A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming

Alceste Scalas\textsuperscript{1}  \textbf{Ornela Dardha}\textsuperscript{2}  Raymond Hu\textsuperscript{1}  Nobuko Yoshida\textsuperscript{1}

Imperial College London  University of Glasgow
(1)  (2)

Open Problems in Concurrency Theory — Vien, 27 June 2017

Supported by the UK EPSRC grant EP/K034413/1, \textit{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 matching 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** $Srv$.

The server $Srv$ 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 $\pi$-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 $\pi$-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

Global Type

Local Type
Local Type
Local Type

Process
Process
Process

Projection
Type checking

(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 $a$, $b$, $c$:

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

$$\mu t. a \to b : \{\begin{cases}
\text{Mov1AB(Int). } b \to c : \text{Mov1BC(Int)}. c \to a : \{\begin{cases}
\text{Mov1CA(Int). } t, \\
\text{Mov2CA(Bool). } t
\end{cases}
\}, \\
\text{Mov2AB(Bool). } b \to c : \text{Mov2BC(Bool)}. c \to a : \{\begin{cases}
\text{Mov1CA(Int). } t, \\
\text{Mov2CA(Bool). } t
\end{cases}
\}
\}$$

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

$$T_b = c ! \text{InfoBC(String)}. a ? \text{InfoAB(String)}. \mu t. a & \{\begin{cases}
\text{Mov1AB(Int). } b \to c : \text{Mov1BC(Int)}. c \to a : \{\begin{cases}
\text{Mov1CA(Int). } t, \\
\text{Mov2CA(Bool). } t
\end{cases}
\}, \\
\text{Mov2AB(Bool). } b \to c : \text{Mov2BC(Bool)}. c \to a : \{\begin{cases}
\text{Mov1CA(Int). } t, \\
\text{Mov2CA(Bool). } t
\end{cases}
\}
\}$$

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

$$\text{srv } ? \text{PlayB}(T_b). \text{end}$$
MPST Theory: Protocols as Types

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

$$G = b \rightarrow_{a} c : \text{InfoBC(String)}. c \rightarrow_{a} a : \text{InfoCA(String)}. a \rightarrow_{b} b : \text{InfoAB(String)}.$$ 

$$\mu t. a \rightarrow_{b} \begin{cases} 
\text{Mov1AB(Int). } b \rightarrow_{c} c : \text{Mov1BC(Int)}. c \rightarrow_{a} a: \begin{cases} 
\text{Mov1CA(Int). } t, \\
\text{Mov2CA(Bool). } t
\end{cases}, \\
\text{Mov2AB(Bool). } b \rightarrow_{c} c : \text{Mov2BC(Bool)}. c \rightarrow_{a} a: \begin{cases} 
\text{Mov1CA(Int). } t, \\
\text{Mov2CA(Bool). } t
\end{cases}
\end{cases}.$$ 

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

$$T_b = c ! \text{InfoBC(String)}. a ? \text{InfoAB(String)}. \mu t. a & \begin{cases} 
? \text{Mov1AB(Int). } c ! \text{Mov1BC(Int). } t, \\
? \text{Mov2AB(Bool). } c ! \text{Mov2BC(Bool). } t
\end{cases}.$$
MPST Theory: Protocols as Types

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

$G = b \rightarrow c : \text{InfoBC(String)}. c \rightarrow a : \text{InfoCA(String)}. a \rightarrow b : \text{InfoAB(String)}.$

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

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

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

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

$srv? \text{PlayB}(T_b). \text{end}$
**MPST Theory: Delegation**

```scala
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**

```scala
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**

```scala
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

```scala
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 srv?PlayB(T_b).end ✓
- y is used as T_b = G ▪ b ✓
**MPST Theory: Delegation and Typing**

```scala
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:

- \( \text{sb} \) is used as \( \text{srv?PlayB}(T_b)\) [OK]
- \( y \) is used as \( T_b = G \upharpoonright b \) [OK]

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

1. encode the full multiparty session calculus into linear π-calculus
2. use the encoding to guide multiparty session API generation

Encodings

Binary Sessions

Linear π-Calculus

Specs

Scala Types + lchannels

ECOOP 2016
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 $\pi$-calculus
   - $\pi$-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

We decompose \( s \) into binary linear channels, and encode \( P_b' \) and \( S_{rv}' \) so that they use the decomposed channels “correctly”:

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

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

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

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

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

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

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
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

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

\[
\begin{align*}
\text{s}[b]: T_b \leftarrow \text{s}[b][c] \oplus \langle \text{InfoBC}("...") \rangle . P' \\
\end{align*}
\]
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

\[
\begin{align*}
[s[b]:T_b] &\vdash s[b][c] \oplus (\text{InfoBC("...")})P' \\
[s[b]:T_b] &\vdash_\pi
\end{align*}
\]

Moreover, our encoding is choreographic:

\[
J_P \parallel Q \equiv J_P \parallel J_Q
\]

Unlike previous works (Caires & Pérez, FORTE’16; Carbone et al., CONCUR’16)
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

\[
\begin{align*}
[s[b]:T_b] & \xleftarrow{\text{\texttt{\textbackslash s\[b]\[c\] \oplus \langle \text{InfoBC}("..."\rangle}.P'}} = \\
[s[b]:T_b] & \xleftarrow{\pi} \text{\texttt{with[a:z_a, c:z_c] = [s[b]] do}}
\end{align*}
\]
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

\[
\begin{align*}
[s[b]:T_b &\vdash s[b][c] \oplus \langle \text{InfoBC("..."})\rangle.P'] \\
[s[b]:T_b] &\vdash \pi \text{with}[a:z_a, c:z_c] = [s[b]] \text{ do} \\
(z'_I, z'_O) &\rightarrow \text{new_lin_channel}();
\end{align*}
\]
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

\[
\begin{align*}
[s[b]:T_b] &\leftarrow s[b][c] \oplus \langle \text{InfoBC(”...”)} \rangle . P' \\
[s[b]:T_b] &\leftarrow \pi \ 	ext{with} \ [a: z_a, c: z_c] = [s[b]] \ 	ext{do} \\
& \quad (z'_I, z'_O) = \text{new_lin_channel}(); \\
& \quad z_c.\text{send}(\text{InfoBC(”...”)});
\end{align*}
\]
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

\[
\left[ s[b]:T_b \vdash s[b][c] \oplus \langle \text{InfoBC("...")}\rangle.P' \right] = \\
\left[ s[b]:T_b \vdash_{\pi} \text{with } [a:z_a, c:z_c] = [s[b]] \text{ do} \right.
\]
\[
(z'_I, z'_O) = \text{new_lin_channel}(); \\
z_c.\text{send(InfoBC("...", z'_I))};
\]
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 ✓

\[
\begin{align*}
[s[b]:T_b &\leftarrow s[b][c] \oplus \langle \text{InfoBC}(\text{"..."})\rangle.P'] = \\
[s[b]:T_b &\leftarrow \pi \text{with } [a:z_a, c:z_c] = [s[b]] \text{ do} \\
(z'_I, z'_O) &= \text{new_lin_channel}(); \\
z_c.\text{send}(\text{InfoBC}(\text{"..."}, z'_1)); \\
\text{let } [s[b]] = [a:z_a, c:z'_O] \text{ in } [P']
\end{align*}
\]
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

\[ [s[b]:T_b \leftarrow s[b][c] \oplus \langle \text{InfoBC}("...").P' \rangle = \]
\[ [s[b]:T_b] \leftarrow \pi \text{ with } [a:z_a, c:z_c] = [s[b]] \text{ do} \]
\[ (z'_I, z'_O) = \text{new_lin_channel}(); \]
\[ z_c.\text{send}(\text{InfoBC}("...", z'_I)); \]
\[ \text{let } [s[b]] = [a:z_a, c:z'_O] \text{ in } [P'] \]

Moreover, our encoding is choreographic: \( [P|Q] = [P] \mid [Q] \)

- 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 \( \llbracket \Gamma \rrbracket \vdash_{\pi} \llbracket P \rrbracket \).

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

If \( \emptyset \vdash P \), then:

1. **(Completeness)** \( P \rightarrow^* P' \) implies \( \exists \tilde{x}, P'' \) such that \( \llbracket P \rrbracket \rightarrow^* (\forall \tilde{x})P'' \) and \( P'' = \llbracket P' \rrbracket \);

2. **(Soundness)** \( \llbracket P \rrbracket \rightarrow^* P_* \) implies \( \exists \tilde{x}, P'', P' \) such that \( P_* \rightarrow^* (\forall \tilde{x})P'' \) and \( P \rightarrow^* P' \) and \( \llbracket P' \rrbracket \xrightarrow{\text{with}}^* P'' \).
Formal Correctness Properties

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

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

1. (Completeness) $P \rightarrow^{*} P'$ implies $\exists \tilde{x}, P''$ such that $[P] \rightarrow^{*} (\forall \tilde{x})P''$ and $P'' = [P']$;

2. (Soundness) $[P] \rightarrow^{*} P_\pi$ implies $\exists \tilde{x}, P'', P'$ such that $P_\pi \rightarrow^{*} (\forall \tilde{x})P''$ and $P \rightarrow^{*} P'$ and $[P'] \xrightarrow{\text{with}} * P''$.

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

- $\iff$: we support the full MPST theory
- $\implies$: we uncover a deep connection between MPST and $\pi$-calculus
Multiparty Channels, in Scala

A multiparty channel typed by $J^T b K$ is a Scala object of type:

```
  case class T[b](a: In[InfoAB(String)], c: Out[InfoBC(String)])
```

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

Tuples of channels (like $S_b$) can be delegated (remotely) for free!

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

$$\xrightarrow{\text{encode}} [s[b]] : [T_b] = \begin{bmatrix}
  a : \text{In}\langle\text{InfoAB}_{-}(\text{String, In}\langle\ldots\rangle)\rangle, \\
  c : \text{Out}\langle\text{InfoBC}_{-}(\text{String, In}\langle\ldots\rangle)\rangle
\end{bmatrix}$$
Multiparty Channels, in Scala

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

```scala
case class T_b( a: , c: )
```
Multiparty Channels, in Scala

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

```scala
case class T_b( a: In[InfoAB], C: Out[InfoBC] )
```
Multiparty Channels, in Scala

\[ Srv' \]

\[ P_b' \]

\[ s \]

\[ a \]

\[ c \]

\[ \text{encode} \]

\[ [s[b]] : [T_b] = \begin{cases} a : \text{In}(\text{InfoAB}(\text{String}, \text{In}(\ldots))) , \\ c : \text{Out}(\text{InfoBC}(\text{String}, \text{In}(\ldots))) \end{cases} \]

A multiparty channel typed by \([T_b]\) is a Scala object of type:

```scala
case class T_b( a: In[InfoAB], C: Out[InfoBC] )
case class InfoAB( p: String, cont: In[...] )
case class InfoBC( p: String, cont: In[...] )
```
Multiparty Channels, in Scala

\[
\begin{align*}
\text{encode} & \quad : \quad [s[b]] : [T_b] = \left[ a : \text{In}(\text{InfoAB}(\text{String}, \text{In}[\ldots])),
\right.
\left. c : \text{Out}(\text{InfoBC}(\text{String}, \text{In}[\ldots])) \right]
\end{align*}
\]

A multiparty channel typed by \([T_b]\) is a Scala object of type:

```scala
case class Tb(a: In[InfoAB], C: Out[InfoBC])
case class InfoAB(p: String, cont: In[\ldots])
case class InfoBC(p: String, cont: In[\ldots])
```

In[\ldots]/Out[\ldots] 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:

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

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

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!)
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

\[ T_b = c!\text{InfoBC}(\text{String}).a?\text{InfoAB}(\text{String}). . . . \]

case class \( T_b(a: \text{In[InfoAB]}, c: \text{Out[InfoBC]}) \) {
  
  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!)
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:

\[ T_b = c!\text{InfoBC}(\text{String}).a?\text{InfoAB}(\text{String}) \ldots \]

case class \( T_b(a: \text{In}[\text{InfoAB}], c: \text{Out}[\text{InfoBC}]) \) {
  def send(v: String) = {
    // v: payload of InfoBC msg
    val c' = c !! InfoBC(v) // send v, return continuation
    T'(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 1channels)
- **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.

![Diagram showing the process from Global Type to Scala API generation]

- **Global Type**
- **Local Type**
- **Linear π-Types**
- **Scala API**
- **Projection**
- **Encoding**
- **API generation**
- **Type checking**

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

**Consistent** *Complete* *Well Documented* *Easy to Reuse* *Evaluated* ECOOP Artifact AEC
Artifact: Scala API Generation in Scribble

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

Global Type

Local Type

Local Type

Local Type

Linear π-Types

Linear π-Types

Linear π-Types

Scala API

Scala API

Scala API

Scribble tool

Projection

Encoding

API generation

Type checking

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

```scala
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: MPMov1AB0rMov2AB): 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.

```scala
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 {
    case Mov1AB(p, cont) => {
      val g2 = cont.send(Mov2BC(true)) // cont only allows to send Mov2BC
      loop(g2) // Keep playing
    }
    case Mov2AB(p, cont) => {
      val g2 = cont.send(Mov2BC(p)) // cont only allows to send Mov2BC
      loop(g2) // Keep playing
    }
  }
}
```

Type mismatch found:

- 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**

```scala
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: MPMov1AB0rMov2AB): Unit = {
  g.receive() match {
    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 $\pi$-calculus

- key: **type-preserving decomposition into linear $\pi$-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 \( \pi \)-calculus**

- key: **type-preserving decomposition into linear \( \pi \)-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 ...)
    - **\( \pi \)-calculus**, with TyPiCal tool (Kobayashi *et al.*, CONCUR’06 ...)

---

*This document is a draft and not intended for publication.*
Thank you!
Try Scribble and lchannels!

http://scribble.org
http://alcestes.github.io/lchannels

ECOOP 2016
ECOOP 2017