Session Types

Logical Foundations of Concurrent Computation

Jorge A. Pérez
University of Groningen, The Netherlands
j.a.perez [[at]] rug.nl

Dutch Winter School 2026
(Part 1.1)

About Myself

- Associate Professor and Leader, Fundamental Computing Group
- Brief bio: PhD in 2010 (Bologna), then Postdoc in Lisbon (2010-2014). Joined Groningen in 2014. Currently main supervisor for two PhD students.
- **Research Interests**:
  Semantics of concurrency, process models, type systems, relative expressiveness

This Course: Session Types / Propositions as Sessions

An introduction to **session types** for concurrency, and to the logical foundations of message-passing computation.

This Course: Session Types / Propositions as Sessions

An introduction to **session types** for concurrency, and to the logical foundations of message-passing computation.

- In a nutshell, a **session** provides a structure for a series of related interactions between communicating programs.

This Course: Session Types / Propositions as Sessions

An introduction to **session types** for concurrency, and to the logical foundations of message-passing computation.

- In a nutshell, a **session** provides a structure for a series of related interactions between communicating programs.
- A disciplined usage of sessions is key to ensure program correctness.
  Not only: sessions have elegant **logical foundations**!

This Course: Session Types / Propositions as Sessions

An introduction to **session types** for concurrency, and to the logical foundations of message-passing computation.

- In a nutshell, a **session** provides a structure for a series of related interactions between communicating programs.
- A disciplined usage of sessions is key to ensure program correctness.
  Not only: sessions have elegant **logical foundations**!
- **Linear Logic**: A resource-oriented view on propositions / session protocols.
  Linear resources can be used exactly once.

This Course: Session Types / Propositions as Sessions

Plan:

1. Motivation, syntax of session types, Multiplicative Linear Logic (MLL)

This Course: Session Types / Propositions as Sessions

Plan:

1. Motivation, syntax of session types, Multiplicative Linear Logic (MLL)
2. The interpretation of MALL as concurrent processes
3. Cut-elimination and correctness for concurrent processes

This Course: Session Types / Propositions as Sessions

Plan:

1. Motivation, syntax of session types, Multiplicative Linear Logic (MLL)
2. The interpretation of MALL as concurrent processes
3. Cut-elimination and correctness for concurrent processes
4. Asynchronous communication and deadlock freedom

This Course: Session Types / Propositions as Sessions

Plan:

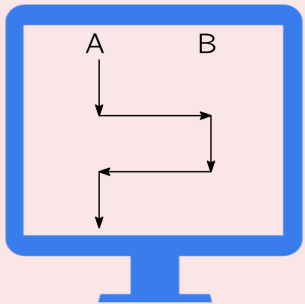
1. Motivation, syntax of session types, Multiplicative Linear Logic (MLL)
2. The interpretation of MALL as concurrent processes
3. Cut-elimination and correctness for concurrent processes
4. Asynchronous communication and deadlock freedom

Your questions and feedback are warmly welcome!

Outline

When is a Program Correct?

Sequential Programs



"Programs produce outputs that are consistent with their input"

Concurrent Programs?

Message-Passing Concurrent Programs?

- Software components (**services**) distributed across networks

Message-Passing Concurrent Programs

- Software components (**services**) distributed across networks
- Coordination takes place by sending and receiving messages

Message-Passing Concurrent Programs

- Software components (**services**) distributed across networks
- Coordination takes place by sending and receiving messages
- Compatible message exchanges are crucial for system correctness

Message-Passing Concurrent Programs

- Software components (**services**) distributed across networks
- Coordination takes place by sending and receiving messages
- Compatible message exchanges are crucial for system correctness
- Even a single faulty exchange can cause system-wide bugs

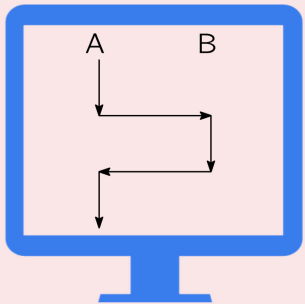
Message-Passing Concurrent Programs

- Software components (**services**) distributed across networks
- Coordination takes place by sending and receiving messages
- Compatible message exchanges are crucial for system correctness
- Even a single faulty exchange can cause system-wide bugs

An (imperfect) analogy:

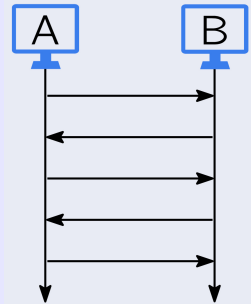
When is a Program Correct?

Sequential Programs



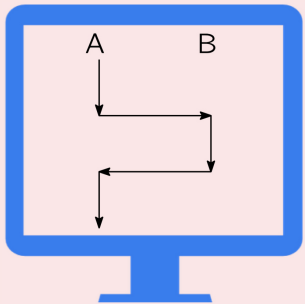
"Programs produce outputs that are consistent with their input"

Concurrent Programs



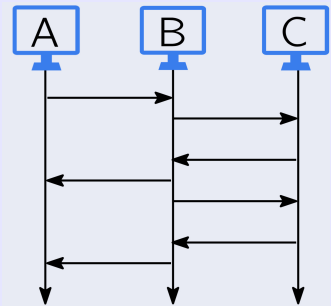
When is a Program Correct?

Sequential Programs



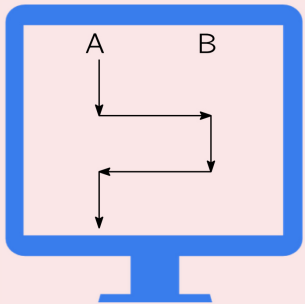
"Programs produce outputs that are consistent with their input"

Concurrent Programs



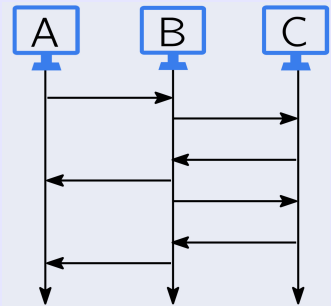
When is a Program Correct?

Sequential Programs



"Programs produce outputs that are consistent with their input"

Concurrent Programs



"Programs always respect their intended **protocols**"

Example: A Two-Buyer Protocol

Alice and **Bob** cooperate in buying a book from **Seller**

Example: A Two-Buyer Protocol

Alice and **Bob** cooperate in buying a book from **Seller** :

1. Alice sends a book title to Seller, who sends a quote back.

Example: A Two-Buyer Protocol

Alice and **Bob** cooperate in buying a book from **Seller** :

1. Alice sends a book title to Seller, who sends a quote back.
2. Alice checks whether Bob can contribute in buying the book.

Example: A Two-Buyer Protocol

Alice and **Bob** cooperate in buying a book from **Seller** :

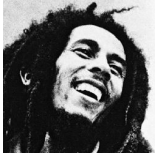
1. Alice sends a book title to Seller, who sends a quote back.
2. Alice checks whether Bob can contribute in buying the book.
3. Alice uses the answer from Bob (yes/no) to interact with Seller, either:
   a) completing the payment and arranging delivery details
   b) canceling the transaction

Example: A Two-Buyer Protocol

Alice and **Bob** cooperate in buying a book from **Seller** :

1. Alice sends a book title to Seller, who sends a quote back.
2. Alice checks whether Bob can contribute in buying the book.
3. Alice uses the answer from Bob (yes/no) to interact with Seller, either:
   a) completing the payment and arranging delivery details
   b) canceling the transaction
4. In case 3(a) Alice contacts Bob to get his address, and forwards it to Seller.

Example: A Two-Buyer Protocol



Alice and **Bob** cooperate in buying a book from **Seller** :

1. Alice sends a book title to Seller, who sends a quote back.
2. Alice checks whether Bob can contribute in buying the book.
3. Alice uses the answer from Bob (yes/no) to interact with Seller, either:
   a) completing the payment and arranging delivery details
   b) canceling the transaction
4. In case 3(a) Alice contacts Bob to get his address, and forwards it to Seller.

4'. In case 3(b) Alice is in charge of gracefully concluding the conversation.

Example: A Two-Buyer Protocol

Alice and **Bob** cooperate in buying a book from **Seller** :

1. Alice sends a book title to Seller, who sends a quote back.
2. Alice checks whether Bob can contribute in buying the book.
3. Alice uses the answer from Bob (yes/no) to interact with Seller, either:
   a) completing the payment and arranging delivery details
   b) canceling the transaction
4. In case 3(a) Alice contacts Bob to get his address, and forwards it to Seller.

4'. In case 3(b) Alice is in charge of gracefully concluding the conversation.

Note: The structure of messaging matters!

Example: A Two-Buyer Protocol

Desiderata for the implementations of Alice, Bob, and Seller:

- **Fidelity** – they **follow the intended protocol**.
  - Alice never ask Bob twice within the same conversation
  - Alice doesn't continue the transaction if Bob can't contribute
  - Alice chooses among the options provided by Seller

Example: A Two-Buyer Protocol

Desiderata for the implementations of Alice, Bob, and Seller:

- **Fidelity** – they **follow the intended protocol**.
- **Safety** – they don't feature **communication errors**.
  - Seller always returns an integer when Alice requests a quote

Example: A Two-Buyer Protocol

Desiderata for the implementations of Alice, Bob, and Seller:

- **Fidelity** – they **follow the intended protocol**.
- **Safety** – they don't feature **communication errors**.
- **Deadlock-Freedom** – they do not **"get stuck"** while running the protocol.
  - Alice eventually receives an answer from Bob on his contribution.

Example: A Two-Buyer Protocol

Desiderata for the implementations of Alice, Bob, and Seller:

- **Fidelity** – they **follow the intended protocol**.
- **Safety** – they don't feature **communication errors**.
- **Deadlock-Freedom** – they do not **"get stuck"** while running the protocol.
- **Termination** – they do not engage in **infinite behavior** (that may prevent them from completing the protocol)

Example: A Two-Buyer Protocol

Desiderata for the implementations of Alice, Bob, and Seller:

- **Fidelity** – they **follow the intended protocol**.
- **Safety** – they don't feature **communication errors**.
- **Deadlock-Freedom** – they do not **"get stuck"** while running the protocol.
- **Termination** – they do not engage in **infinite behavior** (that may prevent them from completing the protocol)

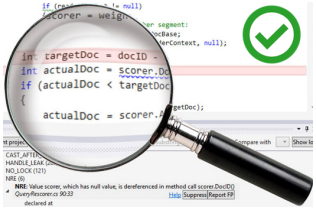
Correctness

- A **non-trivial notion**, which follows from the interplay of these properties.
- **Hard to enforce** due to actions being "scattered around" in programs.

→ A session specifies a protocol's structure, enabling program verification. It stipulates **what** and **when** should be exchanged (along a channel)

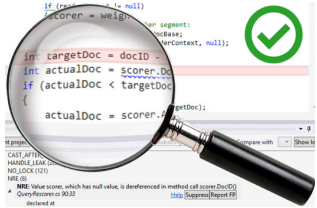
Type Systems

- Can detect bugs before programs are run
- Attached to many programming languages
- Implement a specific notion of correctness
  A program is either correct or incorrect



Type Systems

- Can detect bugs before programs are run
- Attached to many programming languages
- Implement a specific notion of correctness
  A program is either correct or incorrect

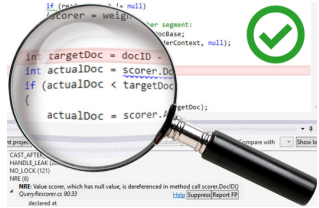


Sequential Languages

- **Data** type systems classify values in a program
- Examples: Integers, strings, etc

Type Systems

- Can detect bugs before programs are run
- Attached to many programming languages
- Implement a specific notion of correctness
  A program is either correct or incorrect



Sequential Languages

- **Data** type systems classify values in a program
- Examples: Integers, strings, etc

Concurrent Languages

- **Behavioral** type systems classify protocols in a program
- Example: “first send username, then receive true/false, finally close”
- A typical bug: sending messages in the wrong order

How to Design Type Systems? Logic to the Rescue!

-

How to Design Type Systems? Logic to the Rescue!

A landmark result in programming language theory:

- **Propositions as types** (Curry, 1935; Howard, 1969)

Propositions in Intuitionistic Logic	$\leftrightarrow$	Types
Proofs in natural deduction	$\leftrightarrow$	Sequential programs (λ-calculus)
Proof simplification	$\leftrightarrow$	Program evaluation (β-reduction)

How to Design Type Systems? Logic to the Rescue!

- **Propositions as types** (Curry, 1935; Howard, 1969)

Propositions in Intuitionistic Logic	$\leftrightarrow$	Types
Proofs in natural deduction	$\leftrightarrow$	Sequential programs (λ-calculus)
Proof simplification	$\leftrightarrow$	Program evaluation (β-reduction)

- What about concurrent programs? A **resource-oriented** view!

How to Design Type Systems? Logic to the Rescue!

- **Propositions as sessions**

Propositions in **Linear Logic** (LL)	$\leftrightarrow$	**Session types**
Proofs in LL	$\leftrightarrow$	Interacting processes
Cut elimination in LL	$\leftrightarrow$	process communication

Outline

Protocols as Session Types

Session types specify protocols in terms of

- communication actions: send and receive
- choices: offers and selections
- recursion

Protocols as Session Types

Session types specify protocols in terms of

- communication actions: send and receive
- choices: offers and selections
- recursion

Session protocols are attached to **interaction devices**:

- communication channels in programs (think Go and Rust)
- TCP-IP sockets
- ...

Protocols as Session Types

A formal syntax for protocols:

$S ::= \ !U; S$ **send value** of type U, continue as S

Protocols as Session Types

A formal syntax for protocols:

$$
\begin{array}{rcll}
S & ::= & !\,U\,;\,S & \textbf{send value} \text{ of type } U\text{, continue as } S \\
 & | & ?\,U\,;\,S & \textbf{receive value} \text{ of type } U\text{, continue as } S
\end{array}
$$

Protocols as Session Types

A formal syntax for protocols:

$$
\begin{array}{rcll}
S & ::= & !\,U\,;\,S & \textbf{send value} \text{ of type } U, \text{ continue as } S \\
& | & ?\,U\,;\,S & \textbf{receive value} \text{ of type } U, \text{ continue as } S \\
& | & \&\{l_1 : S_1, \cdots, l_n : S_n\} & \textbf{offer} \text{ the alternatives } S_1, \ldots, S_n
\end{array}
$$

Protocols as Session Types

A formal syntax for protocols:

$$
\begin{array}{rcll}
S & ::= & !U;S & \textbf{send value} \text{ of type } U, \text{ continue as } S \\
& | & ?U;S & \textbf{receive value} \text{ of type } U, \text{ continue as } S \\
& | & \&\{l_1 : S_1, \cdots, l_n : S_n\} & \textbf{offer} \text{ the alternatives } S_1, \ldots, S_n \\
& | & \oplus\{l_1 : S_1, \cdots, l_n : S_n\} & \textbf{select} \text{ one between } S_1, \ldots, S_n
\end{array}
$$

Protocols as Session Types

A formal syntax for protocols:

$$
\begin{array}{rll}
S ::= & !U; S & \textbf{send value} \text{ of type } U, \text{ continue as } S \\
| & ?U; S & \textbf{receive value} \text{ of type } U, \text{ continue as } S \\
| & \&\{l_1 : S_1, \cdots, l_n : S_n\} & \textbf{offer} \text{ the alternatives } S_1, \ldots, S_n \\
| & \oplus\{l_1 : S_1, \cdots, l_n : S_n\} & \textbf{select} \text{ one between } S_1, \ldots, S_n \\
| & \mu t.S \mid t & \textbf{recursion}
\end{array}
$$

Protocols as Session Types

A formal syntax for protocols:

$$
\begin{array}{rll}
S ::= & !U;S & \textbf{send value} \text{ of type } U, \text{ continue as } S \\
\mid & ?U;S & \textbf{receive value} \text{ of type } U, \text{ continue as } S \\
\mid & \&\{l_1 : S_1, \cdots, l_n : S_n\} & \textbf{offer} \text{ the alternatives } S_1, \ldots, S_n \\
\mid & \oplus\{l_1 : S_1, \cdots, l_n : S_n\} & \textbf{select} \text{ one between } S_1, \ldots, S_n \\
\mid & \mu t.S \mid t & \textbf{recursion} \\
\mid & \mathtt{end} & \textbf{terminated protocol}
\end{array}
$$

Protocols as Session Types

A formal syntax for protocols:

$$
\begin{array}{rll}
S ::= & !U; S & \textbf{send value} \text{ of type } U, \text{ continue as } S \\
\mid & ?U; S & \textbf{receive value} \text{ of type } U, \text{ continue as } S \\
\mid & \&\{l_1 : S_1, \cdots, l_n : S_n\} & \textbf{offer} \text{ the alternatives } S_1, \ldots, S_n \\
\mid & \oplus\{l_1 : S_1, \cdots, l_n : S_n\} & \textbf{select} \text{ one between } S_1, \ldots, S_n \\
\mid & \mu t.S \mid t & \textbf{recursion} \\
\mid & \texttt{end} & \textbf{terminated protocol}
\end{array}
$$

Notice:

- Sequential communication patterns (no built-in concurrency)
- U stands for basic values (e.g. `int`) but also other sessions S

The Two-Buyer Protocol, Revisited

Recall the protocol between Alice, Bob, and Seller:

1. Alice sends a book title to Seller, who sends a quote back.
2. Alice checks whether Bob can contribute in buying the book.
3. Alice uses the answer from Bob (yes/no) to interact with Seller, either:
   a) completing the payment and arranging delivery details
   b) canceling the transaction
4. In case 3(a) Alice contacts Bob to get his address, and forwards it to Seller.

4'. In case 3(b) Alice is in charge of gracefully concluding the conversation.

Session Types for The Two-Buyer Protocol

Two independent protocols, with Alice "leading" the interactions:

1. A session type for Seller (in its interaction with Alice):

$$S_{\mathsf{SA}} = ?\texttt{book};\ !\texttt{quote};\ \&\begin{cases}\text{buy}: & ?\texttt{paym};\ ?\texttt{address};\ !\texttt{ok};\ \text{end}\\ \text{cancel}: & ?\texttt{thanks};\ !\texttt{bye};\ \text{end}\end{cases}$$

Session Types for The Two-Buyer Protocol

Two independent protocols, with Alice "leading" the interactions:

1. A session type for Seller (in its interaction with Alice):

$$S_{\mathsf{SA}} = \texttt{?book}; \texttt{!quote}; \& \begin{cases} \text{buy}: & \texttt{?paym}; \texttt{?address}; \texttt{!ok}; \text{end} \\ \text{cancel}: & \texttt{?thanks}; \texttt{!bye}; \text{end} \end{cases}$$

2. A session type for Alice (in its interaction with Bob):

$$S_{\mathsf{AB}} = \texttt{!cost}; \& \begin{cases} \text{share}: & \texttt{?address}; \texttt{!ok}; \text{end} \\ \text{close}: & \texttt{!bye}; \text{end} \end{cases}$$

Session Types for The Two-Buyer Protocol

Implementations for Alice, Bob, and Seller should be **compatible**.

Session Types for The Two-Buyer Protocol

Implementations for Alice, Bob, and Seller should be **compatible**.

Duality relates session types with opposite behaviors:

- The dual of sending is receiving (and vice versa)
- Branching is the dual of selection (and vice versa)

The dual of session type S is written $\overline{S}$.

Session Types for The Two-Buyer Protocol

Example:

- Recall that S_{AB} describes Alice's viewpoint in her interaction with Bob:

$$S_{\mathsf{AB}} = \ !\texttt{cost}; \& \begin{cases} \text{share}: & ?\texttt{address}; \ !\texttt{ok}; \ \text{end} \\ \text{close}: & !\texttt{bye}; \ \text{end} \end{cases}$$

- Given this, Bob's implementation should conform to $\overline{S_{\mathsf{AB}}}$, the dual of S_{AB}:

Session Types for The Two-Buyer Protocol

Example:

- Recall that S_{AB} describes Alice's viewpoint in her interaction with Bob:

$$S_{\mathsf{AB}} = \ !\texttt{cost}; \& \begin{cases} \text{share} : & ?\texttt{address}; \ !\texttt{ok}; \ \text{end} \\ \text{close} : & !\texttt{bye}; \ \text{end} \end{cases}$$

- Given this, Bob's implementation should conform to $\overline{S_{\mathsf{AB}}}$, the dual of S_{AB}:

$$\overline{S_{\mathsf{AB}}} = \ ?\texttt{cost}; \oplus \begin{cases} \text{share} : & !\texttt{address}; \ ?\texttt{ok}; \ \text{end} \\ \text{close} : & ?\texttt{bye}; \ \text{end} \end{cases}$$

Session Types for The Two-Buyer Protocol

Example:

- Recall that S_{AB} describes Alice's viewpoint in her interaction with Bob:

$$S_{\mathsf{AB}} = \ !\texttt{cost}; \& \begin{cases} \text{share}: & ?\texttt{address}; \ !\texttt{ok}; \ \text{end} \\ \text{close}: & !\texttt{bye}; \ \text{end} \end{cases}$$

- Given this, Bob's implementation should conform to $\overline{S_{\mathsf{AB}}}$, the dual of S_{AB}:

$$\overline{S_{\mathsf{AB}}} = \ ?\texttt{cost}; \oplus \begin{cases} \text{share}: & !\texttt{address}; \ ?\texttt{ok}; \ \text{end} \\ \text{close}: & ?\texttt{bye}; \ \text{end} \end{cases}$$

- Also, Alice's implementation should conform to both $\overline{S_{\mathsf{SA}}}$ and S_{AB}.

Session Type Duality, Formally

Given a (finite) session type S, its dual type $\overline{S}$ is inductively defined as follows:

$$\begin{aligned}
\overline{!U;S} &= ?U;\overline{S} \\
\overline{?U;S} &= !U;\overline{S} \\
\overline{\&\{l_i : S_i\}_{i\in I}} &= \oplus\{l_i : \overline{S_i}\}_{i\in I} \\
\overline{\oplus\{l_i : S_i\}_{i\in I}} &= \&\{l_i : \overline{S_i}\}_{i\in I} \\
\overline{\mathsf{end}} &= \mathsf{end}
\end{aligned}$$

Notice:

- For recursive types, duality is defined **coinductively** (the dual of $\mu t.S$ is *not* $\mu t.\overline{S}$)

Many Classes of Session Types

There are many classes/variants of session types:

- We overviewed **binary** sessions, where types describe two-party protocols. (We used two separate types to handle Alice, Bob, and Seller.)
- With **multiparty** sessions, types describe protocols involving more than two participants. There is a single **global type** and multiple **local types**.

Many Classes of Session Types (contd.)

There are many classes/variants of session types:

- We saw a syntax of types that describes **regular behaviors**: both '$?U; S$' and '$!U; S$' are protocols, in which ';' composes an action with a protocol.
- In **context-free** session types, we can freely compose two protocols: '$S_1; S_2$'. This is arguably the most expressive class of protocols (to date).

Many Classes of Session Types (contd.)

There are many classes/variants of session types:

- Session types can be attached to languages with **synchronous** and **asynchronous** communication.
  - Synchronous: A message is received as soon as it is sent
  - Asynchronous: Message reception is not immediate

  These operational differences are reflected in types. The real world is asynchronous but synchronous is easier to define.

Many Classes of Session Types (contd.)

There are many classes/variants of session types:

- Session types can be attached to languages with **synchronous** and **asynchronous** communication.
  - Synchronous: A message is received as soon as it is sent
  - Asynchronous: Message reception is not immediate

  These operational differences are reflected in types. The real world is asynchronous but synchronous is easier to define.

- Session types are **paradigm-independent**, and can be adapted to functional, object-oriented, and actor-based languages with concurrency.

Taking Stock

Up to here:

- Correctness for communicating programs
- Sessions as protocol specifications
- A formal syntax for session types
- Example: A two-buyer protocol
- Protocol compatibility in terms of duality for session types

Coming next:

- Linear logic, in its intuitionistic variant