A Logical Foundation for Session-based Concurrent Computation

Bernardo Toninho
[Co-advised by: Luís Caires and Frank Pfenning]

Computer Science Department
Carnegie Mellon University
Universidade Nova de Lisboa

Thesis Defense
Motivation
Concurrency and Distribution

Concurrent and Distributed Systems

- Systems composed of multiple logically and/or physically distinct interacting components.
- Pervasive in today’s society.
- Quite error prone, with real consequences:
  - Private information leaks due to security problems.
  - Unavailability of services due to internal orchestration errors.
Motivation
Concurrency and Distribution

Concurrent and Distributed Systems
- Systems composed of multiple logically and/or physically distinct interacting components.
- Pervasive in today’s society.
- Quite error prone, with real consequences:
  - Private information leaks due to security problems.
  - Unavailability of services due to internal orchestration errors.

Challenges
- Coordination of multiple interacting agents to offer a service.
- Resource availability and correct usage disciplines.
- How to ensure such a system is well-behaved?
Motivation

Goals

When is a system well-behaved?

- System doesn’t “get stuck” / deadlock.
- System is responsive (i.e. it exhibits some observable behavior).
- System behavior adheres to some specified protocol.
Motivation

Goals

When is a system well-behaved?

- System doesn’t “get stuck” / deadlock.
- System is responsive (i.e. it exhibits some observable behavior).
- System behavior adheres to some specified protocol.

Goals

- Study and develop language-based models of concurrency.
- Developing a logical foundation for message-passing concurrent computation.
- Robust enough to account for a variety of relevant phenomena.
- Ensuring strong safety properties by construction.
Process Calculi

- Algebraic model of concurrent message-passing processes.
- Enable study of behavior and interaction of systems.
- Enriched with types to enforce properties on processes.
Motivation
Language-based Models

Process Calculi
- Algebraic model of concurrent message-passing processes.
- Enable study of behavior and interaction of systems.
- Enriched with types to enforce properties on processes.

Types and Process Calculi
- Specify what data can be sent along channels (Simple types).
- Specify input/output capabilities of channels (I/O types).
- Specify communication behavior along channels (Session types).
Motivation
Connections with Logic

Issues

- Many different, sometimes *ad-hoc*, language features.
- Hard to reason about these languages in a uniform way.
- Challenging concerns when compared to non-concurrent setting:
  - Interactive behavior.
  - Non-local state and irreversible actions.
  - Resource awareness
- No deep connection with logic.

Why does logic matter?

- New means of reasoning about concurrent phenomena.
- Good metalogical properties map to good program properties.
- Enables compositional and incremental study of language features.
Motivation
Connections with Logic

Issues

- Many different, sometimes *ad-hoc*, language features.
- Hard to reason about these languages in a uniform way.
- Challenging concerns when compared to non-concurrent setting:
  - Interactive behavior.
  - Non-local state and irreversible actions.
  - Resource awareness
- No deep connection with logic.

Why does logic matter?

- New means of reasoning about concurrent phenomena.
- Good metalogical properties map to good program properties.
- Enables compositional and incremental study of language features.
Linear logic, specifically in its intuitionistic formulation, is a suitable logical foundation for message-passing concurrent computation, providing an elegant framework in which to express and reason about a multitude of naturally occurring phenomena in such a concurrent setting.
Contributions

- Linear logic and message-passing concurrency:
  - Background
  - Basic interpretation
  - Properties
  - Extensions
Contributions

- Linear logic and message-passing concurrency:
  - Background
  - Basic interpretation
  - Properties
  - Extensions

- A concurrent programming language:
  - Monadic encapsulation of concurrent computation
  - Recursion
  - Reconciliation with Logic
Contributions

- Linear logic and message-passing concurrency:
  - Background
  - Basic interpretation
  - Properties
  - Extensions

- A concurrent programming language:
  - Monadic encapsulation of concurrent computation
  - Recursion
  - Reconciliation with Logic

- Reasoning Techniques:
  - Linear logical relations
  - Logical equivalence
Linear Logic and Message-Passing Concurrency

Background

Linear Logic [Girard 87]

- A logic of resources and interaction.
- Resource independence captures parallelism.
- Linear logic as the logic of concurrency?
Linear Logic and Message-Passing Concurrency

Background

Linear Logic [Girard 87]
- A logic of resources and interaction.
- Resource independence captures parallelism.
- Linear logic as the logic of concurrency?

Linear Logic and Concurrency
Initial efforts explored the connections to concurrency:
- Abramsky’s computational interpretation [Abramsky 93]
- Bellin and Scott’s refinement to a $\pi$-calculus [BellinScott 94]
- No real “Curry-Howard interpretation”
Session Types [Honda 93]

- A structuring discipline for message passing concurrency.
- Session: Predetermined sequence of interactions along a channel.
- Session Types: Types are descriptions of communication behavior.
- Type correct programs adhere to the session discipline.
Session Types [Honda 93]

- A structuring discipline for message passing concurrency.
- Session: Predetermined sequence of interactions along a channel.
- Session Types: Types are descriptions of communication behavior.
- Type correct programs adhere to the session discipline.

Example

A server that receives login information from a client, on success, clients send a product name and receive back a purchase receipt:

\[
\begin{align*}
\text{ServerT} & \triangleq \text{login} \rightarrow (\text{sucLogin} \oplus \text{badLogin}) \\
\text{sucLogin} & \triangleq \text{prodName} \rightarrow \text{purchReceipt} \otimes 1
\end{align*}
\]
Session Types and ILL (SILL) [CairesPfenning10]

- It's possible to interpret session types as linear logic propositions.
- Proofs as typing derivations.
- Proof dynamics as process dynamics.
Session Types and ILL (SILL) [CairesPfenning10]
- It's possible to interpret session types as linear logic propositions.
- Proofs as typing derivations.
- Proof dynamics as process dynamics.

SILL in hindsight
- Assigns $\pi$-calculus processes to proofs – Not syntax driven.
- Channel forwarding as a derived concept.
- A few key proof reductions do not map to process interactions.
Linear Logic and Message-Passing Concurrency

Logical Interpretation

### Session Types and ILL (SILL) [CairesPfenning10]
- It's possible to interpret session types as linear logic propositions.
- Proofs as typing derivations.
- Proof dynamics as process dynamics.

### SILL in hindsight
- Assigns $\pi$-calculus processes to proofs – Not syntax driven.
- Channel forwarding as a derived concept.
- A few key proof reductions do not map to process interactions.

### Deviating from SILL
- Explicit linear forwarders.
- Syntax-driven assignment.
- Consistently mapped to $\pi$-calculus processes.
## Propositions as Types

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \otimes B$</td>
<td>Output a session of type $A$ and continue as $B$</td>
</tr>
<tr>
<td>$A \multimap B$</td>
<td>Input a session of type $A$ and continue as $B$</td>
</tr>
<tr>
<td>$A \land B$</td>
<td>Offer a choice between a session of type $A$ or $B$</td>
</tr>
<tr>
<td>$A \oplus B$</td>
<td>Select a session of type $A$ or $B$</td>
</tr>
<tr>
<td>$!A$</td>
<td>A persistent session of type $A$</td>
</tr>
<tr>
<td>$1$</td>
<td>Terminated session</td>
</tr>
</tbody>
</table>
### Propositions as Types

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \otimes B$</td>
<td>Output a session of type $A$ and continue as $B$</td>
</tr>
<tr>
<td>$A \multimap B$</td>
<td>Input a session of type $A$ and continue as $B$</td>
</tr>
<tr>
<td>$A \land B$</td>
<td>Offer a choice between a session of type $A$ or $B$</td>
</tr>
<tr>
<td>$A \oplus B$</td>
<td>Select a session of type $A$ or $B$</td>
</tr>
<tr>
<td>$!A$</td>
<td>A persistent session of type $A$</td>
</tr>
<tr>
<td>$1$</td>
<td>Terminated session</td>
</tr>
</tbody>
</table>

### Judgments

- **Process expression typing:** $\Gamma; \Delta \vdash P :: x:A$
- **\(\pi\)-calculus process typing:** $\Gamma; \Delta \Rightarrow \hat{P} :: x:A$
Typing Judgment

\[
\Gamma \vdash A_1, \ldots, A_m ; \Delta \vdash A_1, \ldots, A_n \vdash A
\]
Typing Judgment

\[ \Gamma \vdash u_1 : A_1, \ldots, u_m : A_m ; x_1 : A_1, \ldots, x_n : A_n \vdash P :: x : A \]

Process \( P \) provides \( A \) along \( x \) if composed with sessions in \( \Delta \) and \( \Gamma \).
Typing Judgment

\[ \Gamma \vdash u_1 : A_1, \ldots, u_m : A_m; x_1 : A_1, \ldots, x_n : A_n \vdash P :: x : A \]

Process \( P \) provides \( A \) along \( x \) if composed with sessions in \( \Delta \) and \( \Gamma \).

Cut as Composition

\[ \Gamma; \Delta \vdash A \quad \Gamma; \Delta', A \vdash C \]
\[ \Gamma; \Delta, \Delta' \vdash C \quad \text{cut} \]
Typing Judgment

\[
\Gamma \vdash \Delta \\
\begin{array}{c}
\vdash u_1 : A_1, \ldots, u_m : A_m; x_1 : A_1, \ldots, x_n : A_n \\
P :: x : A
\end{array}
\]

Process \( P \) provides \( A \) along \( x \) if composed with sessions in \( \Delta \) and \( \Gamma \).

Cut as Composition

\[
\Gamma; \Delta \vdash P :: x : A \quad \Gamma; \Delta' \vdash A \vdash C \\
\quad \Gamma; \Delta, \Delta' \vdash C \quad \text{cut}
\]
Typing Judgment

\[
\Gamma \vdash u_1: A_1, \ldots, u_m: A_m; x_1: A_1, \ldots, x_n: A_n \vdash P :: x: A
\]

Process \( P \) provides \( A \) along \( x \) if composed with sessions in \( \Delta \) and \( \Gamma \).

Cut as Composition

\[
\Gamma; \Delta \vdash P :: x: A \quad \Gamma; \Delta', x: A \vdash Q :: z: C
\]

\[
\Gamma; \Delta, \Delta' \vdash C \quad \text{cut}
\]
Typing Judgment

\[ \Gamma \vdash u_1 : A_1, \ldots, u_m : A_m, x_1 : A_1, \ldots, x_n : A_n \vdash P :: x : A \]

Process \( P \) provides \( A \) along \( x \) if composed with sessions in \( \Delta \) and \( \Gamma \).

Cut as Composition

\[ \frac{\Gamma; \Delta \vdash P :: x : A \quad \Gamma; \Delta' \vdash Q :: z : C}{\Gamma; \Delta, \Delta' \vdash \text{new } x.(P \parallel Q) :: z : C} \text{ cut} \]
Typing Judgment

\[
\frac{\Gamma \quad \Delta}{u_1:A_1, \ldots, u_m:A_m; x_1:A_1, \ldots, x_n:A_n \vdash P :: x:A}
\]

Process \( P \) provides \( A \) along \( x \) if composed with sessions in \( \Delta \) and \( \Gamma \).

Cut as Composition

\[
\frac{\Gamma; \Delta \vdash P :: x:A \quad \Gamma; \Delta', x:A \vdash Q :: z:C}{\Gamma; \Delta, \Delta' \vdash \text{new } x.(P || Q) :: z:C \quad \text{cut}}
\]

Parallel composition of \( P \), offering \( x:A \) and \( Q \), using \( x:A \).

Identity as Forwarding

\[
\frac{\Gamma; A \vdash A}{\text{id}}
\]
Linear Logic and Message-Passing Concurrency
Logical Interpretation - Judgmental Principles

Typing Judgment

\[ \Gamma, u_1 : A_1, \ldots, u_m : A_m, x_1 : A_1, \ldots, x_n : A_n \vdash P :: x : A \]

Process \( P \) provides \( A \) along \( x \) if composed with sessions in \( \Delta \) and \( \Gamma \).

Cut as Composition

\[ \Gamma, \Delta \vdash P :: x : A \quad \Gamma, \Delta', x : A \vdash Q :: z : C \]

\[ \Gamma ; \Delta, \Delta' \vdash \text{new} \; x . (P \parallel Q) :: z : C \]

Parallel composition of \( P \), offering \( x : A \) and \( Q \), using \( x : A \).

Identity as Forwarding

\[ \Gamma ; x : A \vdash \text{id} \]

\[ A \]
Typing Judgment

$$\Gamma \vdash u_1 : A_1, \ldots, u_m : A_m; x_1 : A_1, \ldots, x_n : A_n \vdash P :: x : A$$

Process $P$ provides $A$ along $x$ if composed with sessions in $\Delta$ and $\Gamma$.

Cut as Composition

$$\frac{\Gamma; \Delta \vdash P :: x : A \quad \Gamma; \Delta', x : A \vdash Q :: z : C}{\Gamma; \Delta, \Delta' \vdash \text{new } x . (P \parallel Q) :: z : C} \text{ cut}$$

Parallel composition of $P$, offering $x : A$ and $Q$, using $x : A$.

Identity as Forwarding

$$\frac{}{\Gamma; x : A \vdash \text{fwd } x \ z :: z : A} \text{id}$$
Session Output

\[
\begin{align*}
\Gamma; \Delta &\vdash A & \Gamma; \Delta' &\vdash B \\
\Gamma; \Delta, \Delta' &\vdash A \otimes B \otimes R
\end{align*}
\]
Linear Logic and Message-Passing Concurrency
Logical Interpretation - Tensor as Output

Session Output

\[
\Gamma; \Delta \vdash P :: y:A \quad \Gamma; \Delta' \vdash Q :: z:B \\
\Gamma; \Delta, \Delta' \vdash A \otimes B \otimes R
\]
Session Output

\[
\Gamma;\Delta \vdash P :: y:A \quad \Gamma;\Delta' \vdash Q :: z:B
\]

\[
\Gamma;\Delta,\Delta' \vdash \text{output } z \,(y.P);\ Q :: z:A \otimes B \quad \otimes R
\]
Bernardo Toninho (UNL&CMU)  
Thesis Defense  

Session Output

\[
\Gamma; \Delta \vdash P :: y : A \quad \Gamma; \Delta' \vdash Q :: z : B \\
\Gamma; \Delta, \Delta' \vdash \text{output } z \ (y . P) ; Q :: z : A \otimes B \\
\Gamma; \Delta, \quad A, \quad B \vdash \quad C \\
\Gamma; \Delta, \quad A \otimes B \vdash \\
\Gamma; \Delta, \quad C \\
\]

\[\otimes R\]

\[\otimes L\]
Session Output

\[
\Gamma; \Delta \vdash P :: y:A \quad \Gamma; \Delta' \vdash Q :: z:B
\]
\[
\Gamma; \Delta, \Delta' \vdash \text{output } z \ (y.P); Q :: z:A \otimes B \quad \otimes R
\]

\[
\Gamma; \Delta, y : A, x : B \vdash R :: z:C
\]
\[
\Gamma; \Delta, A \otimes B \vdash C \quad \otimes L
\]
Linear Logic and Message-Passing Concurrency
Logical Interpretation - Tensor as Output

Session Output

\[ \Gamma; \Delta \vdash P :: y : A \quad \Gamma; \Delta' \vdash Q :: z : B \]
\[ \Gamma; \Delta, \Delta' \vdash \text{output } z (y.P); Q :: z : A \otimes B \]
\[ \otimes R \]

\[ \Gamma; \Delta, y : A, x : B \vdash R :: z : C \]
\[ \Gamma; \Delta, x : A \otimes B \vdash y \leftarrow \text{input } x; R :: z : C \]
\[ \otimes L \]
Linear Logic and Message-Passing Concurrency

Logical Interpretation - Tensor as Output

Session Output

\[
\frac{\Gamma; \Delta \vdash P :: y:A \quad \Gamma; \Delta' \vdash Q :: z:B}{\Gamma; \Delta, \Delta' \vdash \text{output } z \ (y.P); \ Q :: z:A \otimes B} \quad \otimes R
\]

\[
\frac{\Gamma; \Delta, y : A, x : B \vdash R :: z:C}{\Gamma; \Delta, x:A \otimes B \vdash y \leftarrow \text{input } x; \ R :: z:C} \quad \otimes L
\]

Proof Reduction

\[
\Gamma; \Delta, \Delta' \vdash \text{new } x.((\text{output } x \ (y.P); \ Q) \parallel y \leftarrow \text{input } x; \ R) :: z:C
\]
Session Output

\[
\Gamma; \Delta \vdash P :: y : A \quad \Gamma; \Delta' \vdash Q :: z : B
\]
\[
\Gamma; \Delta, \Delta' \vdash \text{output } z (y.P); Q :: z : A \otimes B
\]
\[
\Gamma; \Delta, y : A, x : B \vdash R :: z : C
\]
\[
\Gamma; \Delta, x : A \otimes B \vdash y \leftarrow \text{input } x; R :: z : C
\]

Proof Reduction

\[
\Gamma; \Delta, \Delta' \vdash \text{new } x.((\text{output } x (y.P); Q) \parallel y \leftarrow \text{input } x; R) :: z : C
\]
\[
\rightarrow \quad \Gamma; \Delta, \Delta' \vdash \text{new } x.(Q \parallel \text{new } y.(P \parallel R)) :: z : C
\]
Session Input

\[
\frac{\Gamma; \Delta, \ A \vdash B}{\Gamma; \Delta \vdash A \rightarrow B} \quad \rightarrow^\circ R
\]
Session Input

\[
\frac{\Gamma; \Delta, x : A \vdash P :: z : B}{\Gamma; \Delta \vdash A \multimap B \quad \multimap R}
\]
Session Input

\[
\Gamma; \Delta, x : A \vdash P :: z : B
\]

\[
\Gamma; \Delta \vdash x \leftarrow \text{input } z; P :: z : A \rightarrow B \quad \rightarrow R
\]
Logical Interpretation - Implication as Input

Session Input

\[
\frac{\Gamma; \Delta, x : A \vdash P :: z : B}{\Gamma; \Delta \vdash x \leftarrow \text{input } z; P :: z : A \rightarrow B} \quad \circ R
\]

\[
\frac{\Gamma; \Delta \vdash A \quad \Gamma; \Delta' \vdash B \vdash C}{\Gamma; \Delta, \Delta', A \rightarrow B \vdash C} \quad \circ L
\]
Linear Logic and Message-Passing Concurrency
Logical Interpretation - Implication as Input

Session Input

\[ \Gamma; \Delta, x : A \vdash P :: z:B \]
\[ \frac{\Gamma; \Delta \vdash x \leftarrow \text{input } z; P :: z:A \rightarrow B}{\rightarrow R} \]

\[ \Gamma; \Delta \vdash Q_1 :: y:A \quad \Gamma; \Delta', x:B \vdash Q_2 :: z:C \]
\[ \frac{\Gamma; \Delta, \Delta', A \rightarrow B \vdash}{\rightarrow L} C \]
Session Input

$$\Gamma; \Delta, x : A \vdash P :: z:B$$

$$\Gamma; \Delta \vdash x \leftarrow \text{input } z; P :: z:A \multimap B \quad \rightarrow R$$

$$\Gamma; \Delta \vdash Q_1 :: y:A \quad \Gamma; \Delta', x:B \vdash Q_2 :: z:C$$

$$\Gamma; \Delta, \Delta', x:A \multimap B \vdash \text{output } x (y.Q_1); Q_2) :: z:C \quad \rightarrow L$$
Linear Logic and Message-Passing Concurrency

Logical Interpretation - Implication as Input

Session Input

\[ \frac{\Gamma; \Delta, x : A \vdash P :: z : B}{\Gamma; \Delta \vdash x \leftarrow \text{input } z; P :: z : A \rightarrow B} \quad \rightarrow R \]

\[ \frac{\Gamma; \Delta \vdash Q_1 :: y : A \quad \Gamma; \Delta', x : B \vdash Q_2 :: z : C}{\Gamma; \Delta, \Delta', x : A \rightarrow B \vdash \text{output } x (y. Q_1); Q_2) :: z : C} \quad \rightarrow L \]

Linear Implication as input. Reduction is the same as for \( \otimes \).
Multiplicative Unit

\[
\frac{\Gamma; \cdot \vdash}{1 \ 1^R}
\]
Multiplicative Unit

\[ \Gamma; \cdot \vdash \text{close } z :: z : 1 \]

\[ 1R \]
Multiplicative Unit

\[
\frac{\Gamma; \cdot \vdash \text{close } z :: z:1}{1R} \quad \frac{\Gamma; \Delta \vdash C}{C \quad 1L}
\]

\[
\frac{\Gamma; \Delta, 1 \vdash}{\Gamma; \Delta \vdash C}
\]
Multiplicative Unit

\[ \Gamma; \vdash \text{close } z :: z : 1 \quad 1R \]

\[ \Gamma; \Delta, 1 \vdash C \quad 1L \]

\[ \Gamma; \Delta \vdash Q :: z : C \]
Linear Logic and Message-Passing Concurrency

Logical Interpretation - Multiplicative Unit as Termination

### Multiplicative Unit

\[ \Gamma; \cdot \vdash \text{close} \, z :: z:1 \]

**1R**

\[ \Gamma; \Delta \vdash Q :: z:C \]

\[ \Gamma; \Delta, x:1 \vdash \text{wait} \, x; Q :: z:C \]

**1L**

### Proof Reduction

\[ \Gamma; \Delta \vdash \text{new} \, x. (\text{close} \, x \parallel (\text{wait} \, x; Q)) :: z:C \]
Multiplicative Unit

\[ \Gamma; \cdot \vdash \text{close } z :: z:1 \quad 1R \]

\[ \Gamma; \Delta \vdash Q :: z:C \]

\[ \Gamma; \Delta, x:1 \vdash \text{wait } x; Q :: z:C \quad 1L \]

Proof Reduction

\[ \Gamma; \Delta \vdash \text{new } x.(\text{close } x \parallel (\text{wait } x; Q)) :: z:C \]

\[ \quad \rightarrow \quad \Gamma; \Delta \vdash Q :: z:C \]
Additive Conjunction

\[
\frac{\Gamma; \Delta \vdash A \quad \Gamma; \Delta \vdash B}{\Gamma; \Delta \vdash A \& B}
\]

& \quad R
Additive Conjunction

\[
\Gamma; \Delta \vdash P_1 :: x:A \quad \Gamma; \Delta \vdash P_2 :: x:B \\
\Gamma; \Delta \vdash A \& B \quad \&R
\]
Additive Conjunction

\[
\frac{\Gamma; \Delta \vdash P_1 :: x:A \quad \Gamma; \Delta \vdash P_2 :: x:B}{\Gamma; \Delta \vdash x.\text{case}(P_1, P_2) :: x:A \& B} \quad \& R
\]
Additive Conjunction

\[
\frac{\Gamma; \Delta \vdash P_1 :: x:A \quad \Gamma; \Delta \vdash P_2 :: x:B}{\Gamma; \Delta \vdash x.\text{case}(P_1, P_2) :: x:A \land B} \quad \&R
\]

\[
\frac{\Gamma; \Delta, \quad A \vdash C}{\Gamma; \Delta, \quad A \land B \vdash C} \quad \&L_1
\]
Additive Conjunction

\[ \frac{\Gamma; \Delta \vdash P_1 :: x:A \quad \Gamma; \Delta \vdash P_2 :: x:B}{\Gamma; \Delta \vdash x.\text{case}(P_1, P_2) :: x:A \land B} \quad R \]

\[ \frac{\Gamma; \Delta, x:A \vdash Q :: z:C}{\Gamma; \Delta, A \land B \vdash C} \quad L_1 \]
Additive Conjunction

\[ \Gamma; \Delta \vdash P_1 :: x:A \quad \Gamma; \Delta \vdash P_2 :: x:B \]
\[ \Gamma; \Delta \vdash x.\text{case}(P_1, P_2) :: x:A \land B \]

& \text{R}

\[ \Gamma; \Delta, x:A \vdash Q :: z:C \]
\[ \Gamma; \Delta, x:A \land B \vdash x.\text{inl}; Q :: z:C \]

& \text{L}_1
Additive Conjunction

\[
\begin{align*}
\frac{\Gamma; \Delta \vdash P_1 :: x:A \quad \Gamma; \Delta \vdash P_2 :: x:B}{\Gamma; \Delta \vdash x.\text{case}(P_1, P_2) :: x:A \& B} & \quad \& R \\
\frac{\Gamma; \Delta, x:A \vdash Q :: z:C}{\Gamma; \Delta, x:A \& B \vdash x.\text{inl}; Q :: z:C} & \quad \& L_1
\end{align*}
\]

Proof Reduction

\[
\begin{align*}
\frac{\Gamma; \Delta, \Delta' \vdash \text{new } x.(x.\text{case}(P_1, P_2) \parallel x.\text{inl}; Q) :: z:C}{\Gamma; \Delta, \Delta' \vdash \text{new } x.(P_1 \parallel Q) :: z:C}
\end{align*}
\]
### Additive Conjunction

$$\Gamma; \Delta \vdash P_1 :: x:A \quad \Gamma; \Delta \vdash P_2 :: x:B$$

$$\Gamma; \Delta \vdash x.\text{case}(P_1, P_2) :: x:A \& B$$

$$\Gamma; \Delta, x:A \vdash Q :: z:C$$

$$\Gamma; \Delta, x:A \& B \vdash x.\text{inl}; Q :: z:C$$

### Proof Reduction

$$\Gamma; \Delta, \Delta' \vdash \text{new } x.(x.\text{case}(P_1, P_2) \parallel x.\text{inl}; Q) :: z:C$$

$$\rightarrow \Gamma; \Delta, \Delta' \vdash \text{new } x.(P_1 \parallel Q) :: z:C$$

Additive disjunction $$\oplus$$ is dual.
A Server and a Client

ServerT ≜ login ↘ (sucLogin ⊕ badLogin)
sucLogin ≜ prodName ↘ purchReceipt ⊗ 1
### Linear Logic and Message-Passing Concurrency

#### Example

**A Server and a Client**

<table>
<thead>
<tr>
<th>ServerT</th>
<th>(\triangleq) login (\rightsquo;) (sucLogin (\oplus) badLogin)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sucLogin</td>
<td>(\triangleq) prodName (\rightsquo;) purchReceipt (\otimes) 1</td>
</tr>
</tbody>
</table>

\[c:ServerT \vdash Client :: z:1\]
Linear Logic and Message-Passing Concurrency

Example

A Server and a Client

ServerT ≜ login → (sucLogin ⊕ badLogin)
sucLogin ≜ prodName → purchReceipt ⊗ 1

\[ c : ServerT \vdash Client :: z : 1 \]

Client ≜ output c login; c.case(Buy, Error)
Buy ≜ output c ticket; r ← input c; wait c; close z
A Server and a Client

ServerT \triangleq \text{login} \multimap (\text{sucLogin} \uplus \text{badLogin})

\text{sucLogin} \triangleq \text{prodName} \multimap \text{purchReceipt} \otimes 1

\text{Client} \triangleq \text{output c login}; \text{c.case}(\text{Buy, Error})

\text{Buy} \triangleq \text{output c ticket}; \text{r} \leftarrow \text{input c}; \text{wait c}; \text{close z}

\vdash \text{new c.(Server |\| Client)} :: z.1
Type Preservation

If $\Gamma; \Delta \Rightarrow P :: z:A$ and $P \rightarrow P'$ then $\Gamma; \Delta \Rightarrow P' :: z:A$
Type Preservation
If $\Gamma; \Delta \Rightarrow P :: z:A$ and $P \rightarrow P'$ then $\Gamma; \Delta \Rightarrow P' :: z:A$

Global Progress
Let $\cdot; \cdot \Rightarrow P :: z:1$. If $P$ is live then $P \rightarrow Q$. 
Limitations of (simple) Session Types

- Hard to extend beyond simple communication patterns.
- No uniform way of expressing properties of exchanged *data*.
- Can scale to richer settings, but technically challenging.
Limitations of (simple) Session Types

- Hard to extend beyond simple communication patterns.
- No uniform way of expressing properties of exchanged data.
- Can scale to richer settings, but technically challenging.

Logical Foundation

- New means of reasoning about concurrent phenomena.
- Compositional and incremental study of language features:
  - Value-dependent Session Types
  - Polymorphic Session Types
Value-dependent Session Types [Toninho et al.11]

- Two new types: $\forall x: \tau. A$ and $\exists x: \tau. A$
- Parametric in the language of types $\tau$.
- $\forall x: \tau. A$ - Input a term $M : \tau$, continue as $A(M)$.
- $\exists x: \tau. A$ - Output a term $M : \tau$, continue as $A(M)$.
- If $\tau$s are dependent: proof communication.
Value-dependent Session Types [Toninho et al.11]

- Two new types: $\forall x:\tau.A$ and $\exists x:\tau.A$
- Parametric in the language of types $\tau$.
- $\forall x:\tau.A$ - Input a term $M : \tau$, continue as $A(M)$.
- $\exists x:\tau.A$ - Output a term $M : \tau$, continue as $A(M)$.
- If $\tau$s are dependent: proof communication.

Revisiting the Server-Client

ServerT$_{\forall\exists} \triangleq \forall u : uid.(\text{succLogin}_{\forall\exists} \oplus \text{badLogin})$

sucLogin$_{\forall\exists} \triangleq \forall p : \text{prodName}.\forall c : \text{cupon}(u, p).\exists r : \text{receipt}(u, p).1$
Extend the Judgment

$$\psi; \Gamma; \Delta \vdash P :: z{:}A$$

$\psi$ accounts for variables from the language of $\tau$. 
Extend the Judgment

\[ \psi; \Gamma; \Delta \vdash P :: z:A \]

\( \psi \) accounts for variables from the language of \( \tau \).

Universal Quantification

\[
\frac{\psi, \tau; \Gamma; \Delta \vdash A}{\forall x : \tau. A}
\]

\[ \forall R \]

\[ \psi; \Gamma; \Delta \vdash \forall x : \tau. A \]
Extend the Judgment

\[ \Psi; \Gamma; \Delta \vdash P :: z:A \]

\(\Psi\) accounts for variables from the language of \(\tau\).

Universal Quantification

\[
\frac{\Psi, x : \tau; \Gamma; \Delta \vdash P :: z:A}{\Psi; \Gamma; \Delta \vdash \forall x : \tau. A} \quad \forall R
\]
Extend the Judgment

\[ \psi; \Gamma; \Delta \vdash P :: z:A \]

\( \psi \) accounts for variables from the language of \( \tau \).

Universal Quantification

\[
\frac{\psi, x : \tau; \Gamma; \Delta \vdash P :: z:A}{\psi; \Gamma; \Delta \vdash x \leftarrow \text{input } z; P :: z: \forall x : \tau. A} \quad \forall R
\]

\[
\frac{\psi \vdash M : \tau \quad \psi; \Gamma; \Delta, A\{M/x\} \vdash}{\psi; \Gamma; \Delta, \forall x : \tau. A \vdash} \quad \forall L
\]
Extend the Judgment

\[ \psi; \Gamma; \Delta \vdash P : z: A \]

\( \psi \) accounts for variables from the language of \( \tau \).

Universal Quantification

\[ \psi, x : \tau; \Gamma; \Delta \vdash P : z: A \]

\[ \frac{\psi; \Gamma; \Delta \vdash x \leftarrow \text{input } z; P : z: \forall x: \tau. A}{ \forall R} \]

\[ \psi \vdash M : \tau \quad \psi; \Gamma; \Delta, A\{M/x\} \vdash \quad C \]

\[ \frac{\psi; \Gamma; \Delta, \forall x: \tau. A \vdash \quad C \quad \forall L}{C} \]
Extend the Judgment

\[ \psi; \Gamma; \Delta \vdash P :: z : A \]

\( \psi \) accounts for variables from the language of \( \tau \).

Universal Quantification

\[ \psi, x : \tau; \Gamma; \Delta \vdash P :: z : A \]

\[ \psi; \Gamma; \Delta \vdash x \leftarrow \text{input } z; P :: z: \forall x: \tau. A \]

\[ \psi \vdash M : \tau \quad \psi; \Gamma, x : A \{ M/x \} \vdash Q :: z : C \]

\[ \psi; \Gamma; \Delta, \forall x: \tau. A \vdash \quad C \]

\( \forall L \)

\( \forall R \)
Linear Logic and Message-Passing Concurrency
Extending the Interpretation - Value Dependencies

Extend the Judgment

\[ \psi; \Gamma; \Delta \vdash P :: z:A \]

\( \psi \) accounts for variables from the language of \( \tau \).

Universal Quantification

\[ \psi, x:\tau; \Gamma; \Delta \vdash P :: z:A \]

\[ \frac{ }{ \psi; \Gamma; \Delta \vdash x \leftarrow \text{input } z; P :: z:\forall x:\tau.A } \] \( \forall R \)

\[ \psi \vdash M : \tau \]

\[ \psi; \Gamma; \Delta, x:A\{M/x\} \vdash Q :: z:C \]

\[ \frac{ }{ \psi; \Gamma; \Delta, x:\forall x:\tau.A \vdash \text{output } x \ M; Q :: z:C } \] \( \forall L \)
Refining Quantification [Toninho et al.11, Pfenning et al.11]

- Proof Irrelevance: $[\tau] -$ Proofs of $\tau$ are not used at runtime.
- Affirmation: $\diamondsuit_{K^\tau} - K$ affirms the existence of a proof of $\tau$. 

Type safety (preservation and progress) also entails adherence to constraints on data!
Linear Logic and Message-Passing Concurrency
Extending the Interpretation - Value Dependencies

Refining Quantification [Toninho et al.11, Pfenning et al.11]

- Proof Irrelevance: $[\tau]$ – Proofs of $\tau$ are not used at runtime.
- Affirmation: $\diamondsuit_{K\tau} - K$ affirms the existence of a proof of $\tau$.

Refining the Example

\[
sucLogin[] \triangleq \forall p : \text{prodName.} \forall c : [\text{cupon}(u, p)]. \exists r : \text{receipt}(u, p) \cdot 1
\]
Refining Quantification [Toninho et al.11, Pfenning et al.11]

- Proof Irrelevance: $[\tau]$ – Proofs of $\tau$ are not used at runtime.
- Affirmation: $\Diamond K^\tau$ – $K$ affirms the existence of a proof of $\tau$.

Refining the Example

\[
\text{sucLogin}_\square \triangleq \forall p : \text{prodName}. \forall c : [\text{cupon}(u, p)]. \exists r : \text{receipt}(u, p). 1
\]

\[
\text{sucLogin}_\Diamond \triangleq \forall p : \text{prodName}. \forall c : [\text{cupon}(u, p)]. \exists r : \Diamond_S [\text{mreceipt}(u, p)]. 1
\]
Refining Quantification [Toninho et al.11, Pfenning et al.11]

- Proof Irrelevance: $[\tau]$ – Proofs of $\tau$ are not used at runtime.
- Affirmation: $\diamond_K \tau$ – $K$ affirms the existence of a proof of $\tau$.

Refining the Example

\[
\text{sucLogin} \triangleq \forall p : \text{prodName} \forall c : \text{cupon}(u, p). \exists r : \text{receipt}(u, p).
\]

\[
\text{sucLogin} \gg \triangleq \forall p : \text{prodName} \forall c : \text{cupon}(u, p). \exists r : \diamond_S \text{mreceipt}(u, p).
\]

Type safety (preservation and progress) also entails adherence to constraints on data!
Parametric Polymorphism [Pérez et al.11, Wadler11]

- Second-order quantification.
- Communication of session types / abstract protocols.
- Parametricity results in the style of System F.

Universal Quantification (Second Order)

\[
\begin{array}{c}
\Omega, X; \Gamma; \Delta \vdash A \\
\Omega; \Gamma; \Delta \vdash \forall X.A
\end{array}
\]

\[\forall R2\]
Parametric Polymorphism [Pérez et al. 2011, Wadler 2011]

- Second-order quantification.
- Communication of session types / abstract protocols.
- Parametricity results in the style of System F.

Universal Quantification (Second Order)

\[ \Omega, \Gamma; \Delta \vdash P :: z : A \]

\[ \Omega; \Gamma; \Delta \vdash \forall X.A \quad \forall \text{R2} \]
Parametric Polymorphism [Pérez et al.11, Wadler11]

- Second-order quantification.
- Communication of session types / abstract protocols.
- Parametricity results in the style of System F.

Universal Quantification (Second Order)

\[
\begin{align*}
\Omega, X; \Gamma; \Delta & \vdash P :: z : A \\
\Omega; \Gamma; \Delta & \vdash X \leftarrow \text{input } z; P :: z : \forall X. A & \forall R2
\end{align*}
\]
Parametric Polymorphism [Pérez et al.11, Wadler11]

- Second-order quantification.
- Communication of session types / abstract protocols.
- Parametricity results in the style of System F.

Universal Quantification (Second Order)

\[
\frac{\Omega, X; \Gamma; \Delta \vdash P :: z:A}{\Omega; \Gamma; \Delta \vdash X \leftarrow \text{input } z; P :: z:\forall X.A} \quad \forall R2
\]

\[
\frac{\Omega \vdash B \text{ type} \quad \Omega; \Gamma; \Delta, A\{B/X\} \vdash \quad C}{\Omega; \Gamma; \Delta, \forall X.A \vdash \quad C} \quad \forall L2
\]
Parametric Polymorphism [Pérez et al. 11, Wadler 11]

- Second-order quantification.
- Communication of session types / abstract protocols.
- Parametricity results in the style of System F.

Universal Quantification (Second Order)

\[\begin{align*}
\Omega, X; \Gamma; \Delta & \vdash P :: z: A \\
\Omega; \Gamma; \Delta & \vdash X \gets \text{input } z; P :: z: \forall X.A \\
\end{align*}\]  \(\forall R2\)

\[\begin{align*}
\Omega & \vdash B \text{ type} \\
\Omega; \Gamma; \Delta & \vdash x : A\{B/X\} \vdash Q :: z: C \\
\Omega; \Gamma; \Delta, & \forall X.A \vdash \\
\Omega; \Gamma; \Delta, & \forall X.A \vdash C \\
\end{align*}\]  \(\forall L2\)
### Parametric Polymorphism

- Second-order quantification.
- Communication of session types / abstract protocols.
- Parametricity results in the style of System F.

### Universal Quantification (Second Order)

\[
\begin{align*}
\Omega, X; \Gamma; \Delta & \vdash P :: z:A \\
\Omega; \Gamma; \Delta & \vdash X \leftarrow \text{input } z; \quad P :: z:\forall X.A
\end{align*}
\]

\[\forall R2\]

\[
\begin{align*}
\Omega & \vdash B \text{ type} \\
\Omega; \Gamma; \Delta, x:A\{B/X\} & \vdash Q :: z:C
\end{align*}
\]

\[\forall L2\]

\[
\begin{align*}
\Omega; \Gamma; \Delta, x:\forall X.A & \vdash \text{output } x \ B; \quad Q :: z:C
\end{align*}
\]
Parametric Polymorphism

CloudServer \triangleq \forall X.!(\text{api} \to X) \dashv \ll 

Specifies a generic service that:

- Inputs any type (e.g. GMaps).
- Inputs a persistent session that, given an implementation of \text{api}, provides the previously input type (e.g. !(\text{api}(\text{GMaps}))).
- Composes the received session accordingly, subsequently providing a persistent session of the given type (e.g. !GMaps).
CloudServer $\triangleq \forall X.!(\text{api} \multimap X) \multimap !X$

Specifies a generic service that:
- Inputs any type (e.g. GMaps).
CloudServer $\triangleq \forall X.!(\text{api} \rightarrow X) \rightarrow !X$

Specifies a generic service that:

- Inputs any type (e.g. GMaps).
- Inputs a persistent session that, given an implementation of api, provides the previously input type (e.g. !(api $\rightarrow$ GMaps)).
CloudServer ≜ ∀X.!(api → X) → !X

Specifies a generic service that:

- Inputs any type (e.g. GMaps).
- Inputs a persistent session that, given an implementation of api, provides the previously input type (e.g. !(api → GMaps)).
- Composes the received session accordingly, subsequently providing a persistent session of the given type (e.g. !GMaps).
Programming using the Logical Interpretation

- Syntax focuses solely on communication behavior.
- No “values” nor functions.
- Hard to write interesting programs.
### A Concurrent Programming Language

#### Overview

**Programming using the Logical Interpretation**
- Syntax focuses solely on communication behavior.
- No “values” nor functions.
- Hard to write interesting programs.

**Towards a Concurrent Programming Language**
- Approximate the syntax of the logical interpretation.
- Introduce recursion for expressiveness.
- Combine concurrent session-typed processes with $\lambda$-calculus.
A Concurrent Programming Language

Overview

Programming using the Logical Interpretation
- Syntax focuses solely on communication behavior.
- No “values” nor functions.
- Hard to write interesting programs.

Towards a Concurrent Programming Language
- Approximate the syntax of the logical interpretation.
- Introduce recursion for expressiveness.
- Combine concurrent session-typed processes with $\lambda$-calculus.

Types
- $\tau \supset A$ and $\tau \land A$ – Input and output of data.
- $\{a_i : A_i \vdash c : A\}$ – Monadic encapsulation of process expressions.
A Concurrent Programming Language
Integrating Concurrency in a Functional Language

Contextual Monad

- Isolate concurrency (and linearity) in a contextual monad.
- Embed into a typical $\lambda$-calculus with recursive types.
- Functional programs can refer to processes and vice-versa:
  - Process layer may send/receive functional values.
  - Monadic objects may be executed (in the process layer).
  - Functions can construct monadic objects.

Example - A Stream of Naturals

We want to produce a stream of naturals, starting at a given number:

```plaintext
stype natStream = nat /\ natStream
init : nat -> {c:natStream}
c <- init n = { output c n
c <- init (n+1) }
```
A Concurrent Programming Language
Integrating Concurrency in a Functional Language

Contextual Monad
- Isolate concurrency (and linearity) in a contextual monad.
- Embed into a typical $\lambda$-calculus with recursive types.
- Functional programs can refer to processes and vice-versa:
  - Process layer may send/receive functional values.
  - Monadic objects may be executed (in the process layer).
  - Functions can construct monadic objects.

Example - A Stream of Naturals
We want to produce a stream of naturals, starting at a given number:

```plaintext
type natStream = nat \times natStream
init : nat -> {c:natStream}
c <- init n = { output c n
c <- init (n+1) }
```
A Concurrent Programming Language
Integrating Concurrency in a Functional Language

Contextual Monad

- Isolate concurrency (and linearity) in a contextual monad.
- Embed into a typical \( \lambda \)-calculus with recursive types.
- Functional programs can refer to processes and vice-versa:
  - Process layer may send/receive functional values.
  - Monadic objects may be executed (in the process layer).
  - Functions can construct monadic objects.

Example - A Stream of Naturals

We want to produce a stream of naturals, starting at a given number:

```plaintext
stype natStream = nat \\ natStream
init : nat -> {c:natStream}
c <- init n = { output c n
    c <- init (n+1) }
```
A Concurrent Programming Language
Integrating Concurrency in a Functional Language

Contextual Monad

- Isolate concurrency (and linearity) in a contextual monad.
- Embed into a typical $\lambda$-calculus with recursive types.
- Functional programs can refer to processes and vice-versa:
  - Process layer may send/receive functional values.
  - Monadic objects may be executed (in the process layer).
  - Functions can construct monadic objects.

Example - A Stream of Naturals

We want to produce a stream of naturals, starting at a given number:

\[
\text{stype natStream = nat} /\!
\text{ natStream}
\]

\[
\text{init : nat -> } \{c:natStream\}
\]

\[
c <= \text{init } n =
\]

\[
\{ \text{ output c n} \\
\text{ c <= init (n+1) } \}
\]
A Concurrent Programming Language

Examples

A Stream Filter

Given a predicate on naturals \( p \), erase from a stream \( d \) all naturals not satisfying the predicate.
A Concurrent Programming Language

Examples

A Stream Filter

Given a predicate on naturals $p$, erase from a stream $d$ all naturals not satisfying the predicate.

$$\text{filter} : (\text{nat} \rightarrow \text{bool}) \rightarrow \{d: \text{natStream} | - c: \text{natStream}\}$$
A Concurrent Programming Language

Examples

A Stream Filter

Given a predicate on naturals \( p \), erase from a stream \( d \) all naturals not satisfying the predicate.

```
filter : (nat -> bool) -> {d:natStream| c:natStream}
c <- filter p <- d =
{ n <- input d
  if (p n) then output c n
    c <- filter p <- d
  else
    c <- filter p <- d }
```
A Concurrent Programming Language

Examples

A Stream Filter

Given a predicate on naturals $p$, erase from a stream $d$ all naturals not satisfying the predicate.

$$\text{filter} : (\text{nat} \to \text{bool}) \to \{d:\text{natStream}|- c:\text{natStream}\}$$

$$c \leftarrow \text{filter} \ p \leftarrow d =$$

$$\{ n \leftarrow \text{input} \ d$$

$$\quad \text{if} \ (p \ n) \ \text{then} \ \text{output} \ c \ n$$

$$\quad \quad c \leftarrow \text{filter} \ p \leftarrow d$$

$$\quad \text{else}$$

$$\quad c \leftarrow \text{filter} \ p \leftarrow d \}$$

A stream of all the even natural numbers:

$$\{c. \ d \leftarrow \text{init} \ 0 ; \ \text{filter} \ (\text{fn} \ x \ => \ x \ \% \ 2 = 0) \leftarrow d\}$$
A Concurrent Programming Language

Examples

**A Stream Filter**

Given a predicate on naturals $p$, erase from a stream $d$ all naturals not satisfying the predicate.

```plaintext
filter : (nat -> bool) -> {d:natStream|- c:natStream}
c <- filter p <- d =
{ n <- input d
  if (p n) then output c n
  c <- filter p <- d
  else
  c <- filter p <- d }
```

A stream of all the even natural numbers:

```plaintext
{c. d <- init 0 ; filter (fn x => x % 2 = 0) <- d}
```
A Concurrent Programming Language

Examples

Sending and Receiving Processes

```haskell
stype AppStore = Choice {weather:
    {d:API, g:GPS |- c:Weather} // 1,
    travel:
    {d:API |- c:Travel} // 1}
```

- `ActivateGPS : unit -> {g:GPS}`
- `WeatherClient : unit -> {a:AppStore, d:API |- c:Weather}`
- `c <- WeatherClient () <- a, d = { a.weather
  w <- input a
  wait a
  g <- ActivateGPS ()
  c <- w <- d, g }`

Deadlock-freedom/Global Progress in the presence of HO mobility.
Sending and Receiving Processes

\[
\text{stype AppStore} = \text{Choice}\{\text{weather}: \{d:API, g:GPS \mid c:\text{Weather}\} \setminus 1, \\
\text{travel}: \{d:API \mid c:\text{Travel}\} \setminus 1\}
\]

\[
\text{ActivateGPS} : \text{unit} \to \{g:\text{GPS}\}
\]

\[
\text{WeatherClient} : \text{unit} \to \{a:\text{AppStore}, d:API \mid c:\text{Weather}\}
\]

\[
c \leftarrow \text{WeatherClient} () \leftarrow a, d = \{ a.\text{weather} \\
\quad \quad \quad w \leftarrow \text{input} a \\
\quad \quad \quad \text{wait} a \\
\quad \quad \quad g \leftarrow \text{ActivateGPS} () \\
\quad \quad \quad c \leftarrow w \leftarrow d, g \}
\]
A Concurrent Programming Language
Examples

Sending and Receiving Processes

\[
\text{stype AppStore} = \text{Choice}\{\text{weather:}\}
\]
\[
\{d:API, g:GPS \mid - c:Weather\} \setminus 1,
\]
\[
\text{travel:}
\]
\[
\{d:API \mid - c:Travel\} \setminus 1\}
\]

\[
\text{ActivateGPS : unit} \rightarrow \{g:GPS\}
\]

\[
\text{WeatherClient : unit} \rightarrow \{a:AppStore, d:API \mid - c:Weather\}
\]
\[
c \leftarrow \text{WeatherClient} () \leftarrow a, d = \{ a.\text{weather}
\]
\[
w \leftarrow \text{input} a
\]
\[
\text{wait a}
\]
\[
g \leftarrow \text{ActivateGPS} ()
\]
\[
c \leftarrow w \leftarrow d, g\}
\]

Deadlock-freedom/Global Progress in the presence of HO mobility.
Recursive types introduce divergence.

Logical soundness is therefore lost.

What is the behavior of:

\[ \{ c. \ d \leftarrow \text{init} \ 0 \ ; \ \text{filter} \ (\text{fn} \ x \Rightarrow \text{false}) \leftarrow d \} \]
A Concurrent Programming Language
Reconciling with Logic

Divergence
- Recursive types introduce divergence.
- Logical soundness is therefore lost.
- What is the behavior of:

  \{ c. d <- init 0 ; filter (fn x => false) <- d \}

Recovering Non-Divergence [Toninho et al. 14]
- Restrict to coinductive types.
- Only allow productive definitions:
  - Process definitions must be \textit{guarded}.
  - Self-interaction with recursive calls must be disallowed.
Reasoning using Logic

- Up to this point we have identified concurrent “features” that can be justified logically.
- Can we do more? Can we use the logical foundation to reason about concurrent programs?
Reasoning using Logic

- Up to this point we have identified concurrent “features” that can be justified logically.
- Can we do more? Can we use the logical foundation to reason about concurrent programs?

Properties of Interest

- Does a (well-typed) concurrent program diverge?
- When do two programs have the same behavior?
- What are the behavioral consequences of polymorphism?
- When are different behaviors compatible?
Linear Logical Relations [Pérez et al. 12, Caires et al. 13, Toninho et al. 14]

- Based on logical relations for $\lambda$-calculus.
- Uniform framework for our session-typed processes.
- Scales to handle polymorphism and coinductive types.
- Allow us to show that well-typed processes are compositionally non-divergent.
Process Equivalence

- Observational equivalence is a fundamental tool for reasoning about processes.
- “Canonical” observational equivalence: Barbed Congruence.
- “Hard” to reason about – Quantification over all contexts.
## Process Equivalence
- Observational equivalence is a fundamental tool for reasoning about processes.
- “Canonical” observational equivalence: Barbed Congruence.
- “Hard” to reason about – Quantification over all contexts.

## Logical Equivalence
- Relational extension of unary formulation of LRs.
- Contextual typed bisimulation – Logical Equivalence.
- “Easy” to reason about.
- Turns out to be a proof technique for barbed congruence.
For open processes, we must place them in a (typed) closed context.

- Actions along $x$ and $z$ will not be visible, only those along $e$.
- Prefix permutation is safe since actions are not observable.
Recall our CloudServer example and consider:

$$\text{CloudServer} \triangleq \forall X. !(\text{api} \rightarrow X) \rightarrow !X$$

- A restaurant finding service rest, relying on some maps app.
- Both $C_1$ and $C_2$ will use the CloudServer for its maps infrastructure to offer service rest.
- One wishes to use GMaps, the other AMaps (related but not quite the same).
Recall our CloudServer example and consider:

\[
\text{CloudServer} \triangleq \forall X. \!(\text{api} \circ X) \circ \!X
\]

- A restaurant finding service \text{rest}, relying on some maps app.
- Both \(C_1\) and \(C_2\) will use the CloudServer for its maps infrastructure to offer service \text{rest}.
- One wishes to use GMaps, the other AMaps (related but not quite the same).

\[
s:!(\text{api} \circ X) \circ \!X \Rightarrow C_1 \simeq_{\perp} C_2 :: z:\text{rest}[X \leftrightarrow \text{GMaps} \leftrightarrow X \leftrightarrow \text{AMaps}]
\]

Regardless of the impl. of the CloudServer, parametricity ensures it behaves uniformly.
Reasoning Techniques
Linear Logical Relations – Equivalence

Session Type Isomorphisms

Two session behaviors \( A \) and \( B \) may be different, but compatible…

- Can we construct \( A \) from \( B \) and vice-versa?
- Can we do it “seamlessly”?
- i.e. Are \( A \) and \( B \) isomorphic (written \( A \simeq B \))?
Reasoning Techniques
Linear Logical Relations – Equivalence

Session Type Isomorphisms

Two session behaviors $A$ and $B$ may be different, but compatible…
- Can we construct $A$ from $B$ and vice-versa?
- Can we do it “seamlessly”?
- i.e. Are $A$ and $B$ isomorphic (written $A \simeq B$)?

Isomorphisms

$$A \otimes B \simeq B \otimes A$$

We have $P^{\langle x,y \rangle}$ and $Q^{\langle y,x \rangle}$ such that:

1. $x:A \otimes B \Rightarrow P^{\langle x,y \rangle} :: y:B \otimes A$ and $y:B \otimes A \Rightarrow Q^{\langle y,z \rangle} :: z:A \otimes B$
2. $x:A \otimes B \Rightarrow (\nu y)(P^{\langle x,y \rangle} | Q^{\langle y,z \rangle}) \approx_{\perp} [x \leftrightarrow z] :: z:A \otimes B$
Isomorphisms

- $A \otimes B \simeq B \otimes A$
- $A \multimap (B \multimap C) \simeq (A \otimes B) \multimap C$
- $(A \oplus B) \multimap C \simeq (A \multimap C) \& (B \multimap C)$
- $\ldots$

Coercions can mediate between the two types, representing isomorphic behavior.
## Summary of Contributions

- **Logical foundation of session types using linear logic:**
  - Concurrent language features explained logically.
  - Logical features explained using concurrency.

- **A concurrent programming language:**
  - Monadic integration of session-based concurrency.
  - General recursion and (re)connections with logic.
  - Deadlock-freedom in a higher-order setting.

- **Reasoning techniques:**
  - Non-trivial properties (e.g. Termination, Parametricity).
  - Logical equivalence as behavioral equivalence.
  - Session Type Isomorphisms
Open Problems / Future Work

- Logical equivalence in the higher-order setting.
- Full dependent types.
- Explicit distribution concerns.
- Non-determinism.
- ...
B. Toninho, L. Caires, F. Pfenning. *Dependent Session Types via Intuitionistic Linear Type Theory*. PPDP’11


B. Toninho, L. Caires, F. Pfenning. *Functions as Session-Typed Processes*. FoSSaCS’12


A Logical Foundation for Session-based Concurrent Computation

Bernardo Toninho
[Co-advised by: Luís Caires and Frank Pfenning]

Computer Science Department
Carnegie Mellon University
Universidade Nova de Lisboa

Thesis Defense