt{InfoBC}(\mathtt{String}).a?\mathtt{InfoAB}(\mathtt{String}).\mu t.a \,\&\, \left\{ \begin{array}{l} ?\mathtt{Mov1AB}(\mathtt{Int}).c!\mathtt{Mov1BC}(\mathtt{Int}).t \,, \\ ?\mathtt{Mov2AB}(\mathtt{Bool}).c!\mathtt{Mov2BC}(\mathtt{Bool}).t \end{array} \right\}$$

MPST Theory: Protocols as Types

The **global type** G is the **game protocol** with **3 players** $\mathsf{a}, \mathsf{b}, \mathsf{c}$:

$$\mathsf{G} = \mathsf{b}\to\mathsf{c}{:}\mathtt{InfoBC}(\mathtt{String}).\mathsf{c}\to\mathsf{a}{:}\mathtt{InfoCA}(\mathtt{String}).\mathsf{a}\to\mathsf{b}{:}\mathtt{InfoAB}(\mathtt{String}).$$
$$\mu\mathbf{t}.\mathsf{a}\to\mathsf{b}{:}\left\{\begin{array}{l}\mathtt{Mov1AB}(\mathtt{Int}).\mathsf{b}\to\mathsf{c}{:}\mathtt{Mov1BC}(\mathtt{Int}).\mathsf{c}\to\mathsf{a}{:}\left\{\begin{array}{l}\mathtt{Mov1CA}(\mathtt{Int}).\mathbf{t},\\ \mathtt{Mov2CA}(\mathtt{Bool}).\mathbf{t}\end{array}\right\},\\ \mathtt{Mov2AB}(\mathtt{Bool}).\mathsf{b}\to\mathsf{c}{:}\mathtt{Mov2BC}(\mathtt{Bool}).\mathsf{c}\to\mathsf{a}{:}\left\{\begin{array}{l}\mathtt{Mov1CA}(\mathtt{Int}).\mathbf{t},\\ \mathtt{Mov2CA}(\mathtt{Bool}).\mathbf{t}\end{array}\right\}\end{array}\right\}$$

The **projection** $\mathsf{G}\restriction\mathsf{b}$ yields the **(local) session type** describing how a **communication channel** should be used to play as b:

$$T_{\mathsf{b}} = \mathsf{c}!\mathtt{InfoBC}(\mathtt{String}).\mathsf{a}?\mathtt{InfoAB}(\mathtt{String}).\mu\mathbf{t}.\mathsf{a}\,\&\left\{\begin{array}{l}?\mathtt{Mov1AB}(\mathtt{Int}).\mathsf{c}!\mathtt{Mov1BC}(\mathtt{Int}).\mathbf{t},\\ ?\mathtt{Mov2AB}(\mathtt{Bool}).\mathsf{c}!\mathtt{Mov2BC}(\mathtt{Bool}).\mathbf{t}\end{array}\right\}$$

This **client-server session type** allows **delegation** for player b (*"send or receive a channel over a channel"*):

$$\mathtt{srv}?\mathtt{PlayB}(T_{\mathsf{b}}).\mathbf{end}$$

MPST Theory: Delegation

```
val msg = sb[srv].receive()
val y = msg.payload

y[c].send(InfoBC("..."))
val info = y[a].receive()
loop(y)

def loop(y) = y[a].receive() {
    case Mov1AB(p) => {
        y[c].send(Mov1BC(p))
        loop(y) }
    case Mov2AB(y) => {
        y[c].send(Mov2BC(p))
        loop(y) } }
```

A process for player b, in **pseudo-Scala**

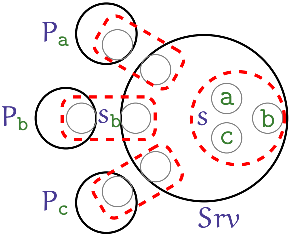
Note the **multiparty session delegation**

MPST Theory: Delegation

```
val msg = sb[srv].receive()
val y = msg.payload

y[c].send(InfoBC("..."))
val info = y[a].receive()
loop(y)

def loop(y) = y[a].receive() {
    case Mov1AB(p) => {
        y[c].send(Mov1BC(p))
        loop(y) }
    case Mov2AB(y) => {
        y[c].send(Mov2BC(p))
        loop(y) } }
```

A process for player b, in **pseudo-Scala**

Note the **multiparty session delegation**

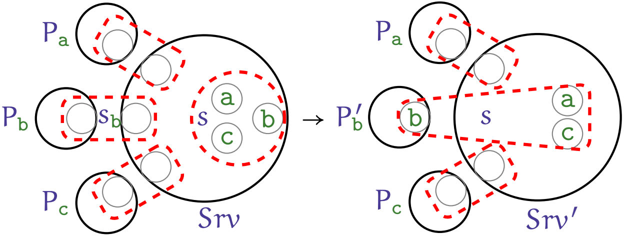


MPST Theory: Delegation

```
val msg = sb[srv].receive()
val y = msg.payload

y[c].send(InfoBC("..."))
val info = y[a].receive()
loop(y)

def loop(y) = y[a].receive() {
    case Mov1AB(p) => {
        y[c].send(Mov1BC(p))
        loop(y) }
    case Mov2AB(y) => {
        y[c].send(Mov2BC(p))
        loop(y) } }
```

A process for player b, in **pseudo-Scala**

Note the **multiparty session delegation**

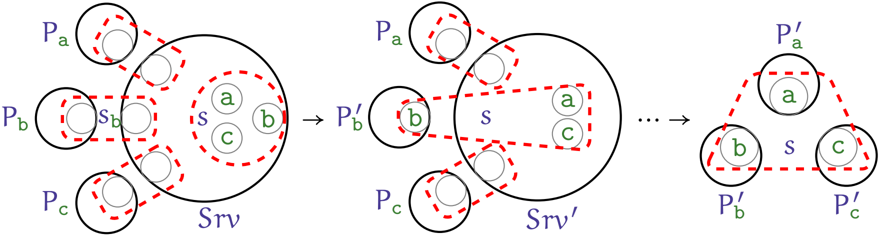


MPST Theory: Delegation

```
val msg = sb[srv].receive()
val y = msg.payload

y[c].send(InfoBC("..."))
val info = y[a].receive()
loop(y)

def loop(y) = y[a].receive() {
    case Mov1AB(p) => {
        y[c].send(Mov1BC(p))
        loop(y) }
    case Mov2AB(y) => {
        y[c].send(Mov2BC(p))
        loop(y) } }
```

A process for player b, in **pseudo-Scala**

Note the **multiparty session delegation**



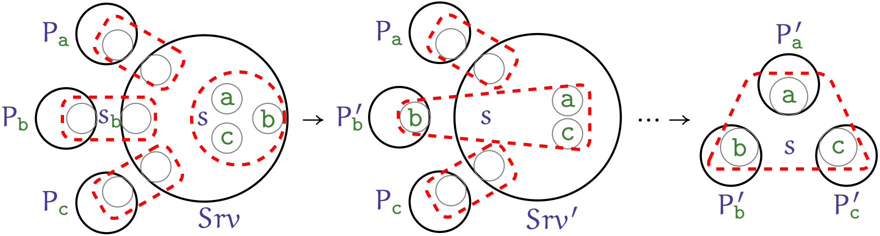
MPST Theory: Delegation and Typing

```
val msg = sb[srv].receive()
val y = msg.payload

y[c].send(InfoBC("..."))
val info = y[a].receive()
loop(y)

def loop(y) = y[a].receive() {
    case Mov1AB(p) => {
        y[c].send(Mov1BC(p))
        loop(y) }
    case Mov2AB(y) => {
        y[c].send(Mov2BC(p))
        loop(y) } }
```

A process for player b, in **pseudo-Scala**
Note the **multiparty session delegation**

The **MPST typing system** can check that:

- sb is used as $\mathtt{srv?PlayB}(T_b).\mathbf{end}$ ✓
- y is used as $T_b = G \restriction b$ ✓



MPST Theory: Delegation and Typing

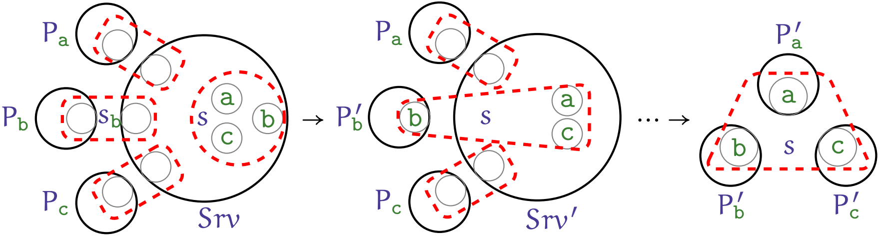
```
val msg = sb[srv].receive()
val y = msg.payload

y[c].send(InfoBC("..."))
val info = y[a].receive()
loop(y)

def loop(y) = y[a].receive() {
    case Mov1AB(p) => {
        y[c].send(Mov1BC(p))
        loop(y) }
    case Mov2AB(y) => {
        y[c].send(Mov2BC(p))
        loop(y) } }
```

A process for player b, in **pseudo-Scala**
Note the **multiparty session delegation**

The **MPST typing system** can check that:

- sb is used as $\texttt{srv?PlayB}(T_b).\mathbf{end}$ ✓
- y is used as $T_b = G \upharpoonright b$ ✓

It can also check if a **set of processes** follows a **global type** G, **without deadlocks**



From MPST Theory to Practice: Challenges

MPST offer **useful modelling and verification** features. **But:**

- **multiparty channels** are a **very high-level** concept
- the **theory is rich** and sometimes **intricate**
- calculus/types are **far from "mainstream" programming**

From MPST Theory to Practice: Challenges

MPST offer **useful modelling and verification** features. **But:**

- **multiparty channels** are a **very high-level** concept
- the **theory is rich** and sometimes **intricate**
- calculus/types are **far from "mainstream" programming**

To **"close the gap"** between theory and practice, we need to:

1. **decompose** MPST channels into **binary channels** *(e.g., TCP sockets)*
2. figure out **how to implement multiparty delegation**
3. provide **types and APIs in a "mainstream" prog. lang.**

From MPST Theory to Practice: Challenges

MPST offer **useful modelling and verification** features. **But:**

- **multiparty channels** are a **very high-level** concept
- the **theory is rich** and sometimes **intricate**
- calculus/types are **far from "mainstream" programming**

To **"close the gap"** between theory and practice, we need to:

1. **decompose** MPST channels into **binary channels** *(e.g., TCP sockets)*
   - without adding centralised *orchestration*, unlike existing theories (Caires & Pérez, FORTE'16; Carbone *et al.*, CONCUR'16)
2. figure out **how to implement multiparty delegation**
3. provide **types and APIs in a "mainstream" prog. lang.**

From MPST Theory to Practice: Challenges

MPST offer **useful modelling and verification** features. **But:**

- **multiparty channels** are a **very high-level** concept
- the **theory is rich** and sometimes **intricate**
- calculus/types are **far from "mainstream" programming**

To **"close the gap"** between theory and practice, we need to:

1. **decompose** MPST channels into **binary channels** *(e.g., TCP sockets)*
    - without adding centralised *orchestration*, unlike existing theories (Caires & Pérez, FORTE'16; Carbone *et al.*, CONCUR'16)
2. figure out **how to implement multiparty delegation**
    - unsupported in existing works (Hu & Yoshida, FASE'16/FASE'17)
3. provide **types and APIs in a "mainstream" prog. lang.**

A New Approach to "Practical" Multiparty Sessions

Binary Sessions

Encoding

Linear π-Calculus

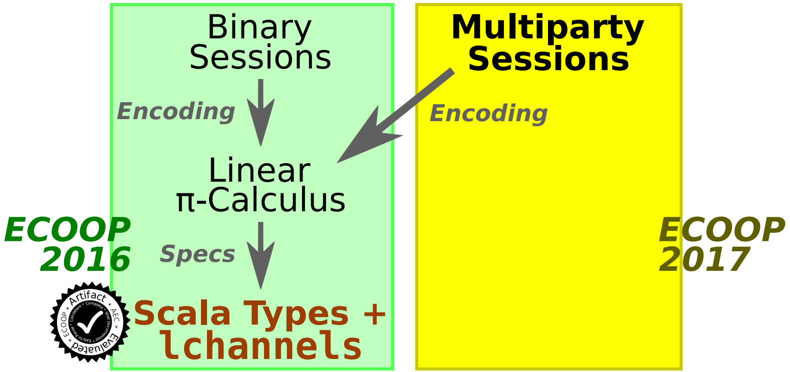
Specs

Scala Types + `lchannels`

ECOOP 2016

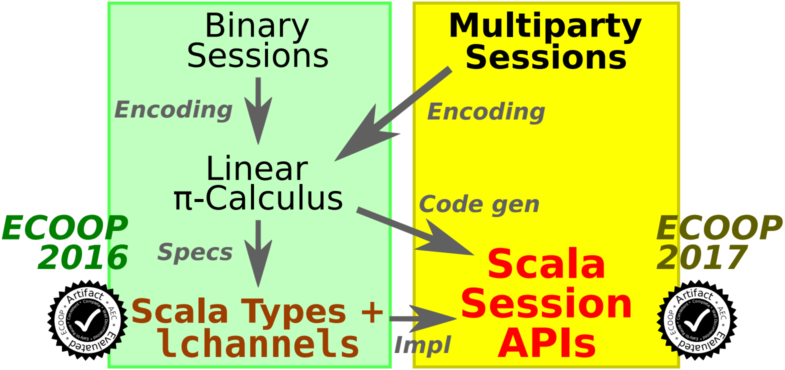
ECOOP · Artifact · AEC · Evaluated
Consistent · Complete · Well Documented · Easy to Reuse

A New Approach to "Practical" Multiparty Sessions



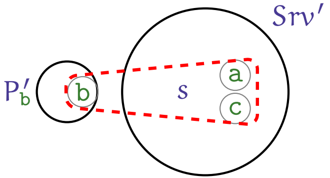
1. **encode** the **full** multiparty session calculus into **linear π-calculus**
   - π-calculus only has **binary channels**, and **no session primitives**

A New Approach to "Practical" Multiparty Sessions



1. **encode** the **full** multiparty session calculus into **linear π-calculus**
   - π-calculus only has **binary channels**, and **no session primitives**
2. use the encoding to **guide multiparty session API generation**
   - "inherit" **correctness**, reuse **code**, better **APIs**, **delegation** for free!

A Linear Decomposition of Multiparty Sessions

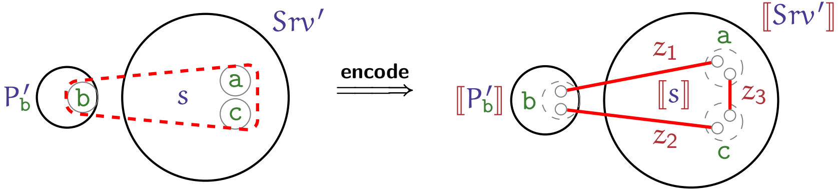


$s[b] : T_b = c!\mathtt{InfoBC}(\mathtt{String}).a?\mathtt{InfoAB}(\mathtt{String})\ldots.$

We **decompose** s **into binary linear channels**, and **encode** P'_{b} and Srv' so that they use the decomposed channels **"correctly"**:

- **no out-of protocol messages** must be sent/received
- **channel usage ordering** must be preserved

A Linear Decomposition of Multiparty Sessions

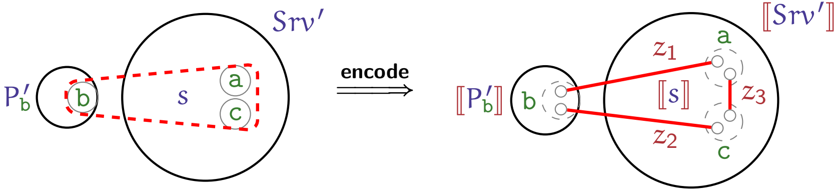


$s[b] : T_b = c!\mathtt{InfoBC}(\mathtt{String}).a?\mathtt{InfoAB}(\mathtt{String})\ldots.$

We **decompose** s **into binary linear channels**, and **encode** P'_b and Srv' so that they use the decomposed channels **"correctly"**:

- **no out-of protocol messages** must be sent/received
- **channel usage ordering** must be preserved

A Linear Decomposition of Multiparty Sessions



$s[b] : T_b = c!\mathtt{InfoBC}(\mathtt{String}).a?\mathtt{InfoAB}(\mathtt{String}).\ldots$

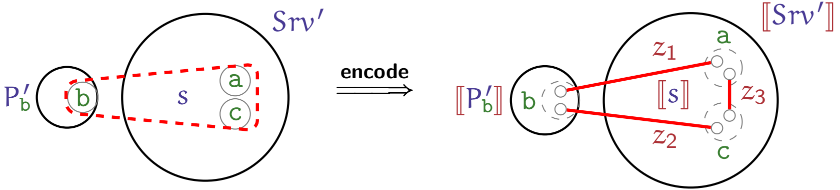
encode $\Longrightarrow$

$$z_1 : [\![T_b \upharpoonright a]\!]$$
$$z_2 : [\![T_b \upharpoonright c]\!]$$

We **decompose** s **into binary linear channels**, and **encode** P'_b and Srv' so that they use the decomposed channels **"correctly"**:

- **no out-of protocol messages** must be sent/received
- **channel usage ordering** must be preserved

A Linear Decomposition of Multiparty Sessions



$s[b] : T_b = c!\mathtt{InfoBC}(\mathtt{String}).a?\mathtt{InfoAB}(\mathtt{String}).\ldots.$

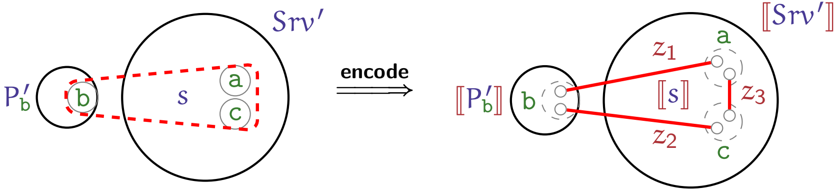
encode $\Longrightarrow$

$$z_1 : [\![T_b \restriction a]\!] = \mathbf{In}\langle \mathtt{InfoAB\_}(\mathtt{String}, \quad) \rangle$$
$$z_2 : [\![T_b \restriction c]\!] = \mathbf{Out}\langle \mathtt{InfoBC\_}(\mathtt{String}, \quad) \rangle$$

We **decompose** s **into binary linear channels**, and **encode** P'_b and Srv' so that they use the decomposed channels **"correctly"**:

- **no out-of protocol messages** must be sent/received
- **channel usage ordering** must be preserved

A Linear Decomposition of Multiparty Sessions



$s[\mathtt{b}] : T_{\mathtt{b}} = \mathtt{c}!\mathtt{InfoBC}(\mathtt{String}).\mathtt{a}?\mathtt{InfoAB}(\mathtt{String})\ldots.$

encode $\Longrightarrow$

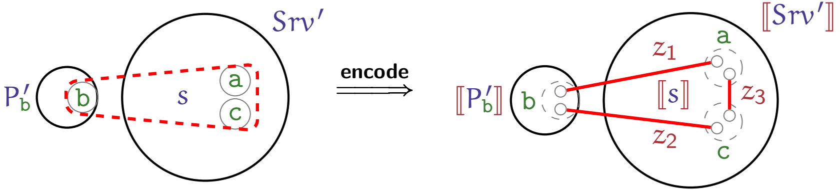
$$z_1 : [\![T_{\mathtt{b}} \upharpoonright \mathtt{a}]\!] = \mathbf{In}\langle \mathtt{InfoAB\_}(\mathtt{String}, \mathbf{In}\langle \ldots \rangle)\rangle$$

$$z_2 : [\![T_{\mathtt{b}} \upharpoonright \mathtt{c}]\!] = \mathbf{Out}\langle \mathtt{InfoBC\_}(\mathtt{String}, \mathbf{In}\langle \ldots \rangle)\rangle$$

We **decompose** s **into binary linear channels**, and **encode** $P'_{\mathtt{b}}$ and Srv' so that they use the decomposed channels **"correctly"**:

- **no out-of protocol messages** must be sent/received
- **channel usage ordering** must be preserved

A Linear Decomposition of Multiparty Sessions



$$s[\mathtt{b}] : T_{\mathtt{b}} = \mathtt{c}!\mathtt{InfoBC}(\mathtt{String}).\mathtt{a}?\mathtt{InfoAB}(\mathtt{String})....$$

$\xRightarrow{\textbf{encode}}$

$$z_1 : [\![T_{\mathtt{b}} \upharpoonright \mathtt{a}]\!] = \mathbf{In}\langle \mathtt{InfoAB\_}(\mathtt{String}, \mathbf{In}\langle \ldots \rangle)\rangle$$

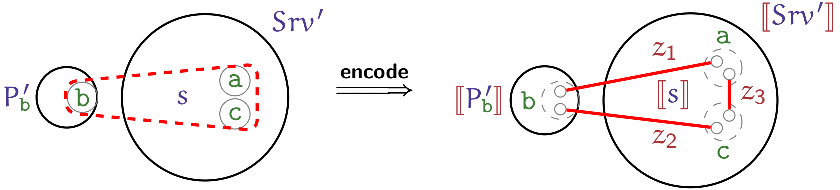
$$z_2 : [\![T_{\mathtt{b}} \upharpoonright \mathtt{c}]\!] = \mathbf{Out}\langle \mathtt{InfoBC\_}(\mathtt{String}, \mathbf{In}\langle \ldots \rangle)\rangle$$

$$[\![s[\mathtt{b}]]\!] = \left[\begin{array}{l} \mathtt{a}: z_1 \, , \\ \mathtt{c}: z_2 \end{array} \right]$$

We **decompose** s **into binary linear channels**, and **encode** $P'_{\mathtt{b}}$ and Srv' so that they use the decomposed channels **"correctly"**:

- **no out-of protocol messages** must be sent/received
- **channel usage ordering** must be preserved

A Linear Decomposition of Multiparty Sessions



$$s[\mathtt{b}] : T_{\mathtt{b}} = \mathtt{c}!\mathtt{InfoBC}(\mathtt{String}).\mathtt{a}?\mathtt{InfoAB}(\mathtt{String})\ldots.$$

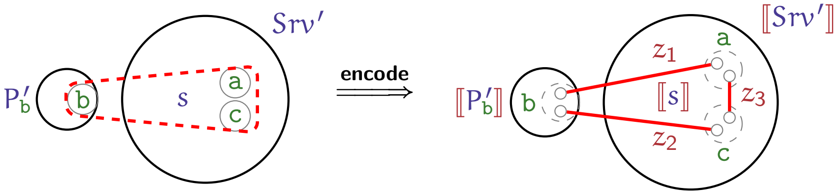
$$\xrightarrow{\textbf{encode}} \quad \begin{array}{lcl} z_1 & : & [\![T_{\mathtt{b}} \upharpoonright \mathtt{a}]\!] = \mathbf{In}\langle \mathtt{InfoAB\_}(\mathtt{String},\ \mathbf{In}\langle\ldots\rangle)\rangle \\ z_2 & : & [\![T_{\mathtt{b}} \upharpoonright \mathtt{c}]\!] = \mathbf{Out}\langle \mathtt{InfoBC\_}(\mathtt{String},\ \mathbf{In}\langle\ldots\rangle)\rangle \end{array}$$

$$[\![s[\mathtt{b}]]\!] = \left[\begin{array}{l} \mathtt{a}{:}\, z_1\,, \\ \mathtt{c}{:}\, z_2 \end{array} \right] : \left[\begin{array}{l} \mathtt{a}{:}\, [\![T_{\mathtt{b}} \upharpoonright \mathtt{a}]\!]\,, \\ \mathtt{c}{:}\, [\![T_{\mathtt{b}} \upharpoonright \mathtt{c}]\!] \end{array} \right]$$

We **decompose** s **into binary linear channels**, and **encode** $P'_{\mathtt{b}}$ and Srv' so that they use the decomposed channels **"correctly"**:

- **no out-of protocol messages** must be sent/received
- **channel usage ordering** must be preserved

A Linear Decomposition of Multiparty Sessions



$s[\mathtt{b}] : \mathsf{T}_{\mathtt{b}} = \mathtt{c}!\mathtt{InfoBC}(\mathtt{String}).\mathtt{a}?\mathtt{InfoAB}(\mathtt{String})\ldots.$

encode $\Longrightarrow$

$$z_1 \;:\; [\![\, \mathsf{T}_{\mathtt{b}} \upharpoonright \mathtt{a} \,]\!] = \mathbf{In}\langle \mathtt{InfoAB\_}(\mathtt{String},\ \mathbf{In}\langle \ldots \rangle) \rangle$$

$$z_2 \;:\; [\![\, \mathsf{T}_{\mathtt{b}} \upharpoonright \mathtt{c} \,]\!] = \mathbf{Out}\langle \mathtt{InfoBC\_}(\mathtt{String},\ \mathbf{In}\langle \ldots \rangle) \rangle$$

$$[\![s[\mathtt{b}]]\!] = \begin{bmatrix} \mathtt{a}: z_1\,, \\ \mathtt{c}: z_2 \end{bmatrix} : \begin{bmatrix} \mathtt{a}: [\![\mathsf{T}_{\mathtt{b}} \upharpoonright \mathtt{a}]\!]\,, \\ \mathtt{c}: [\![\mathsf{T}_{\mathtt{b}} \upharpoonright \mathtt{c}]\!] \end{bmatrix}$$

We **decompose** s **into binary linear channels**, and **encode** $P'_{\mathtt{b}}$ and Srv' so that they use the decomposed channels **“correctly”**:

- **no out-of-protocol messages** must be sent/received ✓
- **channel usage ordering** must be preserved ✗

Encoding of Typed Processes

Our **process encoding**:

- is **"low-level"**, close to an **imperative prog. lang.**
- uses **binary** channels **once** with **continuation-passing style**
- keeps the **communication order** of the original process ✓

Encoding of Typed Processes

Our **process encoding**:

- is **"low-level"**, close to an **imperative prog. lang.**
- uses **binary** channels **once** with **continuation-passing style**
- keeps the **communication order** of the original process ✓

$s[b]:T_b \vdash s[b][c] \oplus \langle \mathtt{InfoBC}(\text{"..."}) \rangle.P'$

Encoding of Typed Processes

Our **process encoding**:

- is **"low-level"**, close to an **imperative prog. lang.**
- uses **binary** channels **once** with **continuation-passing style**
- keeps the **communication order** of the original process ✓

$$[\![s[b]:T_b \vdash s[b][c] \oplus \langle \mathtt{InfoBC}(\text{"..."}) \rangle.P']\!] \; =$$

Encoding of Typed Processes

Our **process encoding**:

- is **"low-level"**, close to an **imperative prog. lang.**
- uses **binary** channels **once** with **continuation-passing style**
- keeps the **communication order** of the original process ✓

$$\llbracket s[\mathtt{b}]{:}T_{\mathtt{b}} \vdash s[\mathtt{b}][\mathtt{c}] \oplus \langle \mathtt{InfoBC}(\text{"..."}) \rangle.P' \rrbracket \;=$$

$$\llbracket s[\mathtt{b}]{:}T_{\mathtt{b}} \rrbracket \vdash_{\pi}$$

Encoding of Typed Processes

Our **process encoding**:

- is **“low-level”**, close to an **imperative prog. lang.**
- uses **binary** channels **once** with **continuation-passing style**
- keeps the **communication order** of the original process ✓

$$\left[\!\!\left[s[\mathrm{b}] : \mathrm{T_b} \vdash s[\mathrm{b}][\mathrm{c}] \oplus \langle \mathtt{InfoBC}(\text{"..."}) \rangle . P' \right]\!\!\right] =$$

$$[\![s[\mathrm{b}] : \mathrm{T_b}]\!] \vdash_{\pi} \mathbf{with}\, [\mathrm{a} : z_{\mathrm{a}} \,,\, \mathrm{c} : z_{\mathrm{c}}] = [\![s[\mathrm{b}]]\!]\, \mathbf{do}$$

Encoding of Typed Processes

Our **process encoding**:

- is **"low-level"**, close to an **imperative prog. lang.**
- uses **binary** channels **once** with **continuation-passing style**
- keeps the **communication order** of the original process ✓

$$\llbracket s[\mathrm{b}]:\mathrm{T_b} \vdash s[\mathrm{b}][\mathrm{c}] \oplus \langle \mathrm{InfoBC}(\text{"..."}) \rangle.P' \rrbracket \;=$$

$$\llbracket s[\mathrm{b}]:\mathrm{T_b} \rrbracket \vdash_\pi \textbf{with}\, [\mathrm{a}:z_\mathrm{a}\,,\, \mathrm{c}:z_\mathrm{c}] = \llbracket s[\mathrm{b}] \rrbracket\, \textbf{do}$$

$$(z'_\mathrm{I}, z'_\mathrm{O}) = \textbf{new\_lin\_channel}();$$

Encoding of Typed Processes

Our **process encoding**:

- is **"low-level"**, close to an **imperative prog. lang.**
- uses **binary** channels **once** with **continuation-passing style**
- keeps the **communication order** of the original process ✓

$$\llbracket s[b]:T_b \vdash s[b][c] \oplus \langle \mathtt{InfoBC}(\text{"..."}) \rangle.P' \rrbracket \;=$$

$$\llbracket s[b]:T_b \rrbracket \vdash_\pi \textbf{with}\, [a : z_a\,,\ c : z_c] = \llbracket s[b] \rrbracket\ \textbf{do}$$

$$(z'_I, z'_O) = \textbf{new\_lin\_channel}();$$

$$z_c.\textbf{send}\big(\mathtt{InfoBC}(\text{"..."}\qquad)\big);$$

Encoding of Typed Processes

Our **process encoding**:

- is **"low-level"**, close to an **imperative prog. lang.**
- uses **binary** channels **once** with **continuation-passing style**
- keeps the **communication order** of the original process ✓

$$\llbracket s[b] : T_b \vdash s[b][c] \oplus \langle \mathtt{InfoBC}(''\ldots'') \rangle . P' \rrbracket \;=$$

$$\llbracket s[b] : T_b \rrbracket \vdash_\pi \textbf{with}\, [\mathtt{a} : z_{\mathtt{a}} \,,\, \mathtt{c} : z_{\mathtt{c}}] = \llbracket s[b] \rrbracket \,\textbf{do}$$

$$(z'_I, z'_O) = \textbf{new\_lin\_channel}();$$

$$z_{\mathtt{c}}.\textbf{send}\big(\mathtt{InfoBC}(''\ldots''\,,\, z'_I)\big);$$

Encoding of Typed Processes

Our **process encoding**:

- is **"low-level"**, close to an **imperative prog. lang.**
- uses **binary** channels **once** with **continuation-passing style**
- keeps the **communication order** of the original process ✓

$$\llbracket s[b]:T_b \vdash s[b][c] \oplus \langle \mathtt{InfoBC}(\text{"..."}) \rangle.P' \rrbracket \;=$$

$$\llbracket s[b]:T_b \rrbracket \vdash_\pi \textbf{with}\ [a : z_a\ ,\ c : z_c] = \llbracket s[b] \rrbracket\ \textbf{do}$$
$$(z'_I, z'_O) = \textbf{new\_lin\_channel}();$$
$$z_c.\textbf{send}\big(\mathtt{InfoBC}(\text{"..."}\ ,\ z'_I)\big);$$
$$\textbf{let}\ \llbracket s[b] \rrbracket = [a : z_a\ ,\ c : z'_O]\ \textbf{in}\ \llbracket P' \rrbracket$$

Encoding of Typed Processes

Our **process encoding**:

- is **"low-level"**, close to an **imperative prog. lang.**
- uses **binary** channels **once** with **continuation-passing style**
- keeps the **communication order** of the original process ✓

$$\left[\!\left[s[b]:T_b \vdash s[b][c] \oplus \langle \mathtt{InfoBC}(\text{"..."}) \rangle.P' \right]\!\right] =$$

$$[\![s[b]:T_b]\!] \vdash_\pi \textbf{with } [a : z_a\,,\; c : z_c] = [\![s[b]]\!] \textbf{ do}$$
$$(z'_I, z'_O) = \textbf{new\_lin\_channel}();$$
$$z_c.\textbf{send}\big(\mathtt{InfoBC}(\text{"..."}\,,\, z'_I)\big);$$
$$\textbf{let } [\![s[b]]\!] = [a : z_a\,,\; c : z'_O] \textbf{ in } [\![P']\!]$$

Moreover, our encoding is **choreographic**: $[\![P \mid Q]\!] = [\![P]\!] \mid [\![Q]\!]$

- unlike previous works (Caires & Pérez, FORTE'16; Carbone *et al.*, CONCUR'16)

Formal Correctness Properties

Encoding is type-preserving. $\Gamma \vdash P$ implies $[\![\Gamma]\!] \vdash_{\pi} [\![P]\!]$.

Formal Correctness Properties

Encoding is type-preserving. $\Gamma \vdash P$ implies $[\![\Gamma]\!] \vdash_\pi [\![P]\!]$.

Operational correspondence. (Gorla, Inf. & Comput., 2010)

If $\varnothing \vdash P$, then:

1. **(Completeness)** $P \to^* P'$ implies $\exists \widetilde{x}, P''$ such that $[\![P]\!] \to^* (\nu\widetilde{x})P''$ and $P'' = [\![P']\!]$;
2. **(Soundness)** $[\![P]\!] \to^* P_*$ implies $\exists \widetilde{x}, P'', P'$ such that $P_* \to^* (\nu\widetilde{x})P''$ and $P \to^* P'$ and $[\![P']\!] \xrightarrow{\textbf{with}}{}^* P''$.

Formal Correctness Properties

Encoding is type-preserving. $\Gamma \vdash P$ implies $[\![\Gamma]\!] \vdash_\pi [\![P]\!]$.

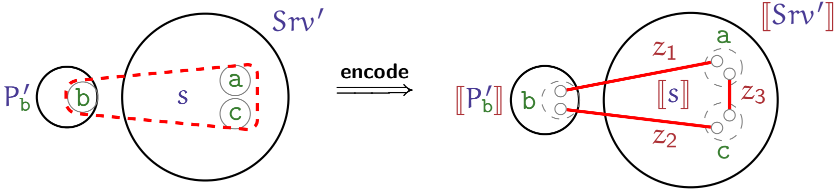
Operational correspondence. (Gorla, Inf. & Comput., 2010)
If $\varnothing \vdash P$, then:

1. **(Completeness)** $P \rightarrow^* P'$ implies $\exists \widetilde{x}, P''$ such that $[\![P]\!] \rightarrow^* (\nu \widetilde{x})P''$ and $P'' = [\![P']\!]$;
2. **(Soundness)** $[\![P]\!] \rightarrow^* P_*$ implies $\exists \widetilde{x}, P'', P'$ such that $P_* \rightarrow^* (\nu \widetilde{x})P''$ and $P \rightarrow^* P'$ and $[\![P']\!] \xrightarrow{\textbf{with}}{}^* P''$.

Our linear decomposition is precise!
$[\![\Gamma]\!]$ is defined *if and only if* Γ is well-formed (*"consistent"*).

- $\Longleftarrow$: we support the full MPST theory
- $\Longrightarrow$: we uncover a deep connection between MPST and π-calculus

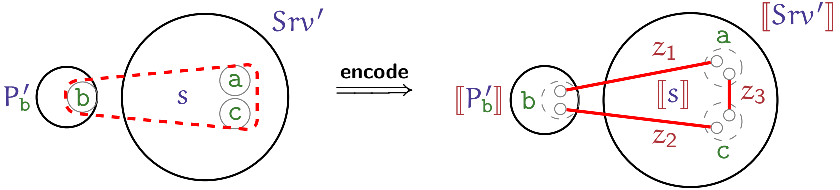
Multiparty Channels, in Scala



$s[b]$: $T_b = c!\mathtt{InfoBC}(\mathtt{String}).a?\mathtt{InfoAB}(\mathtt{String})\ldots$

encode $\Longrightarrow$ $[\![s[b]]\!]$: $[\![T_b]\!] = \left[\begin{array}{l} a : \mathbf{In}\langle \mathtt{InfoAB\_}(\mathtt{String}, \mathbf{In}\langle \ldots\rangle)\rangle, \\ c : \mathbf{Out}\langle \mathtt{InfoBC\_}(\mathtt{String}, \mathbf{In}\langle \ldots\rangle)\rangle \end{array}\right]$

Multiparty Channels, in Scala



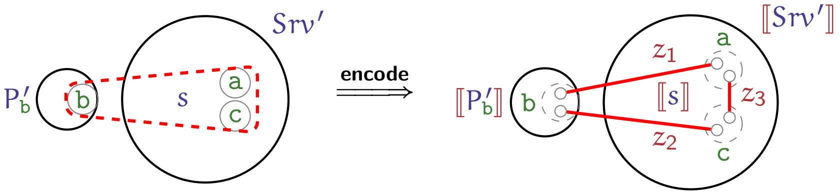
$$s[\mathtt{b}] : T_{\mathtt{b}} = \mathtt{c}!\mathtt{InfoBC}(\mathtt{String}).\mathtt{a}?\mathtt{InfoAB}(\mathtt{String})\ldots.$$

$$\stackrel{\textbf{encode}}{\Longrightarrow} \quad [\![s[\mathtt{b}]]\!] : [\![T_{\mathtt{b}}]\!] = \left[\begin{array}{l} \mathtt{a} : \mathbf{In}\langle \mathtt{InfoAB\_}(\mathtt{String}, \mathbf{In}\langle \ldots \rangle)\rangle, \\ \mathtt{c} : \mathbf{Out}\langle \mathtt{InfoBC\_}(\mathtt{String}, \mathbf{In}\langle \ldots \rangle)\rangle \end{array} \right]$$

A **multiparty channel** typed by $[\![T_{\mathtt{b}}]\!]$ is a **Scala object** of type:

```
case class Tb( a:              , c:                 )
```

Multiparty Channels, in Scala



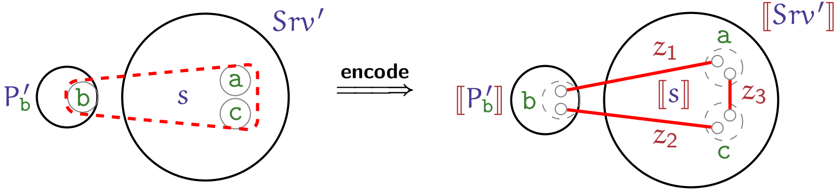
$$s[b] \;:\; T_b = \mathtt{c!InfoBC(String).a?InfoAB(String)....}$$

$$\xrightarrow{\textbf{encode}} \quad [\![s[b]]\!] \;:\; [\![T_b]\!] = \left[\begin{array}{l} \mathtt{a}:\mathbf{In}\langle \mathtt{InfoAB\_(String,}\ \mathbf{In}\langle\ldots\rangle)\rangle\,, \\ \mathtt{c}:\mathbf{Out}\langle \mathtt{InfoBC\_(String,}\ \mathbf{In}\langle\ldots\rangle)\rangle \end{array}\right]$$

A **multiparty channel** typed by $[\![T_b]\!]$ is a **Scala object** of type:

```
case class Tb( a: In[InfoAB], c: Out[InfoBC] )
```

Multiparty Channels, in Scala



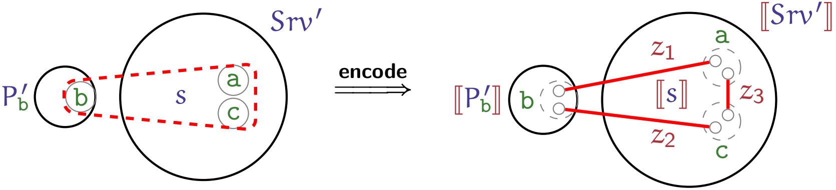
$s[b] \;:\; T_b = c!\mathtt{InfoBC}(\mathtt{String}).a?\mathtt{InfoAB}(\mathtt{String})\ldots.$

$\xRightarrow{\textbf{encode}} \quad [\![s[b]]\!] \;:\; [\![T_b]\!] = \begin{bmatrix} a : \mathbf{In}\langle \mathtt{InfoAB\_}(\mathtt{String},\ \mathbf{In}\langle\ldots\rangle)\rangle, \\ c : \mathbf{Out}\langle \mathtt{InfoBC\_}(\mathtt{String},\ \mathbf{In}\langle\ldots\rangle)\rangle \end{bmatrix}$

A **multiparty channel** typed by $[\![T_b]\!]$ is a **Scala object** of type:

```
case class Tb( a: In[InfoAB], c: Out[InfoBC] )

case class InfoAB( p: String, cont:In[...] )
case class InfoBC( p: String, cont:In[...] )
```

Multiparty Channels, in Scala



$s[b] \; : \; T_b = \mathtt{c!InfoBC}(\mathtt{String}).\mathtt{a?InfoAB}(\mathtt{String})\ldots.$

$\xRightarrow{\textbf{encode}} \quad [\![s[b]]\!] \; : \; [\![T_b]\!] = \begin{bmatrix} \mathtt{a}: \mathbf{In}\langle \mathtt{InfoAB\_}(\mathtt{String}, \; \mathbf{In}\langle \ldots \rangle)\rangle, \\ \mathtt{c}: \mathbf{Out}\langle \mathtt{InfoBC\_}(\mathtt{String}, \; \mathbf{In}\langle \ldots \rangle)\rangle \end{bmatrix}$

A **multiparty channel** typed by $[\![T_b]\!]$ is a **Scala object** of type:

```
case class Tb( a: In[InfoAB], c: Out[InfoBC] )
case class InfoAB( p: String, cont:In[...] )
case class InfoBC( p: String, cont:In[...] )
```

`In[·]`/`Out[·]` are provided by `lchannels` (Scalas & Yoshida, ECOOP'16)

Tuples of channels (like s_b) can be **delegated (remotely) for free!**

Multiparty Channel Endpoints, in Scala (cont'd)

To **guide channel usage order** and **avoid deadlocks**, we **enrich channel tuples** with **typed send/receive methods**

Their implementation **is based on our process encoding**

$$T_b \quad = \quad c!\texttt{InfoBC}(\texttt{String}) \,.\, a?\texttt{InfoAB}(\texttt{String}) \,.\, \ldots$$

```
case class T_b(a: In[InfoAB], c: Out[InfoBC])
```

Multiparty Channel Endpoints, in Scala (cont'd)

To **guide channel usage order** and **avoid deadlocks**, we **enrich channel tuples** with **typed send/receive methods**

Their implementation **is based on our process encoding**

$$T_b \;=\; c!\mathtt{InfoBC}(\mathtt{String}) \,.\, a?\mathtt{InfoAB}(\mathtt{String}) \,.\, \ldots$$

```
case class Tb( a: In[InfoAB], c: Out[InfoBC] )
{
    def send(v: String) = {      // v: payload of InfoBC msg
        val c′ = c !! InfoBC(v)_ // send v, return continuation
        T′b(a, c′)               // return "continuation object"
    }
}
```

Multiparty Channel Endpoints, in Scala (cont'd)

To **guide channel usage order** and **avoid deadlocks**, we **enrich channel tuples** with **typed send/receive methods**

Their implementation **is based on our process encoding**

$$T_b = c!\mathtt{InfoBC}(\mathtt{String}).a?\mathtt{InfoAB}(\mathtt{String})....$$

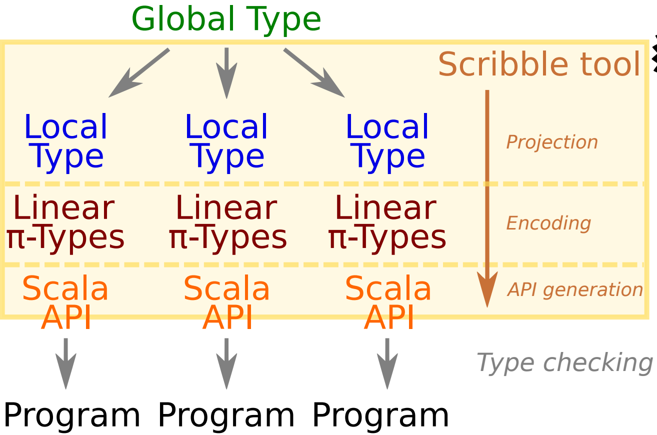
```
case class Tb( a: In[InfoAB], c: Out[InfoBC] )
{
    def send(v: String) = {      // v: payload of InfoBC msg
        val c′ = c !! InfoBC(v)_ // send v, return continuation
        T′b(a, c′)               // return "continuation object"
    }
}
```

The resulting API includes **dynamic linearity checks**, and is:

- **fully type safe** (no type casts)
- **complete** (full MPSTs, incl. type projection/merge and delegation)
- **simple** (most functionality comes from `lchannels`)
- **mechanical** (so we can generate it **automatically!**)

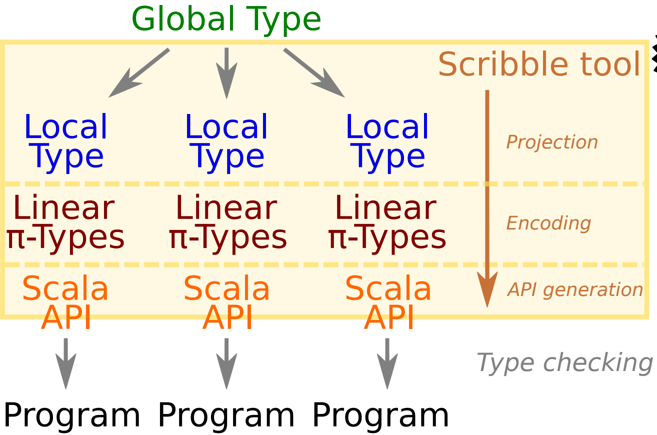
Artifact: Scala API Generation in Scribble

We extended the **Scribble protocol verification tool** to **autogenerate Scala APIs**, following our formal encoding



Artifact: Scala API Generation in Scribble

We extended the **Scribble protocol verification tool** to **autogenerate Scala APIs**, following our formal encoding



Tutorial and examples: peer-to-peer game, HTTP server. . .

Artifact: Scala API Generation Usage

A **working implementation** of a client playing the game as b, based on our Scribble-generated APIs

```
def client(c: MPPlayB) = { // "c" is the channel to the game server
  val g = c.receive().p // Receive multiparty game channel

  val i = g.send(InfoBC("...")).receive() // Send info to C, recv from A
  loop(i.cont) // Game loop
}

def loop(g: MPMov1ABOrMov2AB): Unit = {
  g.receive() match {                  // Check A's move
    case Mov1AB(p, cont) => {
      val g2 = cont.send(Mov1BC(p)) // cont only allows to send Mov1BC
      loop(g2)                      // Keep playing
    }
    case Mov2AB(p, cont) => {
      val g2 = cont.send(Mov2BC(p)) // cont only allows to send Mov2BC
      loop(g2)                      // Keep playing
} } }
```

Artifact: Scala API Generation Usage

A **working implementation** of a client playing the game as b, based on our Scribble-generated APIs with **static protocol checks**

```
def client(c: MPPlayB) = { // "c" is the channel to the game server
  val g = c.receive().p // Receive multiparty game channel

  val i = g.send(InfoBC("...")).receive() // Send info to C, recv from A
  loop(i.cont) // Game loop
}

def loop(g: MPMov1ABOrMov2AB): Unit = {
  g.receive() match {                  // Check A's move
    case Mov1AB(p, cont) => {
      val g2 = cont.send(Mov2BC(true)) ✗ // cont                    1BC
      loop(g2)                         // Keep pl
    }
    case Mov2AB(p, cont) => {
      val g2 = cont.send(Mov2BC(p)) // cont only allows to send Mov2BC
      loop(g2)                      // Keep playing
} } }
```

Type mismatch
found: Mov2BC
required: Mov1BC

Artifact: Scala API Generation Usage

A **working implementation** of a client playing the game as b, based on our Scribble-generated APIs with **static protocol checks**

```
def client(c: MPPlayB) = { // "c" is the channel to the game server
  val g = c.receive().p // Receive multiparty game channel

  val i = g.send(InfoBC("...")).receive() // Send info to C, recv from A
  loop(i.cont) // Game loop
}

def loop(g: MPMov1ABOrMov2AB): Unit = {
  g.receive() match {                    // Check A's move
    case Mov1AB(p, cont) => {
      val g2 = cont.send(Mov1BC(p)) // cont only allows to send Mov1BC
      loop(g2)                       // Keep playing
    }
  ✗
} }
```

Match may not be exhaustive

It would fail on the input: `Mov2AB(_,_)`

Conclusions

We presented the **first choreographic encoding** of the **"full" MPST calculus** into **linear π-calculus**

- key: **type-preserving decomposition into linear π-types**
- important achievement since *Session Types Revisited*

  (Dardha, Giachino, Sangiorgi. PPDP'12)

Conclusions

We presented the **first choreographic encoding** of the **"full" MPST calculus** into **linear π-calculus**

- key: **type-preserving decomposition into linear π-types**
- important achievement since *Session Types Revisited*
  (Dardha, Giachino, Sangiorgi. PPDP'12)

Our encoding gives the **formal basis** for a **complete implementation of multiparty sessions**, in Scala + `lchannels`

- the **first** including **(distributed) multiparty delegation**

Conclusions

We presented the **first choreographic encoding** of the **"full" MPST calculus** into **linear π-calculus**

- key: **type-preserving decomposition into linear π-types**
- important achievement since *Session Types Revisited*
  (Dardha, Giachino, Sangiorgi. PPDP'12)

Our encoding gives the **formal basis** for a **complete implementation of multiparty sessions**, in Scala + `lchannels`

- the **first** including **(distributed) multiparty delegation**

Future work:

- adapt to **other languages and binary session implementations**
  - Haskell, OCaml, Rust, ... (might not support distribution)
- **reuse and compare theoretical results and tools**
  - e.g., **deadlock freedom** (with **interleaved sessions**)
    - MPSTs (Bettini, Coppo *et al.*, CONCUR'08 ...)
    - π-calculus, with TyPiCal tool (Kobayashi *et al.*, CONCUR'06 ...)

Thank you!

Try Scribble and `lchannels`!

http://scribble.org

http://alcestes.github.io/lchannels

ECOOP 2016

ECOOP 2017