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 Proposal
Motivation
Concurrency and Distribution

Concurrent and Distributed Systems

- Ubiquitous: From smart phone apps to business infrastructures.
- Building these systems is hard.
- Building these systems *correctly* is harder.
- *Reasoning* about these systems is even harder.
Motivation
Concurrency and Distribution

Concurrent and Distributed Systems
- Ubiquitous: From smart phone apps to business infrastructures.
- Building these systems is hard.
- Building these systems correctly is harder.
- Reasoning about these systems is even harder.

Why is it hard?
- New problems arise: deadlocks, livelocks, security, etc.
- Compositional reasoning is generally hard to do in this setting (if possible).
- Testing isn’t a reasonable approach (conditions hard to replicate).
Correctness - What does it mean?

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

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

How do we reason about these systems?
- Language-based models (e.g. Process Calculi)
- Logics (e.g. HML, Separation Logic)
- Type Systems (e.g. I/O Types, Session Types)
## Process Calculi

- Algebraic model of concurrent message-passing processes.
- Enable study of behavior and interaction of systems.
- General model of computation (e.g. $\pi$-calculus).
- 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.
- General model of computation (e.g. $\pi$-calculus).
- Enriched with types to enforce properties on processes.

Types and the $\pi$-calculus

- 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.
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.
Outline

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

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

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

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
- Parametricity
- Dependent Types
## 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.

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.
## Deviating from SILL

- Explicit linear forwarders.
- Syntax-driven assignment.
- Syntax as a basis for a concurrent, session-typed language.
- Consistently mapped to $\pi$-calculus processes.
Linear Logic and Message-Passing Concurrency
Logical Interpretation

Deviating from SILL

- Explicit linear forwarders.
- Syntax-driven assignment.
- Syntax as a basis for a concurrent, session-typed language.
- Consistently mapped to $\pi$-calculus processes.

Propositions as Types

- $A \otimes B$: Output a session of type $A$ and continue as $B$
- $A \rightarrow B$: Input a session of type $A$ and continue as $B$
- $A \& B$: Offer a choice between a session of type $A$ or $B$
- $A \oplus B$: Select a session of type $A$ or $B$
- $!A$: A persistent session of type $A$
- $1$: Terminated session
Typing Judgment

\[ \Gamma \vdash A_1, \ldots, A_m ; \quad A_1, \ldots, A_n \vdash A \]
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

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

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

Cut as Composition

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

\[ \frac{}{\Gamma; \Delta \vdash P :: x : A} \]

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

Cut as Composition

\[ \frac{}{\Gamma; \Delta, \Delta' \vdash C \quad \Gamma; \Delta', A \vdash C} \]

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

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

\[
\begin{align*}
\Gamma; \Delta \vdash P :: x : A & \quad \Gamma; \Delta', x : A \vdash Q :: z : C \\
\Gamma; \Delta, \Delta' \vdash \text{new } x. (P \mid Q) :: z : C
\end{align*}
\]

\text{cut}
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

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

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

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

Identity as Forwarding

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

\[ A \]

Bernardo Toninho (UNL&CMU)  Thesis Proposal
Linear Logic and Message-Passing Concurrency

Logical Interpretation - Judgmental Principles

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

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

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

Identity as Forwarding

\[
\frac{}{\Gamma; x:A \vdash \text{id} A}
\]
### Typing Judgment

\[ \Gamma \vdash P :: x:A \quad \Delta \vdash x: A_1, \ldots, x_n: A_n \]

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

### Cut as Composition

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

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

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

### Identity as Forwarding

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

\( \text{id} \)
Linear Logic and Message-Passing Concurrency

Logical Interpretation - Tensor as Output

Session Output

\[
\Gamma; \Delta \vdash A \quad \Gamma; \Delta' \vdash B \\
\Gamma; \Delta, \Delta' \vdash A \otimes B \quad R \;
\]
Session Output

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

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

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

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

\[
\Gamma; \delta, A, B \vdash C
\]

\[
\Gamma; \delta, A \otimes B \vdash C
\]

\[
\otimes R
\]

\[
\otimes L
\]
Session Output

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

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

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

\[
\Gamma; \Delta, y : A, x : B \vdash R :: z:C
\]

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

**Session Output**

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

\[
\begin{align*}
\Gamma; \Delta, y : A, x : B &\vdash R :: z : C \\
\Gamma; \Delta, x : A \otimes B &\vdash y \leftarrow \text{input } x; \ R :: z : C
\end{align*}
\]

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

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

Proof Reduction

\[
\begin{align*}
\Gamma; \Delta, \Delta' \vdash \text{new } x.((\text{output } x (y.P); Q) | y \leftarrow \text{input } x; R) :: z : C \\
\rightarrow \quad \Gamma; \Delta, \Delta' \vdash \text{new } x.(Q | \text{new } y.(P | R)) :: z : C
\end{align*}
\]
Linear Logic and Message-Passing Concurrency

Logical Interpretation - Implication as Input

Session Input

\[
\frac{\Gamma; \Delta, \quad A \vdash B}{\Gamma; \Delta \vdash A \Rightarrow B} \quad \text{R}
\]
Logical Interpretation - Implication as Input

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

\[ \Gamma; \Delta \vdash A \rightarrow B \quad \rightarrow \neg R \]
Session Input

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$\Gamma; \Delta \vdash x \leftarrow \text{input } z; P :: z : A \multimap B$
Session Input

\[
\begin{align*}
\Gamma; \Delta, x : A &\vdash P :: z:B \\
\Gamma; \Delta &\vdash x \leftarrow \text{input } z; P :: z:A \multimap B &\quad \to R \\
\Gamma; \Delta &\vdash Q_1 :: y:A \\
\Gamma; \Delta', x:B &\vdash Q_2 :: z:C
\end{align*}
\]

\[
\begin{align*}
\Gamma; \Delta, \Delta', A \multimap B &\vdash C &\quad \to L
\end{align*}
\]
**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 \text{\(-\circ R\)}
\]

\[
\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 \text{\(-\circ L\)}
\]
Linear Logic and Message-Passing Concurrency

Logical Interpretation - Implication as Input

Session Input

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

\[\rightarrow_R \quad \rightarrow_L\]

Linear Implication as input. Reduction is the same as for \( \otimes \).
Linear Logic and Message-Passing Concurrency
Logical Interpretation - Multiplicative Unit as Termination

Multiplicative Unit

\[ \Gamma; \cdot \vdash 1 \quad 1^R \]
Multiplicative Unit

\[ \Gamma; \cdot \vdash \text{close } z :: z:1 \quad \text{1R} \]
Linear Logic and Message-Passing Concurrency

Logical Interpretation - Multiplicative Unit as Termination

**Multiplicative Unit**

\[
\frac{\Gamma; \cdot \vdash \mathit{close} \, z :: z:1}{1R}
\]

\[
\frac{\Gamma; \Delta \vdash C}{1L}
\]

\[
\frac{\Gamma; \Delta \vdash C}{C}
\]

\[
\frac{\Gamma; \cdot \vdash \mathit{close} \, z :: z:1}{1R}
\]

\[
\frac{\Gamma; \Delta \vdash C}{1L}
\]

\[
\frac{\Gamma; \cdot \vdash \mathit{close} \, z :: z:1}{1R}
\]
Linear Logic and Message-Passing Concurrency
Logical Interpretation - Multiplicative Unit as Termination

Multiplicative Unit

\[ \frac{}{\Gamma; \cdot \vdash \text{close } z :: z:1} 1R \]
\[ \frac{\Gamma; \Delta \vdash Q :: z:C}{\Gamma; \Delta, 1 \vdash C} 1L \]
**Linear Logic and Message-Passing Concurrency**

**Logical Interpretation - Multiplicative Unit as Termination**

**Multiplicative Unit**

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

**Proof Reduction**

\[
\Gamma; \Delta \vdash \text{new } x . (\text{close } x \mid (\text{wait } x ; Q)) :: z : C
\]
Linear Logic and Message-Passing Concurrency
Logical Interpretation - Multiplicative Unit as Termination

**Multiplicative Unit**

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

**Proof Reduction**

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

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

\[
\Gamma; \Delta \vdash A \quad \Gamma; \Delta \vdash B \\
\Gamma; \Delta \vdash A \& B \quad \&R
\]
Additive Conjunction

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

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

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

\&^R

\&^L_1
Additive Conjunction

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

\[ \Gamma; \Delta, x:A \vdash Q :: z:C \]
\[ \Gamma; \Delta, A \& B \vdash C \quad \&L_1 \]
Additive Conjunction

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

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

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

$$\Gamma; \Delta, x:A \& B \vdash x.\text{inl}; Q :: z:C$$ \quad &L_1
Additive Conjunction

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

Proof Reduction

\[
\Gamma; \Delta, \Delta' \vdash \text{new } x.(x.\text{case}(P_1, P_2) \mid x.\text{inl}; Q) :: z:C \\
\rightarrow \Gamma; \Delta, \Delta' \vdash \text{new } x.(P_1 \mid Q) :: z:C
\]
Additive Conjunction

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

\[
\begin{align*}
\Gamma; \Delta, x: A \vdash Q :: z: C \\
\Gamma; \Delta, x: A \& B \vdash x.\text{inl}; Q :: z: C \quad & \& L_1
\end{align*}
\]

Proof Reduction

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

Additive disjunction \(\oplus\) is dual.
Persistent Cut

\[
\frac{\Gamma; \cdot \vdash A \quad \Gamma, A; \Delta \vdash C}{\Gamma; \Delta \vdash C} \text{ cut}^I
\]
**Persistent Cut**

\[
\frac{\Gamma; \cdot \vdash P :: x:A \quad \Gamma, u:A; \Delta \vdash Q :: z:C}{\Gamma; \Delta \vdash C \quad \text{cut}^!}
\]
Persistent Cut

\[
\frac{\Gamma; \cdot \vdash P :: x:A \quad \Gamma, u:A; \Delta \vdash Q :: z:C}{\Gamma; \Delta \vdash \text{new } u.(x \leftarrow !\text{input } u \; P \mid Q) :: z:C} \quad \text{cut}^I
\]
**Persistent Cut**

\[
\begin{align*}
\Gamma; \cdot \vdash P :: x:A & \quad \Gamma, u:A; \Delta \vdash Q :: z:C \\
\Gamma; \Delta \vdash \text{new } u.(x \leftarrow \text{!input } u \ P \mid Q) :: z:C
\end{align*}
\]

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

\[
\frac{\Gamma; \cdot \vdash P :: x:A \quad \Gamma, u:A; \Delta \vdash Q :: z:C}{\Gamma; \Delta \vdash \text{new } u.(x \leftarrow \text{!input } u \ P \mid Q) :: z:C} \quad \text{cut}^! 
\]

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

Copy

\[
\frac{\Gamma, A; \Delta, A \vdash \text{C}}{\Gamma, A; \Delta \vdash \text{C}} \quad \text{copy} 
\]
### Persistent Cut

\[
\frac{\Gamma; \cdot \vdash P :: x:A \quad \Gamma, \ u:A; \Delta \vdash Q :: z:C}{\Gamma; \Delta \vdash \text{new } u.\ (x \leftarrow \text{!input } u \ P \mid Q) :: z:C} \quad \text{cut}'
\]

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

### Copy

\[
\frac{\Gamma, \ u:A; \Delta, \ x:A \vdash R :: z:C}{\Gamma, \ A; \Delta \vdash C} \quad \text{copy}
\]
Linear Logic and Message-Passing Concurrency

Logical Interpretation - Persistent Resources

### Persistent Cut

\[
\frac{\Gamma; \cdot \vdash P :: x:A \quad \Gamma, u:A; \Delta \vdash Q :: z:C}{\Gamma; \Delta \vdash \text{new } u.(x \leftarrow \text{!input } u \ P \mid Q) :: z:C} \quad \text{cut}^! 
\]

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

### Copy

\[
\frac{\Gamma, u:A; \Delta, x:A \vdash R :: z:C}{\Gamma, u:A; \Delta \vdash x \leftarrow \text{copy } u; R :: z:C} \quad \text{copy}
\]
Linear Logic and Message-Passing Concurrency

Logical Interpretation - Persistent Resources

**Persistent Cut**

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

\[
\Gamma; \Delta \vdash \text{new } u. (x \leftarrow !\text{input } u P \mid Q) :: z : C \quad \text{cut}^1
\]

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

**Copy**

\[
\Gamma, u : A; \Delta, x : A \vdash R :: z : C
\]

\[
\Gamma, u : A; \Delta \vdash x \leftarrow \text{copy } u; R :: z : C \quad \text{copy}
\]

**Proof Reduction**

\[
\Gamma; \Delta \vdash \text{new } u. (x \leftarrow !\text{input } u P \mid (x \leftarrow \text{copy } u; Q)) :: z : C
\]

\[
\rightarrow \Gamma; \Delta \vdash \text{new } u. (x \leftarrow !\text{input } u P \mid (\text{new } x.(P \mid Q))) :: z : C
\]
Exponential

\[
\frac{\Gamma; \cdot \vdash A}{\Gamma; \cdot \vdash !A !R}
\]
Exponential

\[ \Gamma; \cdot \vdash P :: x:A \]
\[ \Gamma; \cdot \vdash !A \]
\[ \Gamma; \cdot \vdash !R \]
Exponential

\[
\frac{\Gamma; \cdot \vdash P :: x: A}{\Gamma; \cdot \vdash \text{output } z ! (x.P) :: z: ! A} \quad !R
\]
Exponential

\[
\Gamma; \cdot \vdash P :: x : A \quad \frac{!R}{\Gamma; \cdot \vdash \text{output } z ! (x.P) :: z : !A}
\]

\[
\frac{\Gamma; \Delta, \cdot \vdash \text{output } z ! (x.P) :: z : !A}{!L}
\]

\[
\frac{\Gamma, \cdot \vdash \text{output } z ! (x.P) :: z : !A}{\Gamma; \Delta, \cdot \vdash C}
\]
Exponential

\[
\begin{align*}
\Gamma; \cdot & \vdash P :: x:A \\
\Gamma; \cdot & \vdash \text{output } z \,(! (x.P)) :: z:A
\end{align*}
\]

\[!R\quad \frac{\Gamma; \cdot \vdash P :: x:A}{\Gamma; \cdot \vdash \text{output } z \,(! (x.P)) :: z:A} \quad \frac{\Gamma, u:A; \Delta \vdash Q :: z:C}{\Gamma; \Delta, u:A \vdash C} \quad !L
\]
Exponential

\[
\Gamma; \cdot \vdash P :: x:A \\
\Gamma; \cdot \vdash \text{output } z \,(x.P) :: z!:A
\]

\( \Gamma; \cdot \vdash \text{output } z \,(x.P) :: z!:A \)

\[
\Gamma, u:A; \Delta \vdash Q :: z:C \\
\Gamma, u:A; \Delta \vdash u \leftarrow \text{input } x; Q :: z:C
\]

\( \Gamma, u:A; \Delta \vdash u \leftarrow \text{input } x; Q :: z:C \)

Proof reduction transforms a cut into a cut\(^!\)
Linear Logic and Message-Passing Concurrency

Properties

**Type Preservation**

If $\Gamma; \Delta \vdash P :: z:A$ and $P \rightarrow P'$ then $\Gamma; \Delta \vdash P' :: z:A$

Global Progress

If $\cdot; \cdot \vdash P :: z:1$ then either $P = \text{new } x.(!Q; \text{close } z)$ or $P \rightarrow P_0$

The logical basis gives us deadlock-freedom and session fidelity "for free."
Linear Logic and Message-Passing Concurrency

Properties

Type Preservation

If $\Gamma; \Delta \vdash P :: z:A$ and $P \rightarrow P'$ then $\Gamma; \Delta \vdash P' :: z:A$

Global Progress

If $\cdot; \cdot \vdash P :: z:1$ then either $P = \text{new } \bar{x}.(!Q; \text{close } z)$ or $P \rightarrow P'$

The logical basis gives us deadlock-freedom and session fidelity “for free”.
Limitations of 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 quite challenging.
Limitations of 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 quite 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.
Two new types: \( \forall x : \tau \cdot A \) and \( \exists x : \tau \cdot A \)

Parametric in the language of types \( \tau \).

\( \forall x : \tau \cdot A \) - Input a term \( M : \tau \), continue as \( A(M) \).

\( \exists x : \tau \cdot A \) - Output a term \( M : \tau \), continue as \( A(M) \).

If \( \tau \)'s are dependent: proof communication.

A Simple Example

\[
\text{indexer}_1 \triangleq \forall f : \text{file.pdf}(f) \to \exists g : \text{file.pdf}(g) \otimes 1
\]

\[
\text{indexer}_2 \triangleq \forall f : \text{file.pdf}(f) \to \exists g : \text{file.pdf}(g) \otimes \text{agree}(f, g) \otimes 1
\]
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

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

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

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

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

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

\( \psi; \Gamma; \Delta, \quad \forall x: \tau. A \vdash \)

\[ C \quad \forall L \]

\[ C \quad \forall R \]
Extend the Judgment

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

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

Universal Quantification

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

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

\[ \forall R \]

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

\[ \forall L \]

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

\[ C \]

\[ C \]
Extend the Judgment

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

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

Universal Quantification

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

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

\( \psi; \Gamma; \Delta \vdash P : z : A \)

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

Universal Quantification

\[
\begin{align*}
\psi, x : \tau; \Gamma; \Delta & \vdash P : z: A \\
\psi; \Gamma; \Delta & \vdash x \leftarrow \text{input } z; P : z: \forall x : \tau. A \\
\psi & \vdash M : \tau \\
\psi; \Gamma; \Delta, x : A\{M/x\} & \vdash Q : z : C \\
\psi; \Gamma; \Delta, x : \forall x : \tau. A & \vdash \text{output } x M; Q : z : C
\end{align*}
\]
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 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

$\text{Indexer} \triangleq \forall f:\text{file}. \forall p:\text{pdf}(f). \exists g:\text{file}. \exists q_1:\text{pdf}(g). \exists q_2:\diamond_! [\text{agree}(f, g)] \otimes 1$
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

$$\text{Indexer} \triangleq \forall f:\text{file}. \forall p:\text{pdf}(f). \exists g:\text{file}. \exists q_1:\text{pdf}(g). \exists q_2: \diamond I[\text{agree}(f, g)] \otimes 1$$

$\diamond I[\text{agree}(f, g)]$ is a signed certificate of the agreement of $f$ and $g$. 
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)

\[
\frac{\Omega, \, \Lambda; \, \Gamma; \, \Delta \vdash \, A}{\Omega; \, \Gamma; \, \Delta \vdash \, \forall X. A} \quad \forall R2
\]
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)

\[
\Omega, X; \Gamma; \Delta \vdash P :: z : A \\
\Omega; \Gamma; \Delta \vdash \forall X.A \quad \forall R2
\]
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)

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

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

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

\(\forall R2\)

\[
\Omega \vdash B \text{ type} \\
\Omega; \Gamma; \Delta, \quad A\{B/X\} \vdash C \\
\Omega; \Gamma; \Delta, \quad \forall X. A \vdash C
\]

\(\forall L2\)
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)

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

\[
\frac{\Omega \vdash B \text{ type} \quad \Omega; \Gamma; \Delta, x:A\{B/X\} \vdash Q :: z:C}{\Omega; \Gamma; \Delta, \forall X.A \vdash C} \quad \forall L2
\]
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)

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

\[
\begin{align*}
\Omega &\vdash B \text{ type} \\
\Omega; \Gamma; \Delta; x:A\{B/X\} &\vdash Q :: z:C \\
\Omega; \Gamma; \Delta &\vdash x: \forall X.A \leftarrow \text{output } x \ B; Q :: z:C \\
\quad \quad \quad \forall L2
\end{align*}
\]
CloudServer $\triangleq \forall X. !\text{(api } \rightsquigarrow X \leftsquigarrow !X)$
CloudServer ⩵ ∀X.!(api → X) → !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 \text{api}, provides the previously input type (e.g. !(\text{api} \rightarrow \text{GMaps})).
Parametric Polymorphism

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

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 $\to$ GMaps)).
- Composes the received session accordingly, subsequently providing a persistent session of the given type (e.g. !GMaps).
Other concurrency-theoretic features that can be understood logically (and vice-versa):

- Asynchronous communication [DeYoung et al. 12]
- Concurrent evaluation strategies for λ-calculus [Toninho et al. 12]
- Session type isomorphisms [Pérez et al. 12]
### Programming with Processes

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

Overview

Programming with Processes

- 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 with Processes
- 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.
- $\{a_i:A_i \vdash c:A\}$ – Encapsulation of open process expressions.
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.
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.

Typing Rules

\[
\begin{align*}
\Delta &= \text{lin}(a_i:A_i) \quad \Gamma = \text{shd}(a_i:A_i) \\
\psi; \Gamma; \Delta &\vdash P :: c:A \\
\psi \vdash c \leftarrow \{P_{c,a_i}\} &\leftarrow \overline{a_i:A_i} : \{\overline{a_i:A_i} \vdash c:A\}
\end{align*}
\]
Contextual Monad

- Isolate concurrency (and linearity) in a contextual monad.
- Embed into a typical λ-calculus with recursive types.
- Functional programs can refer to processes and vice-versa.

Typing Rules

\[
\begin{align*}
\Delta &= \text{lin}(a_i:A_i) \quad \Gamma = \text{shd}(a_i:A_i) \\
&\quad \vdash \Psi; \Gamma; \Delta \vdash P : c:A \\
&\quad \vdash \Psi \vdash c \leftarrow \{P_{c,a_i}\} \leftarrow a_i:A_i : \{a_i:A_i \vdash c:A\} \\
\end{align*}
\]

\[
\begin{align*}
\Delta &= \text{lin}(a_i:A_i) \quad \Gamma \supset \text{shd}(a_i:A_i) \\
&\quad \vdash \Psi \vdash M : \{a_i:A_i \vdash c:A\} \\
&\quad \vdash \Psi; \Gamma; \Delta', c:A \vdash Q_c : d:D \\
&\quad \vdash \Psi; \Gamma; \Delta, \Delta' \vdash c \leftarrow M \leftarrow a_i; Q_c : d:D \\
\end{align*}
\]
Example

Stream Session

- Output of an infinite sequence of naturals, counting up.
- A coinductive session (as a recursive type):

  \[
  \text{stype natStream = nat } \backslash\backslash \text{ intStream}
  \]
A Concurrent Programming Language

Example

Stream Session

- Output of an infinite sequence of naturals, counting up.
- A coinductive session (as a recursive type):

\[ \text{stype natStream} = \text{nat} \lor \text{intStream} \]

Stream Transducer

To write a recursive process, we write a recursive function:

\[
\text{trans : } (\text{nat} -> \text{nat}) -> \\
\{ \text{d:natStream |- c:natStream} \}
\]

\[
c \leftarrow \text{trans } f = \\
\{ \text{n1 <- input } d \\
\text{output } c \ (f \ n1) \\
c' \leftarrow \text{trans } f \leftarrow d \\
fwd \ c \ c' \} 
\]
Recursive types introduce divergence.

Logical soundness is therefore lost.
Divergence

- Recursive types introduce divergence.
- Logical soundness is therefore lost.

Recovering Non-Divergence [Toninho et al. 14]

- Restrict to inductive and coinductive types.
- Only allow productive definitions:
  - Process definitions must be *guarded*.
  - Self-interaction with recursive calls must be disallowed.
Recursive types introduce divergence. Logical soundness is therefore lost.

Restrict to inductive and coinductive types. Only allow productive definitions:
- Process definitions must be *guarded*.
- Self-interaction with recursive calls must be disallowed.

\[ \text{c} \leftarrow \text{P} = \{ \text{output c 0 ; c' } \leftarrow \text{P ; x } \leftarrow \text{input c' ; fwd c c'} \} \]

Loops after first output...
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?

Logical Relations

- Standard PL technique for $\lambda$-calculus.
- Mostly unexplored for typed process calculi.
- Useful for showing non-trivial properties (e.g. termination, confluence, etc.).
Linear Logical Relations [Pérez et al. 12, Caires et al. 13, Toninho et al. 14]

- Uniform framework for our session-typed processes.
- Scales to handle polymorphism and coinductive types.
Reasoning Techniques

Linear Logical Relations

Linear Logical Relations [Pérez et al. 12, Caires et al. 13, Toninho et al. 14]

- Uniform framework for our session-typed processes.
- Scales to handle polymorphism and coinductive types.

Informal Idea - Unary Case

- Candidates – Set of processes satisfying the relevant property.
- Inductive on typing: $\mathcal{L}[\Gamma; \Delta \vdash T]$
- Base case is inductive on r.h.s types:
  - $P \in \mathcal{L}[z:A \rightarrow B] \triangleq$ if $P \xrightarrow{z(y)} P'$ then $\forall Q \in \mathcal{L}[y:A].(\nu y)(P' | Q) \in \mathcal{L}[z:B]$
  - ...
- Show that well-typed processes are in the predicate.
Equivalence is a fundamental tool for reasoning about processes. “Canonical” observational equivalence: Barbed Congruence. “Hard” to reason about – Quantification over all contexts.
Reasoning Techniques
Linear Logical Relations – Equivalence

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

**Logical Equivalence**
- Natural extension of unary predicate.
- Contextual typed bisimulation – Logical Equivalence.
- “Easy” to reason about.
- Logical Equivalence vs Barbed Congruence:
  - Propositional: Sound and Complete
  - Polymorphic: Sound and Complete
  - Higher-order: ???
(Functional) Type Theory

- Types depend on terms.
- Uniform integration of programming and reasoning.
- Truly, *Proofs as Programs*.
(Functional) Type Theory

- Types depend on terms.
- Uniform integration of programming and reasoning.
- Truly, *Proofs as Programs*.

Towards Dependent Session Types

- Value dependencies are insufficient.
- Separation of data and behavior – only specify properties of data.
- Monadic language “closes the gap”.
- Restrict to logically sound fragment.
- Coinduction is crucial when reasoning about processes.
Reasoning Techniques
Dependent Session Types

Challenges

- Understanding definitional equality.
- Rich enough for practical proofs? Decidable?
- Adequacy

Potential Outcome – Coinductive Proofs as Processes

stype bisim P Q = ...

S1 : nat -> {c:natStream}
S2 : nat -> {c:natStream}

bisim_proof : Pi n:nat.{c:bisim (S1 n) (S2 n)}
c <- bisim_proof n = { ... }
Conclusion

Summary

- 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.

- Reasoning techniques:
  - Non-trivial properties (e.g. Termination, Parametricity).
  - Behavioral equivalence.
Conclusion

Summary

- 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.
- Reasoning techniques:
  - Non-trivial properties (e.g., Termination, Parametricity).
  - Behavioral equivalence.

Research Plan (Approx. 6 months)

- Study behavioral theory in HO setting.
- Definitional equality and dependent session types (speculative).
- Develop examples showcasing expressiveness.
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 Proposal