Introduction to Session Types

Ornela Dardha

School of Computing Science
University of Glasgow, UK

1st Scottish Programming Languages and Verification (SPLV)
Summer School
Glasgow, August 5–9, 2019

Session Types in One Slide

- In complex distributed systems communicating participants agree on a protocol to follow, specifying *type* and *direction* of data exchanged.
- Session types are a type formalism used to model structured communication-based programming.
- Guarantee *privacy*, *communication safety* and *session fidelity*.
- Designed for
  - π- calculus
  - functional languages
  - object-oriented languages
  - binary or multiparty communication
  - ...

Outline

Session Types

- Session types were born more than 25 years ago.
- The π-calculus is the original and most used framework.
- Seminal work:
  - Honda, "*Types for Dyadic Interaction*", CONCUR 1993.
  - Takeuchi, Honda & Kubo, "*An Interaction-Based Language and its Typing System*", PARLE 1994.
  - Honda, Vasconcelos & Kubo, "*Language Primitives and Type Discipline for Structured Communication-Based Programming*", ESOP 1998.
    Awarded the *ETAPS Test-of-Time Award* at ETAPS 2019.

Session Types

- Since their appearance, session types have developed into a significant theme in programming languages.
- Computing has moved from the era of data processing to the era of communication.
- Data types codify the structure of data and make it available to programming tools.
- Session types codify the structure of communication and make it available to programming tools.

Outline

The Maths Server and Client: Types / Protocols

- The session type of the server's channel endpoint:

$$
\begin{aligned}
S \quad \triangleq \quad \&\{\, & \mathit{add} : ?\mathtt{Int}.?\mathtt{Int}.!\mathtt{Int}.S, \\
& \mathit{neg} : ?\mathtt{Int}.!\mathtt{Int}.S \\
& \mathit{quit} : \mathtt{end} \,\}
\end{aligned}
$$

The Maths Server and Client: Types / Protocols

- The session type of the server's channel endpoint:

$$S \triangleq \&\{\, \mathit{add} : ?\mathtt{Int}.?\mathtt{Int}.!\mathtt{Int}.S, \\ \mathit{neg} : ?\mathtt{Int}.!\mathtt{Int}.S \\ \mathit{quit} : \mathsf{end} \,\}$$

- The session type of the client's channel endpoint:

$$C \triangleq \oplus\{\, \mathit{add} : !\mathtt{Int}.!\mathtt{Int}.?\mathtt{Int}.C, \\ \mathit{neg} : !\mathtt{Int}.?\mathtt{Int}.C \\ \mathit{quit} : \mathsf{end} \,\}$$

The Maths Server and Client: Types / Protocols

- The session type of the server's channel endpoint:

$$S \triangleq \&\{\ add : ?\texttt{Int}.?\texttt{Int}.!\texttt{Int}.S, \\ neg : ?\texttt{Int}.!\texttt{Int}.S \\ quit : \texttt{end}\ \}$$

- The session type of the client's channel endpoint:

$$C \triangleq \oplus\{\ add : !\texttt{Int}.!\texttt{Int}.?\texttt{Int}.C, \\ neg : !\texttt{Int}.?\texttt{Int}.C \\ quit : \texttt{end}\ \}$$

Duality: $S = \overline{C}$

The Maths Server and Client: Types / Protocols

Legend

- &: branch/offer/external choice;
- $\oplus$: select/internal choice;
- ?`Int`.T: input `Int`, continue as T;
- !`Int`.T: output `Int`, continue as T;
- "." indicates sequencing;
- *add, neg, quit*: choice labels, all different;
- `end` marks the end of the protocol.

The Maths Server: Program and Type

A server *srv*, parametrised in its channel endpoint x of type S:

$$
\begin{aligned}
srv(x : S) \quad = x \triangleright \{ \; & add : x?(a : \mathtt{Int}).x?(b : \mathtt{Int}).x!\langle a + b\rangle.srv(x), \\
& neg : x?(a : \mathtt{Int}).x!\langle -a\rangle.srv(x) \\
& quit : \mathbf{0} \; \}
\end{aligned}
$$

$$
\begin{aligned}
S \quad = \& \; \{ \; & add : ?\mathtt{Int}.?\mathtt{Int}.!\mathtt{Int}.S, \\
& neg : ?\mathtt{Int}.!\mathtt{Int}.S \\
& quit : \mathsf{end} \; \}
\end{aligned}
$$

The Maths Client: Program and Type

A client *clt*, parametrised in its channel endpoint x of type C, assuming $P(a)$ does not use x:

$$clt(x : C) \quad = x \triangleleft neg.x!\langle 2\rangle.x?(a : \mathtt{Int}).x \triangleleft quit.P(a)$$

$$\begin{aligned} C \quad = \oplus\{\, & add : !\mathtt{Int}.!\mathtt{Int}.?\mathtt{Int}.C, \\ & neg : !\mathtt{Int}.?\mathtt{Int}.C \\ & quit : \mathtt{end} \,\} \end{aligned}$$

Client/Server Interaction (π- calculus OS)

$$(\nu c : S)(srv(c^+) \mid clt(c^-))$$

Client/Server Interaction (π- calculus OS)

$$(\nu c : S)(srv(c^+) \mid clt(c^-))$$
$$\downarrow$$
$$(\nu c : ?\mathtt{Int}.!\mathtt{Int}.S)(c^+?(a : \mathtt{Int}).c^+!\langle -a\rangle.srv(c^+) \mid c^-!\langle 2\rangle.c^-?(a : \mathtt{Int}).c^- \triangleleft quit.P(a))$$

Client/Server Interaction (π- calculus OS)

$$(\nu c : S)(srv(c^+) \mid clt(c^-))$$
$$\downarrow$$
$$(\nu c : ?\mathtt{Int}.!\mathtt{Int}.S)(c^+?(a : \mathtt{Int}).c^+!\langle -a \rangle.srv(c^+) \mid c^-!\langle 2 \rangle.c^-?(a : \mathtt{Int}).c^- \triangleleft quit.P(a))$$
$$\downarrow$$
$$(\nu c : !\mathtt{Int}.S)(c^+!\langle -2 \rangle.srv(c^+) \mid c^-?(a : \mathtt{Int}).c^- \triangleleft quit.P(a))$$

Client/Server Interaction (π-calculus OS)

$$
\begin{array}{c}
(\nu c : S)(\mathit{srv}(c^+) \mid \mathit{clt}(c^-)) \\
\downarrow \\
(\nu c : ?\mathtt{Int}.!\mathtt{Int}.S)(c^+?(a : \mathtt{Int}).c^+!\langle -a \rangle.\mathit{srv}(c^+) \mid c^-!\langle 2 \rangle.c^-?(a : \mathtt{Int}).c^- \triangleleft \mathit{quit}.P(a)) \\
\downarrow \\
(\nu c : !\mathtt{Int}.S)(c^+!\langle -2 \rangle.\mathit{srv}(c^+) \mid c^-?(a : \mathtt{Int}).c^- \triangleleft \mathit{quit}.P(a)) \\
\downarrow \\
(\nu c : S)(\mathit{srv}(c^+) \mid c^- \triangleleft \mathit{quit}.P(-2))
\end{array}
$$

Client/Server Interaction (π- calculus OS)

$$
\begin{array}{c}
(\nu c : S)(\mathit{srv}(c^+) \mid \mathit{clt}(c^-)) \\
\downarrow \\
(\nu c : \texttt{?Int.!Int.}S)(c^+?(a : \texttt{Int}).c^+!\langle -a\rangle.\mathit{srv}(c^+) \mid c^-!\langle 2\rangle.c^-?(a : \texttt{Int}).c^- \triangleleft \mathit{quit}.P(a)) \\
\downarrow \\
(\nu c : \texttt{!Int.}S)(c^+!\langle -2\rangle.\mathit{srv}(c^+) \mid c^-?(a : \texttt{Int}).c^- \triangleleft \mathit{quit}.P(a)) \\
\downarrow \\
(\nu c : S)(\mathit{srv}(c^+) \mid c^- \triangleleft \mathit{quit}.P(-2)) \\
\downarrow \\
(\nu c : \texttt{end})(\mathbf{0} \mid P(-2))
\end{array}
$$

Client/Server Interaction (π- calculus OS)

$$
\begin{array}{c}
(\nu c : S)(srv(c^+) \mid clt(c^-)) \\
\downarrow \\
(\nu c : ?\mathtt{Int}.!\mathtt{Int}.S)(c^+?(a : \mathtt{Int}).c^+!\langle -a \rangle.srv(c^+) \mid c^-!\langle 2 \rangle.c^-?(a : \mathtt{Int}).c^- \triangleleft quit.P(a)) \\
\downarrow \\
(\nu c : !\mathtt{Int}.S)(c^+!\langle -2 \rangle.srv(c^+) \mid c^-?(a : \mathtt{Int}).c^- \triangleleft quit.P(a)) \\
\downarrow \\
(\nu c : S)(srv(c^+) \mid c^- \triangleleft quit.P(-2)) \\
\downarrow \\
(\nu c : \mathtt{end})(\mathbf{0} \mid P(-2)) \\
\equiv \\
P(-2)
\end{array}
$$

Establishing a Connection

- The server listens on a standard channel a of type $\sharp S$, and receives a session channel for srv to use.

$$server(a) \quad = \quad a?(x : S).srv(x)$$

Establishing a Connection

- The server listens on a standard channel a of type $\sharp S$, and receives a session channel for *srv* to use.

$$server(a) \quad = \quad a?(x : S).srv(x)$$

- The global declaration $a : \sharp S$ advertises the server and its protocol.

Establishing a Connection

- The server listens on a standard channel a of type $\sharp S$, and receives a session channel for *srv* to use.

$$\mathit{server}(a) \quad = \quad a?(x : S).\mathit{srv}(x)$$

- The global declaration $a : \sharp S$ advertises the server and its protocol.
- The client creates a session channel and sends it to the server.

$$\mathit{client}(a) \quad = \quad (\nu c : S)(a!\langle c^{+} \rangle.\mathit{clt}(c^{-}))$$

Establishing a Connection

- The server listens on a standard channel a of type $\sharp S$, and receives a session channel for *srv* to use.

$$\mathit{server}(a) \quad = \quad a?(x : S).\mathit{srv}(x)$$

- The global declaration $a : \sharp S$ advertises the server and its protocol.
- The client creates a session channel and sends it to the server.

$$\mathit{client}(a) \quad = \quad (\nu c : S)(a!\langle c^+ \rangle.\mathit{clt}(c^-))$$

- After one step, execution proceeds as before.

Outline

Session Types: Key Features

- Duality: the relationship between the types of opposite endpoints of a session channel.

Session Types: Key Features

- Duality: the relationship between the types of opposite endpoints of a session channel.
- Linearity: each channel endpoint occurs exactly once in a collection of parallel processes.

Session Types: Key Features

- Duality: the relationship between the types of opposite endpoints of a session channel.
- Linearity: each channel endpoint occurs exactly once in a collection of parallel processes.
- The structure of session types matches the structure of communication.

Session Types: Key Features

- Duality: the relationship between the types of opposite endpoints of a session channel.
- Linearity: each channel endpoint occurs exactly once in a collection of parallel processes.
- The structure of session types matches the structure of communication.
- Session types change as communication occurs.

Session Types: Key Features

- Duality: the relationship between the types of opposite endpoints of a session channel.
- Linearity: each channel endpoint occurs exactly once in a collection of parallel processes.
- The structure of session types matches the structure of communication.
- Session types change as communication occurs.
- Connection is established among participants.

Properties of Session Types

- Communication Safety: the exchanged data has the expected type.

Properties of Session Types

- Communication Safety: the exchanged data has the expected type.
- Session Fidelity: the session channel has the expected structure.

Properties of Session Types

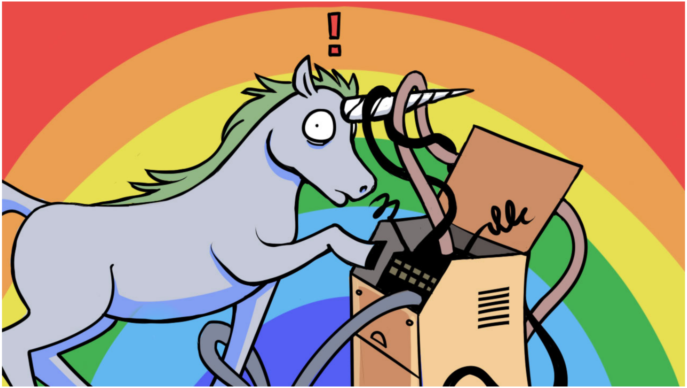
- Communication Safety: the exchanged data has the expected type.
- Session Fidelity: the session channel has the expected structure.
- Privacy: the session channel is owned only by the communicating parties.

Properties of Session Types

- Communication Safety: the exchanged data has the expected type.
- Session Fidelity: the session channel has the expected structure.
- Privacy: the session channel is owned only by the communicating parties.

Main Theorem: at runtime, communication follows the protocol.

The Calculus and Typing Rules



The Calculus: Types

S	$::=$	$\texttt{end}$	termination
		$!T.S$	send
		$?T.S$	receive
		$\oplus\{l_i : S_i\}_{i \in I}$	select
		$\&\{l_i : S_i\}_{i \in I}$	branch
T	$::=$	S	session type
		$\texttt{Bool}$	boolean type
		$\sharp T$	standard channel type
		$\ldots$	other type constructs

The Calculus: Terms

P, Q	$::=$	$\mathbf{0}$	inaction
		$P \mid Q$	composition
		$(\boldsymbol{\nu} x)P$	restriction
		$x^{p}!\langle v^{q}\rangle.P$	output
		$x^{p}?(y).P$	input
		$x^{p} \triangleleft l_j.P$	selection
		$x^{p} \triangleright \{l_i : P_i\}_{i\in I}$	branching
v	$::=$	x, y	channel
		$\texttt{true} \mid \texttt{false}$	boolean values
		$\ldots$	other values

$p, q \in \{+, -, \epsilon\}$ are **optional** *polarities* for channels.

Typing Rules

(T-Par)
$$\frac{\Gamma_1 \vdash P \quad \Gamma_2 \vdash Q}{\Gamma_1 + \Gamma_2 \vdash P \mid Q}$$

(T-Res)
$$\frac{\Gamma, x^p : S, x^{\overline{p}} : \overline{S} \vdash P \quad p, q \in \{+, -\}}{\Gamma \vdash (\nu x)P}$$

(T-In)
$$\frac{\Gamma, x^p : S, y : T \vdash P}{\Gamma, x^p : ?T.S \vdash x^p?(y).P}$$

(T-Out)
$$\frac{\Gamma_1, x^p : S \vdash P \quad \Gamma_2 \vdash v^q : T}{(\Gamma_1, x^p : !T.S) + \Gamma_2 \vdash x^p!\langle v^q \rangle.P}$$

(T-Brch)
$$\frac{\Gamma, x^p : S_i \vdash P_i \quad \forall i \in I}{\Gamma, x^p : \&\{l_i : S_i\}_{i \in I} \vdash x^p \triangleright \{l_i : P_i\}_{i \in I}}$$

(T-Sel)
$$\frac{\Gamma, x^p : S_j \vdash P \quad j \in I}{\Gamma, x^p : \oplus\{l_i : S_i\}_{i \in I} \vdash x^p \triangleleft l_j.P}$$

Gay & Hole, "*Subtyping for Session Types in the Pi Calculus*". ESOP 1999, Acta Informatica 2005.

Combination of Typing Contexts

$$
\begin{aligned}
\Gamma + x^+ : S &= \Gamma, x^+ : S && \text{if } x, x^+ \notin \mathrm{dom}(\Gamma) \\
\Gamma + x^- : S &= \Gamma, x^- : S && \text{if } x, x^- \notin \mathrm{dom}(\Gamma) \\
\Gamma + x : T &= \Gamma, x : T && \text{if } x, x^+, x^- \notin \mathrm{dom}(\Gamma) \\
(\Gamma, x : T) + x : T &= \Gamma, x : T && \text{if T is not a session type}
\end{aligned}
$$

Exercise: Is it well typed?

$$(\boldsymbol{\nu} x)(x^{+}?(t : \texttt{Bool}).\mathbf{0} \mid x^{-}!\langle \texttt{true} \rangle.\mathbf{0})$$

Exercise: Is it well typed?

$$(\nu x)(x^{+}?(t : \texttt{Bool}).\mathbf{0} \mid x^{-}!\langle\texttt{true}\rangle.\mathbf{0})$$

✓

Exercise: Is it well typed?

$$(\nu x)(x^{+}?(t : \texttt{Bool}).\mathbf{0} \mid x^{-}!\langle \texttt{true} \rangle.\mathbf{0})$$ ✓

$$(\nu x)(x^{+}!\langle t \rangle.\mathbf{0} \mid x^{-}!\langle \texttt{true} \rangle.\mathbf{0})$$

Exercise: Is it well typed?

$$(\nu x)(x^+?(t : \texttt{Bool}).\mathbf{0} \mid x^-!\langle \texttt{true} \rangle.\mathbf{0})$$ ✓

$$(\nu x)(x^+!\langle t \rangle.\mathbf{0} \mid x^-!\langle \texttt{true} \rangle.\mathbf{0})$$ ✗

Exercise: Is it well typed?

$$(\boldsymbol{\nu} x)(x^{+}?(t : \texttt{Bool}).\mathbf{0} \mid x^{-}!\langle\texttt{true}\rangle.\mathbf{0})$$ ✓

$$(\boldsymbol{\nu} x)(x^{+}!\langle t\rangle.\mathbf{0} \mid x^{-}!\langle\texttt{true}\rangle.\mathbf{0})$$ ✗

$$(\boldsymbol{\nu} x)(x^{-}!\langle\texttt{false}\rangle.\mathbf{0} \mid x^{+}?(t : \texttt{Bool}).\mathbf{0} \mid x^{+}?(w : \texttt{Bool}).\mathbf{0})$$

Exercise: Is it well typed?

$$(\nu x)(x^{+}?(t : \texttt{Bool}).\mathbf{0} \mid x^{-}!\langle \texttt{true} \rangle.\mathbf{0}) \quad ✓$$

$$(\nu x)(x^{+}!\langle t \rangle.\mathbf{0} \mid x^{-}!\langle \texttt{true} \rangle.\mathbf{0}) \quad ✗$$

$$(\nu x)(x^{-}!\langle \texttt{false} \rangle.\mathbf{0} \mid x^{+}?(t : \texttt{Bool}).\mathbf{0} \mid x^{+}?(w : \texttt{Bool}).\mathbf{0}) \quad ✗$$

Exercise: Is it well typed?

$$(\boldsymbol{\nu} x)(x^{+}?(t : \texttt{Bool}).\mathbf{0} \mid x^{-}!\langle \texttt{true} \rangle.\mathbf{0})$$ ✓

$$(\boldsymbol{\nu} x)(x^{+}!\langle t \rangle.\mathbf{0} \mid x^{-}!\langle \texttt{true} \rangle.\mathbf{0})$$ ✗

$$(\boldsymbol{\nu} x)(x^{-}!\langle \texttt{false} \rangle.\mathbf{0} \mid x^{+}?(t : \texttt{Bool}).\mathbf{0} \mid x^{+}?(w : \texttt{Bool}).\mathbf{0})$$ ✗

$$(\boldsymbol{\nu} x)(x^{-} \triangleleft k.\mathbf{0} \mid x^{+} \triangleright \{l_i : P_i\}_{i \in I}.\mathbf{0})$$

Exercise: Is it well typed?

$$(\nu x)(x^{+}?(t : \texttt{Bool}).\mathbf{0} \mid x^{-}!\langle \texttt{true} \rangle.\mathbf{0})$$ ✓

$$(\nu x)(x^{+}!\langle t \rangle.\mathbf{0} \mid x^{-}!\langle \texttt{true} \rangle.\mathbf{0})$$ ✗

$$(\nu x)(x^{-}!\langle \texttt{false} \rangle.\mathbf{0} \mid x^{+}?(t : \texttt{Bool}).\mathbf{0} \mid x^{+}?(w : \texttt{Bool}).\mathbf{0})$$ ✗

$$(\nu x)(x^{-} \triangleleft k.\mathbf{0} \mid x^{+} \triangleright \{l_i : P_i\}_{i \in I}.\mathbf{0})$$ ✗

Exercise: Is it well typed?

$$(\boldsymbol{\nu} x)(x^{+}?(t : \texttt{Bool}).\mathbf{0} \mid x^{-}!\langle \texttt{true} \rangle.\mathbf{0})$$ ✓

$$(\boldsymbol{\nu} x)(x^{+}!\langle t \rangle.\mathbf{0} \mid x^{-}!\langle \texttt{true} \rangle.\mathbf{0})$$ ✗

$$(\boldsymbol{\nu} x)(x^{-}!\langle \texttt{false} \rangle.\mathbf{0} \mid x^{+}?(t : \texttt{Bool}).\mathbf{0} \mid x^{+}?(w : \texttt{Bool}).\mathbf{0})$$ ✗

$$(\boldsymbol{\nu} x)(x^{-} \triangleleft k.\mathbf{0} \mid x^{+} \triangleright \{l_i : P_i\}_{i \in I}.\mathbf{0})$$ ✗

$$(\boldsymbol{\nu} x)(\boldsymbol{\nu} y)(x^{+}?(z : \texttt{Int}).y^{-}!\langle 42 \rangle.\mathbf{0} \mid x^{-}!\langle 11 \rangle.y^{+}?(w : \texttt{Int}).\mathbf{0})$$

Exercise: Is it well typed?

$$(\nu x)(x^+?(t : \texttt{Bool}).\mathbf{0} \mid x^-!\langle \texttt{true} \rangle.\mathbf{0})$$ ✓

$$(\nu x)(x^+!\langle t \rangle.\mathbf{0} \mid x^-!\langle \texttt{true} \rangle.\mathbf{0})$$ ✗

$$(\nu x)(x^-!\langle \texttt{false} \rangle.\mathbf{0} \mid x^+?(t : \texttt{Bool}).\mathbf{0} \mid x^+?(w : \texttt{Bool}).\mathbf{0})$$ ✗

$$(\nu x)(x^- \triangleleft k.\mathbf{0} \mid x^+ \triangleright \{l_i : P_i\}_{i \in I}.\mathbf{0})$$ ✗

$$(\nu x)(\nu y)(x^+?(z : \texttt{Int}).y^-!\langle 42 \rangle.\mathbf{0} \mid x^-!\langle 11 \rangle.y^+?(w : \texttt{Int}).\mathbf{0})$$ ✓

Exercise: Is it well typed?

$$(\boldsymbol{\nu} x)(x^{+}?(t : \texttt{Bool}).\mathbf{0} \mid x^{-}!\langle \texttt{true} \rangle.\mathbf{0})$$ ✓

$$(\boldsymbol{\nu} x)(x^{+}!\langle t \rangle.\mathbf{0} \mid x^{-}!\langle \texttt{true} \rangle.\mathbf{0})$$ ✗

$$(\boldsymbol{\nu} x)(x^{-}!\langle \texttt{false} \rangle.\mathbf{0} \mid x^{+}?(t : \texttt{Bool}).\mathbf{0} \mid x^{+}?(w : \texttt{Bool}).\mathbf{0})$$ ✗

$$(\boldsymbol{\nu} x)(x^{-} \triangleleft k.\mathbf{0} \mid x^{+} \triangleright \{l_i : P_i\}_{i \in I}.\mathbf{0})$$ ✗

$$(\boldsymbol{\nu} x)(\boldsymbol{\nu} y)(x^{+}?(z : \texttt{Int}).y^{-}!\langle 42 \rangle.\mathbf{0} \mid x^{-}!\langle 11 \rangle.y^{+}?(w : \texttt{Int}).\mathbf{0})$$ ✓

$$(\boldsymbol{\nu} x)(\boldsymbol{\nu} y)(y^{-}!\langle 42 \rangle.x^{+}?(z : \texttt{Int}).\mathbf{0} \mid x^{-}!\langle 11 \rangle.y^{+}?(w : \texttt{Int}).\mathbf{0})$$

Exercise: Is it well typed?

$$(\nu x)(x^+?(t : \texttt{Bool}).\mathbf{0} \mid x^-!\langle\texttt{true}\rangle.\mathbf{0})$$ ✓

$$(\nu x)(x^+!\langle t\rangle.\mathbf{0} \mid x^-!\langle\texttt{true}\rangle.\mathbf{0})$$ ✗

$$(\nu x)(x^-!\langle\texttt{false}\rangle.\mathbf{0} \mid x^+?(t : \texttt{Bool}).\mathbf{0} \mid x^+?(w : \texttt{Bool}).\mathbf{0})$$ ✗

$$(\nu x)(x^- \triangleleft k.\mathbf{0} \mid x^+ \triangleright \{l_i : P_i\}_{i\in I}.\mathbf{0})$$ ✗

$$(\nu x)(\nu y)(x^+?(z : \texttt{Int}).y^-!\langle 42\rangle.\mathbf{0} \mid x^-!\langle 11\rangle.y^+?(w : \texttt{Int}).\mathbf{0})$$ ✓

$$(\nu x)(\nu y)(y^-!\langle 42\rangle.x^+?(z : \texttt{Int}).\mathbf{0} \mid x^-!\langle 11\rangle.y^+?(w : \texttt{Int}).\mathbf{0})$$ ✓

Progress, Deadlock Freedom and Lock Freedom

Comparing Liveness Properties of Communication

- Deadlock Freedom: communications will eventually succeed, *unless* the whole process diverges. (Standard π)
- Lock Freedom: communications will eventually succeed, *even if* the whole process diverges. (Standard π)
- Progress: In-session communications will eventually succeed, provided that a suitable context can be found. (Session π)

Comparing Liveness Properties of Communication

- Deadlock Freedom: communications will eventually succeed, *unless* the whole process diverges. (Standard π)
- Lock Freedom: communications will eventually succeed, *even if* the whole process diverges. (Standard π)
- Progress: In-session communications will eventually succeed, provided that a suitable context can be found. (Session π)

Note: the type system by Gay & Hole does not satisfy the *liveness* properties, i.e., does not guarantee progress, deadlock freedom or lock freedom.

Deadlock Freedom vs. Lock Freedom

- Consider again the process from the exercise slide:

  $$P = (\boldsymbol{\nu} x)(\boldsymbol{\nu} y)\big(y^{-}!\langle 42\rangle.x^{+}?(z : \texttt{Int}).\mathbf{0} \mid x^{-}!\langle 11\rangle.y^{+}?(w : \texttt{Int}).\mathbf{0}\big)$$

  It is deadlocked *and* hence locked!

Deadlock Freedom vs. Lock Freedom

- Consider again the process from the exercise slide:

  $P = (\boldsymbol{\nu} x)(\boldsymbol{\nu} y)\big(y^{-}!\langle 42\rangle.x^{+}?(z : \texttt{Int}).\mathbf{0} \mid x^{-}!\langle 11\rangle.y^{+}?(w : \texttt{Int}).\mathbf{0}\big)$

  It is deadlocked *and* hence locked!

- Consider the process:

  $$Q = (\boldsymbol{\nu} x)(x^{+}?(z) \mid \Omega)$$

  It is deadlock-free *but* locked!

Research Question

What is the relationship among deadlock freedom, lock freedom and progress?

Research Question

What is the relationship among deadlock freedom, lock freedom and progress?

- Lock freedom is a **stronger** property than deadlock freedom.
- Progress is a **compositional** form of lock freedom.

Carbone et al. (COORDINATION 2014)

Outline

Outline

Research Timeline

Milner, Parrow, Walker 1989/1992

- Honda 1993
  - ↓ Takeuchi, Honda, Kubo 1994
  - ↓ Honda, Vasconcelos, Kubo 1998
  - ↓ To be continued...
- Milner 1993
  - ↓ Pierce, Sangiorgi 1993
  - ↓ Kobayashi, Pierce, Turner 1996
  - ↓ Sangiorgi 1998

On standard types for π- calculus

- $\sharp T$: channel used in input/output to transmit data of type T.

On standard types for π- calculus

- $\sharp T$: channel used in input/output to transmit data of type T.
- iT/oT: channel used *only* in input/output to transmit data of type T. [Pierce,Sangiorgi'93]

On standard types for π- calculus

- $\sharp T$: channel used in input/output to transmit data of type T.
- iT/oT: channel used *only* in input/output to transmit data of type T. [Pierce,Sangiorgi'93]
- $\ell_i T/\ell_o T$: channel used *only* in input/output and *exactly once* to transmit data of type T. [Kobayashi,Pierce,Turner'96]

On standard types for π- calculus

- $\sharp T$: channel used in input/output to transmit data of type T.
- iT/oT: channel used *only* in input/output to transmit data of type T. [Pierce,Sangiorgi'93]
- $\ell_i T/\ell_o T$: channel used *only* in input/output and *exactly once* to transmit data of type T. [Kobayashi,Pierce,Turner'96]
- $\langle l_i : T_i \rangle_{i \in I}$: labelled disjoint union of types. [Sangiorgi'98]

Key words for standard π- types

For session-typed π- calculus:

1. Structure
2. Duality
3. Restriction
4. Branch/Select

Key words for standard π- types

For session-typed π- calculus:

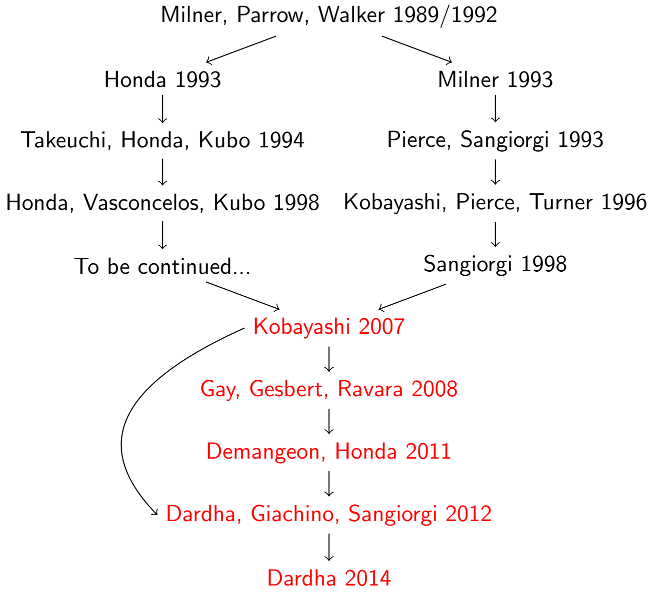
1. Structure
2. Duality
3. Restriction
4. Branch/Select

1. Linearity forces a π channel to be used exactly once.
2. Capability of input/output of the same π channel split between two partners.
3. Restriction construct permits the creation of fresh private π channels.
4. Variant type permits choice.

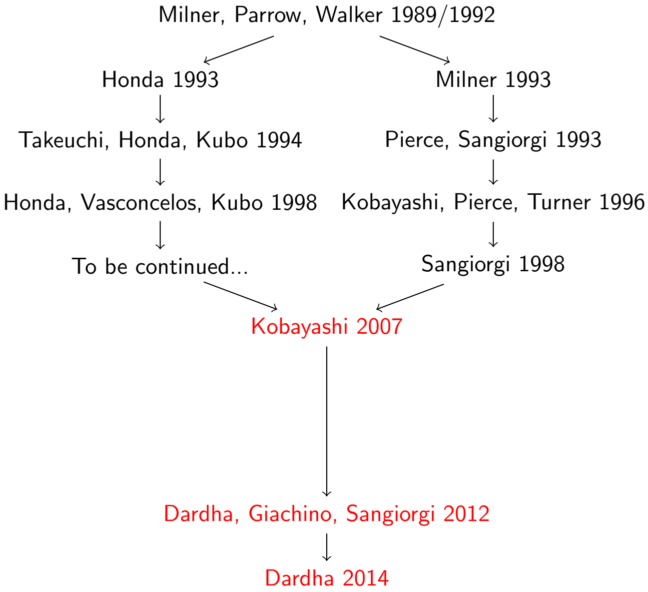
Bridging the two worlds

To which extent session constructs are more complex and more expressive than the standard π- calculus constructs?

Research Timeline



Research Timeline



Milner, Parrow and Walker 1989/1992
A calculus of mobile processes

Honda 1993
Types for dyadic interaction

Takeuchi, Honda and Kubo 1994
An interaction based language and its typing system

Honda, Vasconcelos and Kubo 1998
Language primitives and type discipline for structured communication-based programming

To be continued...

Milner 1993
The polyadic pi-calculus: a tutorial

Pierce and Sangiorgi 1993
Typing and subtyping for mobile processes

Kobayashi, Pierce and Turner 1996
Linearity and the pi-calculus

Sangiorgi 1998
An interpretation of typed objects into typed pi-calculus

Kobayashi 2007
Type systems for concurrent programs

Gay, Gesbert and Ravara 2008
Session types as generic process types

Demangeon and Honda 2011
Full abstraction in a subtyped pi-calculus with linear types

Dardha, Giachino and Sangiorgi 2012
Session types revisited

Dardha 2014
Recursive session types revisited

Key idea of the encoding

Encoding is based on:

1. Linearity of π- calculus channel types;
2. Input/Output channel capabilities;
3. Continuation-Passing principle.
4. Variant types for the π- calculus.

Intuition of the encoding

- Session types are encoded as linear channel types.
- ? and ! are encoded as ℓ_i and ℓ_o.
- $\&\{l_i : S_i\}_{i \in I}$ and $\oplus\{l_i : S_i\}_{i \in I}$ are encoded using *variant types*.
- Continuation of a session type becomes carried type.
- Dual operations in continuation become equal when carried.

Why is this interesting?

Benefits of the encoding:

1. Large reusability of standard typed π-calculus theory.
2. Derivation of properties for session π-calculus from the standard typed π-calculus. (e.g. SR, TS)
3. Elimination of redundancy in the syntax of types and terms and in the theory.
4. Encoding is robust (subtyping, polymorphism, higher-order).
5. Expressivity result for session types.
6. Most importantly, implementation of session types in mainstream programming languages (cf. **lchannels** for Scala, **FuSe** for Ocaml, later on...)

Encoding Finite Session Types: Example

Let

$$S = ?\mathtt{Int}.?\mathtt{Int}.!\mathtt{Bool}.\mathtt{end}$$

Then

$$[\![S]\!] = \ell_i[\mathtt{Int}, \ell_i[\mathtt{Int}, \ell_o[\mathtt{Bool}, \emptyset[]]]]$$

Encoding Finite Session Types: Example

Let

$$S = ?\mathtt{Int}.?\mathtt{Int}.!\mathtt{Bool}.\mathtt{end}$$

Then

$$[\![S]\!] = \ell_i[\mathtt{Int}, \ell_i[\mathtt{Int}, \ell_o[\mathtt{Bool}, \emptyset[]]]]$$

Encoding Finite Session Types: Example

Let

$$S = ?\mathtt{Int}.?\mathtt{Int}.!\mathtt{Bool}.\mathtt{end}$$

Then

$$[\![S]\!] = \ell_i[\mathtt{Int}, \ell_i[\mathtt{Int}, \ell_o[\mathtt{Bool}, \emptyset[]]]]$$

Encoding Finite Session Types: Example

Let

$$S = ?\texttt{Int}.?\texttt{Int}.!\texttt{Bool}.\texttt{end}$$

Then

$$[\![S]\!] = \ell_i[\texttt{Int}, \ell_i[\texttt{Int}, \ell_o[\texttt{Bool}, \emptyset[]]]]$$

Encoding Finite Session Types: Example

Let

$$S = ?\texttt{Int}.?\texttt{Int}.!\texttt{Bool}.\texttt{end}$$

Then

$$[\![S]\!] = \ell_i[\texttt{Int}, \ell_i[\texttt{Int}, \ell_o[\texttt{Bool}, \emptyset[]]]]$$

Encoding Finite Session Types: Example

Let

$$\overline{S} = \texttt{!Int.!Int.?Bool.end}$$

Then

$$[\![\overline{S}]\!] = \ell_o[\texttt{Int}, \ell_i[\texttt{Int}, \ell_o[\texttt{Bool}, \emptyset[]]]]$$

Remark

The encoding of dual types is as follows:

$$[\![S]\!] = \ell_i[\texttt{Int}, \ell_i[\texttt{Int}, \ell_o[\texttt{Bool}, \emptyset[]]]]$$

and

$$[\![\overline{S}]\!] = \ell_o[\texttt{Int}, \ell_i[\texttt{Int}, \ell_o[\texttt{Bool}, \emptyset[]]]]$$

Remark

The encoding of dual types is as follows:

$$[\![S]\!] = \ell_i[\texttt{Int}, \ell_i[\texttt{Int}, \ell_o[\texttt{Bool}, \emptyset[]]]]$$

and

$$[\![\overline{S}]\!] = \ell_o[\texttt{Int}, \ell_i[\texttt{Int}, \ell_o[\texttt{Bool}, \emptyset[]]]]$$

Remark

duality on session types boils down to opposite capabilities (i/o) of channel types, only in the outermost level!

Encoding of Session Types: Formally

$$
\begin{aligned}
[\![\mathtt{end}]\!] &\triangleq \emptyset[] \\
[\![!T.S]\!] &\triangleq \ell_o[[\![T]\!], [\![\overline{S}]\!]] \\
[\![?T.S]\!] &\triangleq \ell_i[[\![T]\!], [\![S]\!]] \\
[\![\oplus\{l_i : S_i\}_{i\in I}]\!] &\triangleq \ell_o[\langle l_i : [\![\overline{S_i}]\!]\rangle_{i\in I}] \\
[\![\&\{l_i : S_i\}_{i\in I}]\!] &\triangleq \ell_i[\langle l_i : [\![S_i]\!]\rangle_{i\in I}]
\end{aligned}
$$

Properties of the Encoding

Theorem

Encoding preserves typability of programs.

Theorem

Encoding preserves evaluation of programs.

Lemma

Encoding of dual session types gives dual linear π-types.

Outline

Curry-Howard Correspondences

propositions	as	types
proofs	as	programs
proof normalisation	as	program evaluation

Intuitionistic Natural Deduction	$\leftrightarrow$	Simply-Typed Lambda Calculus
Quantification over propositions	$\leftrightarrow$	Polymorphism
Modal Logical	$\leftrightarrow$	Monads (state, exceptions)

Curry-Howard Correspondences

propositions	as	types
proofs	as	programs
proof normalisation	as	program evaluation
Intuitionistic Natural Deduction	$\leftrightarrow$	Simply-Typed Lambda Calculus
Quantification over propositions	$\leftrightarrow$	Polymorphism
Modal Logical	$\leftrightarrow$	Monads (state, exceptions)
???	$\leftrightarrow$	Process Calculus

What is the Curry-Howard correspondence for concurrency?

- Since the beginning of linear logic (Girard 1987), there were suggestions that it should be relevant to concurrency.

What is the Curry-Howard correspondence for concurrency?

- Since the beginning of linear logic (Girard 1987), there were suggestions that it should be relevant to concurrency.

"*The new connectives of linear logic have obvious meanings in terms of parallel computation. [...] Linear logic is the first attempt to solve the problem of parallelism at the logical level, i.e., by making the success of the communication process only dependent of the fact that the programs can be viewed as proofs of something, and are therefore sound.*"

— Girard 1987

π- Calculus and Linear Logic

- Abramsky (1994); Bellin & Scott (1994) established a correspondence between linear logic and standard π- calculus.
- Caires & Pfenning (2010) established a correspondence between dual intuitionistic linear logic (DILL) and session typed π- calculus.
- Later on, Wadler (2012) established a correspondence between classical linear logic (CLL) and session typed π- calculus.
- The logical approach to session types has been extended: dependent types, failures, sharing and races...

Session Types and Linear Logic Correspondence

propositions	as	session types
proofs	as	π- processes
proof normalisation / cut elimination	as	communication

Session Types and Classical Linear Logic (1)

- $A \parr B$ is interpreted as "input A then behave like B" ($?A.B$)

Session Types and Classical Linear Logic (1)

- $A \parr B$ is interpreted as "input A then behave like B" $(?A.B)$
- $A \otimes B$ is interpreted as "output A then behave like B" $(!A.B)$

Session Types and Classical Linear Logic (1)

- $A \parr B$ is interpreted as "input A then behave like B" $(?A.B)$
- $A \otimes B$ is interpreted as "output A then behave like B" $(!A.B)$
- $\&$ and $\oplus$ are interpreted as branch and select.

Session Types and Classical Linear Logic (1)

- $A \parr B$ is interpreted as "input A then behave like B" ($?A.B$)
- $A \otimes B$ is interpreted as "output A then behave like B" ($!A.B$)
- $\&$ and $\oplus$ are interpreted as branch and select.
- The correspondence has led to a re-examination of all aspects of session types, from a logical viewpoint.

Session Types and Classical Linear Logic (2)

(T-⅋)
$$\frac{P \vdash \Delta, y : A, x : B}{x?(y).P \vdash \Delta, x : A ⅋ B}$$

(T-⊗)
$$\frac{P \vdash \Delta, y : A \qquad Q \vdash \Delta', x : B}{x!(y).(P \mid Q) \vdash \Delta, \Delta', x : A \otimes B}$$

(T-cut)
$$\frac{P \vdash \Delta, x : \overline{A} \qquad Q \vdash \Delta', x : A}{(\nu x)(P \mid Q) \vdash \Delta, \Delta'}$$

(T-&)
$$\frac{P_i \vdash \Delta, x{:}A_i \qquad \forall i \in I}{x \triangleright \{l_i : P_i\}_{i \in I} \vdash \Delta, x{:}\&\{l_i : A_i\}_{i \in I}}$$

(T-⊕)
$$\frac{P \vdash \Delta, x{:}A_j \qquad j \in I}{x \triangleleft l_j.P \vdash \Delta, x{:}\oplus\{l_i : A_i\}_{i \in I}}$$

Wadler 2012; Caires 2014 (@Luca Cardelli Fest)

Session Types and Classical Linear Logic (3)

The session type system based on (Classical) Linear Logic propositions guarantees:

- Type Preservation (or Subject Reduction): Well-typed processes reduce to well-typed processes.
- Deadlock-Freedom (by design): If process P is well typed and it is a *cut*, then there is some Q, such that P reduces to Q and Q is not a *cut*.

Outline

Session Types in Programming Languages: A Collection of Implementations[1]

Binary vs. Multiparty

Are sessions between two participants (generally implemented as a typed channel with dual endpoints), or multiple?

Primitive vs. Library vs. External

What form do the session types take?

- Primitive: Session types are implemented as language primitives, or as part of a compiler plugin
- Library: Session types are provided using a library
- External: A tool checking session types as a static analysis pass, or providing functionality that is not necessarily verifying conformance to a protocol.

Static vs. Dynamic vs. Hybrid Verification

When and how is conformance to the session types checked?

- Static: Conformance to session types is fully checked at compile time. Any error (be it sending the wrong message, not completing a session, or duplicating an endpoint) will be reported before a program compiles.
- Dynamic: Conformance to session types is checked at runtime. Session types are compiled into communicating finite-state machines, and messages are verified against these CFSMs. These approaches are very flexible, extending session types to dynamically-checked languages, and allowing things like assertions on data.
- Hybrid: Sending messages in the right order is checkeed statically. Linearity is checkeed dynamically. This is a promising approach, with drop-in libraries available to be used in general-purpose languages today!

[1] http://groups.inf.ed.ac.uk/abcd/session-implementations.html

Programming Languages with Primitive Binary Session Types: Static Typechecking

Sill:

- Functional programming language that supports session-typed message passing concurrency.
- Based on the Curry-Howard correspondence of session types and intuitionistic linear logic (Caires & Pfenning 2010).
- Type preservation; deadlock and race freedom; support of subtyping, polymorphism and recursive types.

Resources:

- *From Linear Logic to Session-Typed Concurrent Programming*, F.Pfenning.
- *Polarised Substructural Session Types*, F.Pfenning and D.Griffith. FoSSaCS 2015.

Programming Languages with Primitive Binary Session Types: Static Typechecking

SePi:

- Concurrent, message-passing programming language based on the π- calculus.
- Features synchronous, bidirectional channel-based communication.
- Primitives for send/receive as well as offer/select choices.

Resources:

- *A Concurrent Programming Language with Refined Session Types*, J.Franco and V.T.Vasconcelos. BEAT 2013.
- *Linearly Refined Session Types*, P.Baltazar, D.Mostrous, and V.T.Vasconcelos. LINEARITY 2012.
- *Fundamentals of Session Types*, V.T.Vasconcelos. Information and Computation, 2012.

Programming Languages with Primitive Binary Session Types: Static Typechecking

Links:

- Programming language for web applications.
- Binary session types added as language primitives and fully statically typechecked, using an extension of the type system to support linear types.

Resources:

- *Lightweight Functional Session Types*, S.Lindley and J.G.Morris. In Behavioural Types: from Theory to Tools.

Mainstream Programming Languages with Binary Session Types

Haskell:

- **effect-sessions**: implementation of session types in Concurrent Haskell, through the observation that session types can be encoded using an effect system (and vice versa). Orchard & Yoshida (POPL 2016)
- **simple-sessions**: a library implementation of Haskell session types, using parameterised monads and a channel stack Pucella & Tov (Haskell 2008)
- **sessions**: an alternative embedding of session types in Haskell. Sackman & Eisenbach (TR 2008)
- **GVinHS**: embedding session types in Haskell with first-class channels; builds on Polakow's embedding of a linear λ-calculus in Haskell. Lindley & Morris (Haskell 2016); Polakow (Haskell 2015).

Mainstream Programming Languages with Binary Session Types

Java:

- **CO2 Middleware**: for Java applications, based on timed session types; *dynamic monitoring* for conformance of timing constraints.
  Bartoletti et al. (FACS 2015, FORTE 2015)

- **(Eventful) Session Java**: a frontend and runtime library for Java, supporting binary session types, statically; the tool also supports event-driven programming.
  Hu, Yoshida & Honda (ECOOP 2008);
  Hu et al. (ECOOP 2010)

Mainstream Programming Languages with Binary Session Types

Scala

- **lchannels**: based on the continuation-passing encoding of session types into linear π- calculus types (Kobayashi 2007; Dardha et al. 2012)
- Message ordering is checked *statically*.
- Linearity is checked *dynamically*.
- Scalas & Yoshida (ECOOP 2016)

Mainstream Programming Languages with Binary Session Types

OCaml

- **FuSe**: lightweight implementation of BST in OCaml; based on the continuation-passing encoding of session types into linear π- calculus types (Kobayashi 2007; Dardha et al. 2012)
- *Static* check of message ordering and *dynamic* check of linearity. (Padovani 2015)

Rust:

- Implementation of BST in Mozilla's Rust; use of Rust's *affine* type system. Jespersen, Munksgaard & Larsen in WGP 2015.

Outline

Multiparty Session Types (1)

- Honda, Yoshida & Carbone (POPL 2008) developed a theory of multiparty session types.
  Awarded the ACM SIGPLAN *Most Influential POPL Paper Award* at POPL 2018.

Multiparty Session Types (1)

- Honda, Yoshida & Carbone (POPL 2008) developed a theory of multiparty session types.
  Awarded the ACM SIGPLAN *Most Influential POPL Paper Award* at POPL 2018.
- A global (session) type specifies a multi-party protocol.

Multiparty Session Types (1)

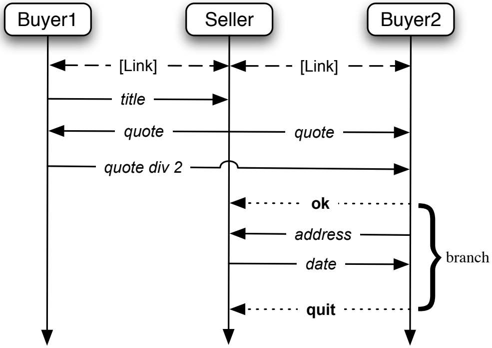
- Honda, Yoshida & Carbone (POPL 2008) developed a theory of multiparty session types.
  Awarded the ACM SIGPLAN *Most Influential POPL Paper Award* at POPL 2018.
- A global (session) type specifies a multi-party protocol.
- A global type can be validated and projected to local (session) types, which specify the communication behaviour of each participant.

Multiparty Session Types (1)

- Honda, Yoshida & Carbone (POPL 2008) developed a theory of multiparty session types.
  Awarded the ACM SIGPLAN *Most Influential POPL Paper Award* at POPL 2018.
- A global (session) type specifies a multi-party protocol.
- A global type can be validated and projected to local (session) types, which specify the communication behaviour of each participant.
- Local session type checking guarantees privacy, communication safety and session fidelity.

Multiparty Session Types (2)

A buyer-seller example from Honda et al (POPL 2018):



Multiparty Session Types (3)

The global type describes the whole protocol:

$$
\begin{array}{lll}
1. & B1 \to S : & \text{title}. \\
2. & S \to B1 : & \text{quote}. \\
3. & S \to B2 : & \text{quote}. \\
4. & B1 \to B2 : & \text{quote}. \\
5. & B2 \to S : & \left\{ \begin{array}{rl} \text{ok} : & \begin{array}{l} B2 \to S : \text{address}. \\ S \to B2 : \text{date.end}, \end{array} \\ \text{quit} : & \text{end} \end{array} \right\}
\end{array}
$$

Multiparty Session Types (4)

- Projection gives a local session type for each participant.
  For $B1$:

$$S!\mathrm{title}.S?\mathrm{quote}.B2!\mathrm{quote}$$

  and for $B2$:

$$S?\mathrm{quote}.B1?\mathrm{quote}.S \oplus \{\mathrm{ok} : S!\mathrm{address}.S?\mathrm{date}.\mathrm{end},\ \mathrm{quit} : \mathrm{end}\}$$

Multiparty Session Types (4)

- Projection gives a local session type for each participant. For $B1$:

$$S!\mathsf{title}.S?\mathsf{quote}.B2!\mathsf{quote}$$

and for $B2$:

$$S?\mathsf{quote}.B1?\mathsf{quote}.S \oplus \{\mathsf{ok} : S!\mathsf{address}.S?\mathsf{date}.\mathsf{end},\ \mathsf{quit} : \mathsf{end}\}$$

- Local session type checking is similar to binary session type checking.
- Consistency conditions on the global type guarantee that the protocol can be realised by independent local participants.

Outline

Outline

www.scribble.org

Scribble: Describing Multi Party Protocols

Scribble is a language to describe application-level protocols among communicating systems. A protocol represents an agreement on how participating systems interact with each other. Without a protocol, it is hard to do meaningful interaction: participants simply cannot communicate effectively, since they do not know when to expect the other parties to send data, or whether the other party is ready to receive data. However, having a description of a protocol has further benefits. It enables verification to ensure that the protocol can be implemented without resulting in unintended consequences, such as deadlocks.

Describe

Scribble is a language for describing multiparty protocols from a global, or endpoint neutral, perspective.

Verify

Scribble has a theoretical foundation, based on the Pi Calculus and Session Types, to ensure that protocols described using the language are sound, and do not suffer from deadlocks or livelocks.

Project

Endpoint projection is the term used for identifying the responsibility of a particular role (or endpoint) within a protocol.

Implement

Various options exist, including (a) using the endpoint projection for a role to generate a skeleton code, (b) using session type APIs to clearly describe the behaviour, and (c) statically verify the code against the projection.

Monitor

Use the endpoint projection for roles defined within a Scribble protocol, to monitor the activity of a particular endpoint, to ensure it correctly implements the expected behaviour.

"Scribbling is necessary for architects, either physical or computing, since all great ideas of architectural construction come from that unconscious moment, when you do not realise what it is, when there is no concrete shape, only a whisper which is not a whisper, an image which is not an image, somehow it starts to urge you in your mind, in so small a voice but how persistent it is, at that point you start scribbling." (Dr. Kohei Honda, 2007)

Scribble

- Scribble is a protocol specification language used to describe application-level protocols among communicating agents.
- It is based on multiparty session types theory @ POPL 2008.
- Allows:
  - **specification** of a protocol in the form of global session type;
  - **validation** of the protocol;
  - **projection** into the communicating participants, i.e., roles.
- Contributors: K.Honda, Imperial College team.

Scribble by example: The Bookstore Global Protocol

```
global protocol Bookstore(role Buyer1, role Buyer2,
    role Seller) {
  book(title) from Buyer1 to Seller;
  book(quote) from Seller to Buyer1, Buyer2;
  contribution(quote) from Buyer1 to Buyer2;
  choice at Buyer2 {
    ok from Buyer2 to Seller;
    deliver(address) from Buyer2 to Seller;
    deliver(date) from Seller to Buyer2;
   } or {
    quit from Buyer2 to Seller;
  }
}
```

The Bookstore Protocol: Buyer1

```
local protocol Bookstore_Buyer1(self Buyer1, role
    Buyer2, role Seller) {
  book(title) to Seller;
  book(quote) from Seller;
  contribution(quote) to Buyer2;
}
```

The Bookstore Protocol: Buyer2

```
local protocol Bookstore_Buyer2(role Seller, self
    Buyer2, role Buyer1) {
  book(quote) from Seller;
  contribution(quote) from Buyer1;
  choice at Buyer2{
    ok to Seller;
    deliver(address) to Seller;
    deliver(date) from Seller;
  } or {
    quit to Seller;
  }
}
```

Outline

www.dcs.gla.ac.uk/research/mungo/

Mungo

University of Glasgow

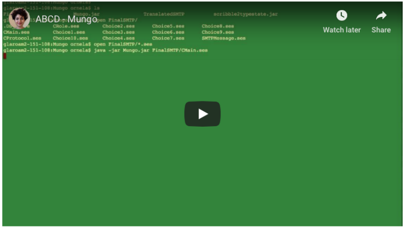
Home Tools Team Publications Contact

What is Mungo?

Mungo is a Java front-end tool that statically checks the order of method calls. It is based on the notion of *typestate* describing non-uniform objects, where the availability of methods to be called depends on the state of the object. Hence, the typestate defines a *protocol*.

A typestate file is defined and associated to a class. The Mungo tool checks that the object instantiating the class performs method calls by following its declared typestate. If the object respects the typestate, then .java files are produced, for every .mungo file in the program. Finally, the Java tools can be used to compile and run the standard Java code. If the typestate is violated, Mungo reports the errors.

The following video shows Mungo at work on SMTP.



Use case

The FileProtocol typestate describes a protocol on a file object. It imposes an order in which the methods of a file object should be called.

The first method available to be called on a file is open().

Depending on the return state of this method call, there is a transition either to a state where the file is open or to the end state, denoting the terminatation of the protocol.

```
typestate FileProtocol {

   Init =  {
      Status open(): <OK: Open, ERROR:end>
   }

   Open =  {
      Boolean hasNext(): <TRUE: Read, FALSE: Close>,
```

Mungo

- Mungo is a Java front-end tool that *statically* checks the order of method calls of an object.
- Based on the notions of session types and **typestate**, it describes *non-uniform* objects, where available methods change according to the state of the object.
- A Java class is annotated with a typestate, `@Typestate`. Mungo checks method calls follow the declared typestate of an object.
- **Resources**:
  Kouzapas et al. (PPDP 2016, Sci. Comp. Journal 2018)
  Based on Gay et al (POPL 2010).
  Developer: D. Kouzapas.

The FileProtocol Example

```
typestate FileProtocol {
  Init =          {
     Status open(): <OK: Open, ERROR: end>
  }
  Open =          {
     BooleanEnum hasNext(): <TRUE: Read, FALSE: Close>,
     void close(): end
  }
  Read =          {
     void read(): Open
  }
  Close =         {
     void close(): end
  }
}
```

Outline

StMungo

- StMungo (Scribble-to-Mungo) is a Java-based tool used to translate *Scribble local protocols* into *typestate specifications*.
- After the translation, Mungo is used to statically typecheck the typestate specification.
- **Resources**:
  Kouzapas et al. (PPDP 2016, Sci. Comp. Journal 2018)
  Developer: O. Dardha

The Buyer2 Local Protocol

```
local protocol Bookstore_Buyer2(role Seller, self
    Buyer2, role Buyer1) {
  book(quote) from Seller;
  contribution(quote) from Buyer1;
  choice at Buyer2{
    ok to Seller;
    deliver(address) to Seller;
    deliver(date) from Seller;
  } or {
    quit to Seller;
  }
}
```

The Buyer2 Local Protocol as Typestate

```
typestate Buyer2Protocol {
  State0 = {
    quote receive_quoteFromSeller(): State1
  }
  State1 = {
    quote receive_quoteFromBuyer1(): State2
  }
  State2 = {
    void send_OKToSeller(): State3,
    void send_QUITToSeller(): State5
  }
  State3 = {
     void send_addressToSeller(address): State4
  }
  State4 = {
    date receive_dateFromSeller(): end
  }
        ...
}
```

Outline

The SMTP Protocol: A Case Study

Mungo

University of Glasgow

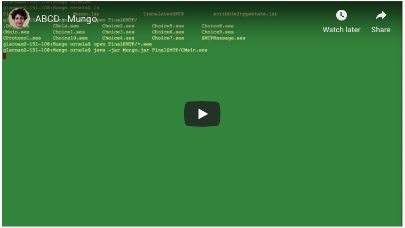
Home Tools Team Publications Contact

What is Mungo?

Mungo is a Java front-end tool that statically checks the order of method calls. It is based on the notion of *typestate* describing non-uniform objects, where the availability of methods to be called depends on the state of the object. Hence, the typestate defines a *protocol*.

A typestate file is defined and associated to a class. The Mungo tool checks that the object instantiating the class performs method calls by following its declared typestate. If the object respects the typestate, then .java files are produced, for every .mungo file in the program. Finally, the Java tools can be used to compile and run the standard Java code. If the typestate is violated, Mungo reports the errors.

The following video shows Mungo at work on SMTP.



Use case

The FileProtocol typestate describes a protocol on a file object. It imposes an order in which the methods of a file object should be called.

The first method available to be called on a file is open().

Depending on the return state of this method call, there is a transition either to a state where the file is open or to the end state, denoting the terminatation of the protocol.

```
typestate FileProtocol {

   Init =  {
      Status open(): <OK: Open, ERROR:end>
   }

   Open =  {
      Boolean hasNext(): <TRUE: Read, FALSE: Close>,
```

Mainstream Programming Languages with Multiparty Session Types

Multiparty Session C

- *Static* typechecking of MST in C programming language.
- Session communication happens by using a runtime library; type-checking is done via a plugin.
- Ng, Yoshida & Honda (TOOLS 2012); Ng et al. (HEART 2012)

Erlang

- A framework for monitoring Erlang applications by *dynamically* verifying communication against multiparty session types. Erlang actors can take part in multiple roles in multiple instances of multiple protocols. Fowler (ICE 2016)

Mainstream Programming Languages with Multiparty Session Types

Go (external tools)

- **DinGo Hunter**: a static analyser for Go programs, which can *statically* detect deadlocks. The tool works by extracting CFSMs from Go programs, and attempting to synthesise a global graph. Should this fail, then there is a deadlock. Ng & Yoshida (CC 2016)
- **Gong**: a static analyser for Go, building on a minimal core calculus for Go, called MiGo. MiGo types can be extracted from Go programs using another tool called GoInfer. Lange et al. (POPL 2017)

Mainstream Programming Languages with Multiparty Session Types

Python

- **SPY**: implementation of MST in Python using runtime monitoring. Neykova (PLACES 2013); Neykova, Yoshida & Hu (RV 2013); Hu et al (RV 2013)

- **Session Actor**: an implementation for combining session types and the actor model. Each actor may be involved in multiple roles, in multiple sessions. Communication is checked dynamically via compilation of Scribble protocols into CFSMs. Neykova & Yoshida (COORDINATION 2014)

Mainstream Programming Languages with Multiparty Session Types

Scala

- **Scribble-Scala**: Building upon **lchannels** and encoding of multiparty session types into linear types. Scalas et al. (ECOOP 2017, DARTS 2017)
- Order of messages is checked statically; linearity is checked dynamically as in **lchannels**
- Distributed multiparty session delegation is implemented here for the first time!

Outline

Conclusions

- Session types are a very simple but powerful formalism to model communication protocols in distributed systems.
- Developed for calculi as well as programming languages and various paradigms.
- Many interesting features.
- Part of behavioural types, including also contracts, typestates etc...

Acknowledgement

I am thankful to:

- Simon Gay
- Phil Wadler

for borrowing some of their slides.

$$
\begin{aligned}
&\mathsf{Audience}!\langle \mathsf{ThankYou}\rangle.\\
&\mathsf{rec}\ X\{\ \&\ \{\\
&\qquad\qquad \mathit{more} : \mathsf{Audience}?(y : \mathsf{Question}).\mathsf{Audience}!\langle \mathsf{Answer}\rangle.X,\\
&\qquad\qquad \mathit{quit} : \mathsf{end}\}\\
&\}
\end{aligned}
$$