Propositions as Sessions

Logical Foundations of Concurrent Computation

Dan Frumin and Jorge A. Pérez
University of Groningen, The Netherlands
www.rug.nl/fse/fc
d.frumin | j.a.perez [[at]] rug.nl

ESSLLI 2024
(Part 1, v1.1)

About Us

Dan

- Assistant Professor, Fundamental Computing Group
- **Research Interests**: Semantics of programming languages, program verification, and concurrent separation logics

Jorge

- Associate Professor and Leader, Fundamental Computing Group
- **Research Interests**: Models and semantics of concurrency, type systems, relative expressiveness

This Course: Propositions as Sessions

This course concerns the logical foundations of concurrent computation.

- You may have heard about ‘**propositions**’, but what do we mean by ‘**sessions**’?
- In a nutshell, a session is a convenient way of structuring a series of related interactions between communicating programs.
- A disciplined of these structures is key to ensure program correctness.
  Not only: they have elegant logical foundations!

This Course: Propositions as Sessions

Plan:

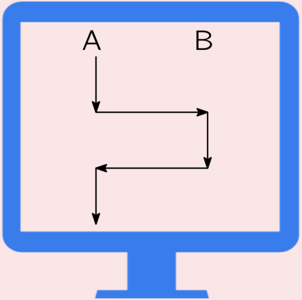
1. **Motivation** (Jorge) - Multiplicative Linear Logic (MLL) (Dan)
2. The concurrent interpretation of MALL (Jorge)
3. Cut-elimination and correctness for concurrent processes (Jorge)
4. Beyond linear resources: the !-modality and resource sharing (Dan)
5. An alternative view of resource sharing: Bunched Implications (Dan)

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

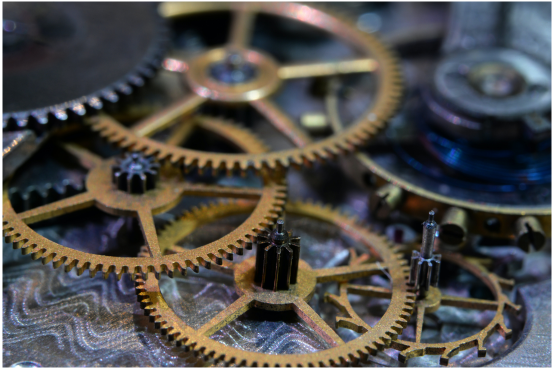
Message-Passing Concurrent Programs

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

Message-Passing Concurrent Programs

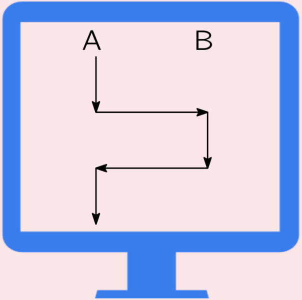
- Software components (**services**) distributed across networks
- Coordination takes place by sending and receiving messages
- Compatible message exchanges are crucial for system correctness
- A single faulty exchange can cause system-wide bugs

An (imperfect) analogy:



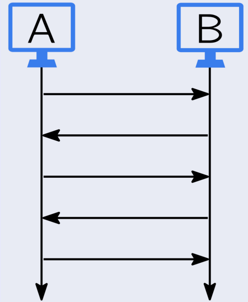
When is a Program Correct?

Sequential Programs



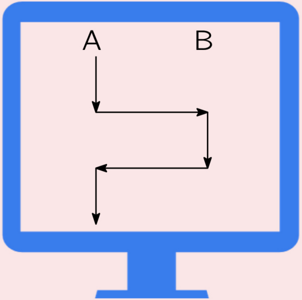
"Programs produce outputs that are consistent with their input"

Concurrent Programs



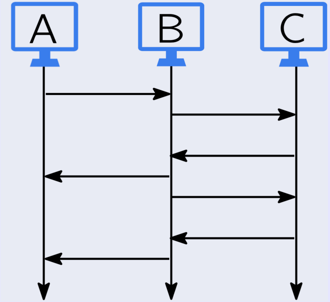
When is a Program Correct?

Sequential Programs



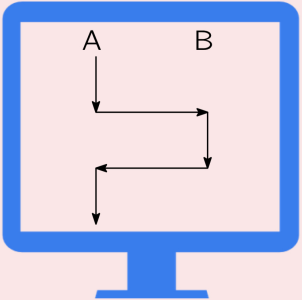
"Programs produce outputs that are consistent with their input"

Concurrent Programs



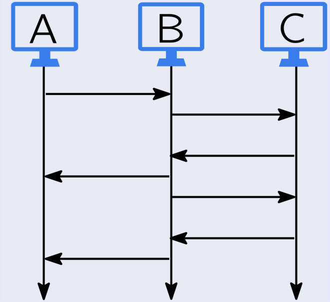
When is a Program Correct?

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.

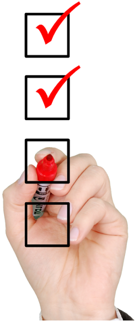
4'. In case 3(b) Alice is in charge of gracefully concluding the conversation.

Note: The structure and sequentiality of messaging matters!

Example: A Two-Buyer Protocol

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

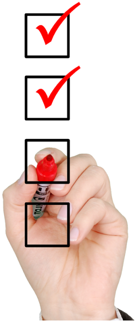
- **Fidelity** – they **follow the intended protocol**.
  - Alice never ask Bob twice within the same conversation
  - Alice doesn't continue the transaction if Bob can't contribute
  - Alice chooses among the options provided by Seller



Example: A Two-Buyer Protocol

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

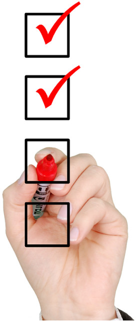
- **Fidelity** – they **follow the intended protocol**.
- **Safety** – they don't feature **communication errors**.
  - Seller always returns an integer when Alice requests a quote



Example: A Two-Buyer Protocol

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

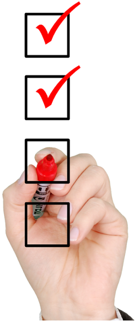
- **Fidelity** – they **follow the intended protocol**.
- **Safety** – they don't feature **communication errors**.
- **Deadlock-Freedom** – they do not "**get stuck**" while running the protocol.
  - Alice eventually receives an answer from Bob on his contribution.



Example: A Two-Buyer Protocol

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

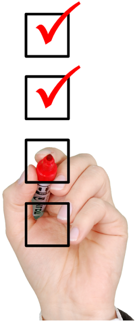
- **Fidelity** – they **follow the intended protocol**.
- **Safety** – they don't feature **communication errors**.
- **Deadlock-Freedom** – they do not "**get stuck**" while running the protocol.
- **Termination** – they do not engage in **infinite behavior** (that may prevent them from completing the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

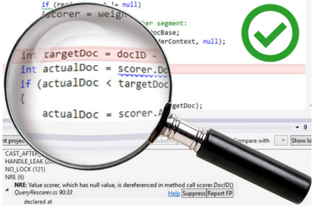
$\rightarrow$ Sessions specify a protocol's structure, enabling program verification.
A session stipulates **what** and **when** should be exchanged (along a channel)

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

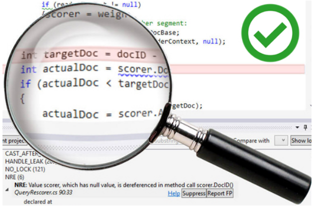
Type Systems

- 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



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 of characters

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	$\leftrightarrow$	Sequential programs
Proof simplification	$\leftrightarrow$	Program evaluation

How to Design Type Systems? Logic to the Rescue!

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

Propositions in Intuitionistic Logic	$\leftrightarrow$	Types
Proofs	$\leftrightarrow$	Sequential programs
Proof simplification	$\leftrightarrow$	Program evaluation

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

How to Design Type Systems? Logic to the Rescue!

- **Propositions as sessions** (this course!)

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

Protocols as Session Types

Session types uniformly describe protocols in terms of

- communication actions (send and receive)
- choices (offers and selections)
- sequential composition
- recursion

Protocols as Session Types

Session types uniformly describe protocols in terms of

- communication actions (send and receive)
- choices (offers and selections)
- sequential composition
- 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:

$$S ::= \; !U; S \qquad \text{send value of type } U\text{, continue as } S$$
$$\quad\mid\; ?U; S \qquad \text{receive value of type } U\text{, continue as } S$$

Protocols as Session Types

A formal syntax for protocols:

$S ::= \; !U; S$	**send value** of type U, continue as S
$\mid \; ?U; S$	**receive value** of type U, continue as S
$\mid \; \&\{l_1 : S_1, \cdots, l_n : S_n\}$	**offer** the alternatives $S_1, \ldots, S_n$

Protocols as Session Types

A formal syntax for protocols:

$S ::=$	$!U; S$	**send value** of type U, continue as S
$\mid$	$?U; S$	**receive value** of type U, continue as S
$\mid$	$\&\{l_1 : S_1, \cdots, l_n : S_n\}$	**offer** the alternatives $S_1, \ldots, S_n$
$\mid$	$\oplus\{l_1 : S_1, \cdots, l_n : S_n\}$	**select** one between $S_1, \ldots, S_n$

Protocols as Session Types

A formal syntax for protocols:

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

Protocols as Session Types

A formal syntax for protocols:

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

Protocols as Session Types

A formal syntax for protocols:

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

Notice:

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

Example: A Two-Buyer Protocol

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.

Example: A Two-Buyer Protocol

Two independent protocols, with Alice “leading” the interactions:

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

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

Example: A Two-Buyer Protocol

Two independent protocols, with Alice “leading” the interactions:

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

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

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

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

Example: A Two-Buyer Protocol

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

- **Duality** relates session types with opposite behaviors.
- Intuitively, the dual of sending is receiving (and vice versa).
- Similarly, branching is the dual of selection (and vice versa)

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

Example: A Two-Buyer Protocol

Example:

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

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

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

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

Example: A Two-Buyer Protocol

Example:

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

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

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

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

- 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:

- Duality for recursive session types is defined coinductively (the dual of $\mu t.S$ is *not* $\mu t.\overline{S}$)

Taking Stock

Up to here:

- Correctness for communicating programs
- Sessions as protocol specifications
- A formal syntax for session types
- Example: A two-buyer protocol
- The notion of duality for session types

Coming next:

- Linear logic, in its intuitionistic variant