Session-Typed Concurrent Programming
Lecture 2

Stephanie Balzer
Carnegie Mellon University

OPLSS 2021
June 24, 2021
Recap
Recap

• Roadmap and learning objectives
Recap

- Roadmap and learning objectives
- Message-passing concurrent programming
  - pi-calculus as formal model
  - nondeterminism
Recap

• Roadmap and learning objectives

• Message-passing concurrent programming
  • pi-calculus as formal model
  • nondeterminism

• Session types as types of message-passing concurrency
  • challenge: preservation because type changes with protocol
  • strategies: (a) disallow aliasing or (b) control aliasing
Recap

• Roadmap and learning objectives

• Message-passing concurrent programming
  • pi-calculus as formal model
  • nondeterminism

• Session types as types of message-passing concurrency
  • challenge: preservation because type changes with protocol
  • strategies: (a) disallow aliasing or (b) control aliasing

• Intuitionistic linear logic as a foundation for session types
Recap

- Roadmap and learning objectives
- Message-passing concurrent programming
  - pi-calculus as formal model
  - nondeterminism
- Session types as types of message-passing concurrency
  - challenge: preservation because type changes with protocol
  - strategies: (a) disallow aliasing or (b) control aliasing
- Intuitionistic linear logic as a foundation for session types

we’ll resume here
Intuitionistic linear logic session types
Intuitionistic linear logic session types

Types:

\[ A, B \quad \triangleq \quad A \otimes B \quad \text{multiplicative conjunction} \quad \text{“channel output”} \]
\[ A \rightarrow B \quad \text{multiplicative implication} \quad \text{“channel input”} \]
\[ A \& B \quad \text{additive conjunction} \quad \text{“external choice”} \]
\[ A \oplus B \quad \text{additive disjunction} \quad \text{“internal choice”} \]
\[ 1 \quad \text{unit for } \otimes \quad \text{“termination”} \]

Queue session type:

\[
\text{queue } A = \&\{ \text{enq} : A \rightarrow \text{queue } A, \\
\text{deq} : \oplus\{ \text{none} : 1, \text{some} : A \otimes \text{queue } A \}\}
\]
Typing judgment and rules

Intuitionistic linear sequent:

\[ x_1 : A_1, \ldots, x_n : A_n \vdash P :: (x : A) \]

“Process P offers a session of type A along channel x using session \( A_1, \ldots, A_n \) provided along channels \( x_1, \ldots, x_n \).”

Inference rule:

\[
\begin{align*}
\text{premise} & : \Delta' \vdash Q :: (x : A') \\
\text{conclusion} & : \Delta \vdash P; Q :: (x : A)
\end{align*}
\]

Left and right rules:

\[
\begin{align*}
\Delta', \ x : B & \vdash Q :: (z : C) \\
\Delta, \ x : A \bowtie B & \vdash P; Q :: (z : C) \quad \diamond_L \\
\Delta' & \vdash Q :: (x : B) \\
\Delta & \vdash P; Q :: (x : A \bowtie B) \quad \diamond_R
\end{align*}
\]
Connectives so far

\[
\begin{align*}
\Delta \vdash P : (x : B) & \quad \Delta, x : A \vdash \text{send } x y ; P : (x : A \otimes B) & \otimes_R \\
\Delta, y : A \vdash P_y : (x : B) & \quad \Delta, x : A \otimes B \vdash y \leftarrow \text{recv } x ; Q_y : (z : C) & \otimes_L \\
\Delta \vdash y \leftarrow \text{recv } x ; P_y : (x : A \multimap B) & \quad \Delta, x : A \multimap B \vdash \text{send } x y ; Q : (z : C) & \multimap_L \\
\Delta \vdash P : (x : A) & \quad \Delta \vdash x \cdot \text{inl}; P : (x : A \oplus B) & \oplus_{R_1} \\
\Delta \vdash P : (x : B) & \quad \Delta \vdash x \cdot \text{inr}; P : (x : A \oplus B) & \oplus_{R_2} \\
\Delta, x : A \vdash Q_1 : (z : C) & \quad \Delta, x : B \vdash Q_2 : (z : C) & \oplus_L \\
\Delta, x : A \oplus B \vdash \text{case } x \text{ of } (Q_1, Q_2) : (z : C) \\
\Delta \vdash P_1 : (x : A) & \quad \Delta \vdash P_2 : (x : B) & \&_R \\
\Delta \vdash \text{case } x \text{ of } (P_1, P_2) : (x : A \& B) \\
\Delta, x : A \& B \vdash x \cdot \text{inl}; Q : (z : C) & \&_{L_1} \\
\Delta, x : A \& B \vdash x \cdot \text{inr}; Q : (z : C) & \&_{L_2}
\end{align*}
\]
Unit for multiplicative conjunction - termination
Unit for multiplicative conjunction - termination

\[ \vdash \text{close } x :: (x : 1)^1_R \]
Unit for multiplicative conjunction - termination

\[
\vdash \text{close } x :: (x : 1)^{1_R}
\]
Unit for multiplicative conjunction - termination

\[ \vdash \text{close } x :: (x : 1) \quad 1^R \]

no orphan providers
Unit for multiplicative conjunction - termination

\[ \vdash \text{close } x :: (x : 1) \quad 1^R \]
Unit for multiplicative conjunction - termination

\[ \cdot \vdash \text{close } x :: (x : 1) \quad 1_R \]

\[ \vdash Q :: (z : C') \]

\[ \Delta, x : 1 \vdash \text{wait } x; Q :: (z : C') \quad 1_L \]
Unit for multiplicative conjunction - termination

\[
\vdash \text{close } x :: (x : 1) \quad 1_R
\]

\[
\Delta \vdash Q :: (z : C') \\
\Delta, x : 1 \vdash \text{wait } x; Q :: (z : C') \quad 1_L
\]
Unit for multiplicative conjunction - termination

\[
\begin{align*}
& \vdash \text{close } x :: (x : 1) \quad 1_R \\
& \Delta \vdash Q :: (z : C') \\
& \Delta, x : 1 \vdash \text{wait } x ; Q :: (z : C') \quad 1_L
\end{align*}
\]

we have lost x!
Unit for multiplicative conjunction - termination

\[ \cdot \vdash \text{close } x :: (x : 1) \quad 1^R \]

we have lost x!

\[ \Delta, x : 1 \vdash \text{wait } x; Q :: (z : C') \quad 1^L \]

no unit for \& and \(\oplus\), since must consist of at least one label
Judgmental rules
Judgmental rules

Cut - spawning new process:
Judgmental rules

Cut - spawning new process:

\[
\Delta_1, \Delta_2 \vdash x \leftarrow P; Q : (z : C) \quad \text{Cut}
\]
Judgmental rules

Cut - spawning new process:

\[
\frac{x : A \vdash Q :: (z : C)}{\Delta_1, \Delta_2 \vdash x \leftarrow P; Q :: (z : C)} \quad \text{Cut}
\]
Judgmental rules

Cut - spawning new process:

\[
\frac{\vdash P :: (x : A) \\ x : A \vdash Q :: (z : C)}{\Delta_1, \Delta_2 \vdash x \leftarrow P; Q :: (z : C)} \text{ Cut}
\]
Judgmental rules

Cut - spawning new process:

\[
\begin{align*}
\Delta_1 \vdash P :: (x : A) & \quad x : A \vdash Q :: (z : C) \\
\Delta_1, \Delta_2 \vdash x \leftarrow P; Q :: (z : C) & \quad \text{Cut}
\end{align*}
\]
Judgmental rules

Cut - spawning new process:

\[
\frac{\Delta_1 \vdash P :: (x : A) \quad \Delta_2, x : A \vdash Q :: (z : C)}{\Delta_1, \Delta_2 \vdash x \leftarrow P; Q :: (z : C)} \quad \text{Cut}
\]
Judgmental rules

Cut - spawning new process:

\[ \Delta_1 \vdash P :: (x : A) \quad \Delta_2, x : A \vdash Q :: (z : C) \]

\[ \Delta_1, \Delta_2 \vdash x \leftarrow P; Q :: (z : C) \]

\( \text{Cut} \)

split context
Judgmental rules

Cut - spawning new process:

\[
\frac{\Delta_1 \vdash P :: (x : A) \quad \Delta_2, x : A \vdash Q :: (z : C)}{\Delta_1, \Delta_2 \vdash x \leftarrow P; Q :: (z : C)} \quad \text{Cut}
\]
Judgmental rules

Cut - spawning new process:

\[
\frac{\Delta_1 \vdash P :: (x : A) \quad \Delta_2, x : A \vdash Q :: (z : C)}{\Delta_1, \Delta_2 \vdash x \leftarrow P; Q :: (z : C)} \quad \text{Cut}
\]

Identity - forwarding:

\[
\text{Identity - forwarding:}
\]
Judgmental rules

Cut - spawning new process:

$$
\Delta_1 \vdash P :: (x : A) \quad \Delta_2, x : A \vdash Q :: (z : C)
$$

$$
\Delta_1, \Delta_2 \vdash x \leftarrow P; Q :: (z : C)
$$

Cut

Identity - forwarding:

$$
\vdash \text{fwd } x y :: (x : A)
$$

Id
Judgmental rules

Cut - spawning new process:

\[
\Delta_1 \vdash P :: (x : A) \quad \Delta_2, x : A \vdash Q :: (z : C) \\
\Delta_1, \Delta_2 \vdash x \leftarrow P; Q :: (z : C)
\]

Identity - forwarding:

\[
\vdash \text{fwd} \ x \ y :: (x : A)
\]

process offering along x terminates, client henceforth interacts with process offering along y
Judgmental rules

Cut - spawning new process:

\[
\frac{\Delta_1 \vdash P :: (x : A) \quad \Delta_2, x : A \vdash Q :: (z : C)}{\Delta_1, \Delta_2 \vdash x \leftarrow P; Q :: (z : C)} \quad \text{Cut}
\]

Identity - forwarding:

\[
\frac{y : A \vdash \text{fwd} \ x \ y :: (x : A)}{\text{Id}}
\]

process offering along x terminates, client henceforth interacts with process offering along y
Judgmental rules

Cut - spawning new process:

\[
\begin{align*}
\Delta_1 &\vdash P :: (x : A) \\
\Delta_2, x : A &\vdash Q :: (z : C) \\
\Delta_1, \Delta_2 &\vdash x \leftarrow P; Q :: (z : C)
\end{align*}
\]

Identity - forwarding:

\[
y : A \vdash \text{fwd } x \ y :: (x : A)
\]

- no orphan providers

- process offering along x terminates, client henceforth interacts with process offering along y
Let’s implement the queue!

We use the formal language SILL used in research papers
The connection to linear logic
The connection to linear logic

if we erase process terms in typing rules, we get left and right rules of intuitionistic linear logic
The connection to linear logic

if we erase process terms in typing rules, we get left and right rules of intuitionistic linear logic

\[
\frac{\Delta \vdash P :: (x : A)}{\Delta \vdash \text{inl}; P :: (x : A \oplus B) \quad \oplus_R_1}
\]

\[
\frac{\Delta \vdash P :: (x : B)}{\Delta \vdash \text{inr}; P :: (x : A \oplus B) \quad \oplus_R_2}
\]

\[
\frac{\Delta, x : A \vdash Q_1 :: (z : C) \quad \Delta, x : B \vdash Q_2 :: (z : C)}{\Delta, x : A \oplus B \vdash \text{case } x \text{ of } (Q_1, Q_2) :: (z : C) \quad \oplus_L}
\]
The connection to linear logic

if we erase process terms in typing rules, we get left and right rules of intuitionistic linear logic

\[
\begin{align*}
\Delta \vdash P :: (x : A) \\
&\quad \Delta \vdash x.\text{inl}; P :: (x : A \oplus B) \quad \oplus_{R_1} \\
\Delta \vdash P :: (x : B) \\
&\quad \Delta \vdash x.\text{inr}; P :: (x : A \oplus B) \quad \oplus_{R_2} \\
\Delta, x : A \vdash Q_1 :: (z : C) &\quad \Delta, x : B \vdash Q_2 :: (z : C) \\
&\quad \Delta, x : A \oplus B \vdash \text{case } x \text{ of } (Q_1, Q_2) :: (z : C) \quad \oplus_L \\
\Delta, A \vdash C &\quad \Delta, B \vdash C \\
&\quad \Delta, A \oplus B \vdash C \quad \oplus_L
\end{align*}
\]
The connection to linear logic

if we erase process terms in typing rules, we get left and right rules of intuitionistic linear logic
The connection to linear logic

- if we erase process terms in typing rules, we get left and right rules of intuitionistic linear logic
- rewrite higher-order channel output with spawn/forward:
The connection to linear logic

if we erase process terms in typing rules, we get left and right rules of intuitionistic linear logic

rewrite higher-order channel output with spawn/forward:

\[
\frac{\Delta \vdash P :: (x : B)}{\Delta, y : A \vdash \text{send } x \ y; P :: (x : A \otimes B)^{\otimes R}}
\]
The connection to linear logic

if we erase process terms in typing rules, we get left and right rules of intuitionistic linear logic

rewrite higher-order channel output with spawn/forward:

\[
\frac{\Delta \vdash P :: (x : B)}{\Delta, y : A \vdash \text{send } x \, y; P :: (x : A \otimes B)} \quad \otimes_R
\]

\[
\frac{y : A \vdash \text{fwd } z \, y :: (z : A) \quad \Delta \vdash P :: (x : B)}{\Delta, y : A \vdash \text{send } x (z \leftarrow \text{fwd } z \, y); P :: (x : A \otimes B)} \quad \otimes_R
\]
The connection to linear logic

if we erase process terms in typing rules, we get left and right rules of intuitionistic linear logic

rewrite higher-order channel output with spawn/forward:

\[
\frac{\Delta \vdash P :: (x : B)}{\Delta, y : A \vdash \text{send } x y; P :: (x : A \otimes B)} \quad \otimes_R
\]

\[
\frac{y : A \vdash \text{fwd } z y :: (z : A) \quad \Delta \vdash P :: (x : B)}{\Delta, y : A \vdash \text{send } x (z \leftarrow \text{fwd } z y); P :: (x : A \otimes B)} \quad \otimes_R
\]

\[
\frac{A \vdash A \quad \Delta \vdash B}{\Delta, A \vdash A \otimes B} \quad \otimes_R
\]
Curry-Howard correspondence
Curry-Howard correspondence

Correspondence between linear logic and session-typed pi-calculus
Curry-Howard correspondence

Correspondence between linear logic and session-typed pi-calculus

<table>
<thead>
<tr>
<th>Logic:</th>
<th>Type theory:</th>
</tr>
</thead>
</table>

Curry-Howard correspondence

Correspondence between linear logic and session-typed pi-calculus

Logic: linear propositions

Type theory:
Curry-Howard correspondence

Correspondence between linear logic and session-typed pi-calculus

<table>
<thead>
<tr>
<th>Logic:</th>
<th>Type theory:</th>
</tr>
</thead>
<tbody>
<tr>
<td>linear propositions</td>
<td>session types</td>
</tr>
</tbody>
</table>
Curry-Howard correspondence

Correspondence between linear logic and session-typed pi-calculus

Logic:
linear propositions
proofs

Type theory:
session types
Curry-Howard correspondence

Correspondence between linear logic and session-typed pi-calculus

<table>
<thead>
<tr>
<th>Logic:</th>
<th>Type theory:</th>
</tr>
</thead>
<tbody>
<tr>
<td>linear propositions</td>
<td>session types</td>
</tr>
<tr>
<td>proofs</td>
<td>programs</td>
</tr>
</tbody>
</table>
Curry-Howard correspondence

Correspondence between linear logic and session-typed pi-calculus

Logic:
- linear propositions
- proofs
- cut reduction

Type theory:
- session types
- programs
Curry-Howard correspondence

Correspondence between linear logic and session-typed pi-calculus

<table>
<thead>
<tr>
<th>Logic:</th>
<th>Type theory:</th>
</tr>
</thead>
<tbody>
<tr>
<td>linear propositions</td>
<td>session types</td>
</tr>
<tr>
<td>proofs</td>
<td>programs</td>
</tr>
<tr>
<td>cut reduction</td>
<td>communication</td>
</tr>
</tbody>
</table>
## Curry-Howard correspondence

Correspondence between linear logic and session-typed pi-calculus

<table>
<thead>
<tr>
<th>Logic:</th>
<th>Type theory:</th>
</tr>
</thead>
<tbody>
<tr>
<td>linear propositions</td>
<td>session types</td>
</tr>
<tr>
<td>proofs</td>
<td>programs</td>
</tr>
<tr>
<td>cut reduction</td>
<td>communication</td>
</tr>
</tbody>
</table>

## Curry-Howard correspondence

Correspondence between linear logic and session-typed pi-calculus

<table>
<thead>
<tr>
<th>Logic:</th>
<th>Type theory:</th>
</tr>
</thead>
<tbody>
<tr>
<td>linear propositions</td>
<td>session types</td>
</tr>
<tr>
<td>proofs</td>
<td>programs</td>
</tr>
<tr>
<td>cut reduction</td>
<td>communication</td>
</tr>
</tbody>
</table>


Philip Wadler. Propositions as sessions. ICFP, 2012.
Benefits of linear logic for programming
Benefits of linear logic for programming

Linear logic is a substructural logic because it rejects the structural rules of weakening and contraction:
Benefits of linear logic for programming

Linear logic is a substructural logic because it rejects the structural rules of weakening and contraction:

\[
\frac{\Gamma \vdash C}{\Gamma, A \vdash C} \quad \text{weaken} \quad \frac{\Gamma, A, A \vdash C}{\Gamma, A \vdash C} \quad \text{contract}
\]
Benefits of linear logic for programming

Linear logic is a substructural logic because it rejects the structural rules of weakening and contraction:

\[
\frac{\Gamma \vdash C}{\Gamma, A \vdash C} \quad weaken \quad \frac{\Gamma, A, A \vdash C}{\Gamma, A \vdash C} \quad contract
\]

“drop resource”
Benefits of linear logic for programming

Linear logic is a substructural logic because it rejects the structural rules of weakening and contraction:

\[
\frac{\Gamma \vdash C}{\Gamma, A \vdash C} \quad \text{weaken} \quad \frac{\Gamma, A, A \vdash C}{\Gamma, A \vdash C} \quad \text{contract}
\]

“drop resource” \quad “duplicate resource”
Benefits of linear logic for programming

Linear logic is a substructural logic because it rejects the structural rules of weakening and contraction:

\[
\frac{\Gamma \vdash C}{\Gamma, A \vdash C} \quad \text{weaken} \quad \frac{\Gamma, A, A \vdash C}{\Gamma, A \vdash C} \quad \text{contract}
\]

“drop resource”  “duplicate resource”

without weakening, every provider has at least one client
Benefits of linear logic for programming

Linear logic is a substructural logic because it rejects the structural rules of weakening and contraction:

\[
\frac{\Gamma \vdash C}{\Gamma, A \vdash C} \quad \text{weaken}\]

\[
\frac{\Gamma, A, A \vdash C}{\Gamma, A \vdash C} \quad \text{contract}
\]

"drop resource"  "duplicate resource"

without weakening, every provider has at least one client

without contraction, every provider has at most one client
Benefits of linear logic for programming

Linear logic is a substructural logic because it rejects the structural rules of weakening and contraction:

\[
\frac{\Gamma \vdash C}{\Gamma, A \vdash C} \quad \text{weaken} \quad \frac{\Gamma, A, A \vdash C}{\Gamma, A \vdash C} \quad \text{contract}
\]

“drop resource” \hspace{1cm} “duplicate resource”

- without weakening, every provider has at least one client
- without contraction, every provider has at most one client
- thus, every provider has exactly one client
Benefits of linear logic for programming

Let’s identify absence of weakening and contraction in our rules:
Benefits of linear logic for programming

Let’s identify absence of weakening and contraction in our rules:

\[ \cdot \vdash \text{close } x :: (x : 1)^{1_R} \]
Benefits of linear logic for programming

Let’s identify absence of weakening and contraction in our rules:

\[ \vdash \text{close} \, x :: (x : 1) \quad 1^R \]
Benefits of linear logic for programming

Let’s identify absence of weakening and contraction in our rules:

\[
\vdash \text{close } x :: (x : 1) \quad 1^R
\]

no resources can be dropped
Benefits of linear logic for programming

Let’s identify absence of weakening and contraction in our rules:

- no resources can be dropped
- every provider has at least one client

\[ \vdash \text{close } x :: (x : 1) \quad 1_R \]
Benefits of linear logic for programming

Let’s identify absence of weakening and contraction in our rules:

\[
\frac{}{\vdash \text{close } x :: (x : 1)}^{1_R}
\]
Benefits of linear logic for programming

Let’s identify absence of weakening and contraction in our rules:

\[ \Delta_1 \vdash P :: (x : A) \quad \Delta_2, x : A \vdash Q :: (z : C) \]
\[ \Delta_1, \Delta_2 \vdash x \leftarrow P; Q :: (z : C) \]

\( \text{Cut} \)
Benefits of linear logic for programming

Let's identify absence of weakening and contraction in our rules:

\[
\vdash \text{close } x :: (x : 1) \quad 1_R
\]

\[
\frac{\Delta_1 \vdash P :: (x : A) \quad \Delta_2, x : A \vdash Q :: (z : C)}{\Delta_1, \Delta_2 \vdash x \leftarrow P; Q :: (z : C)} \quad \text{Cut}
\]
Benefits of linear logic for programming

Let’s identify absence of weakening and contraction in our rules:

\[ \cdot \vdash \text{close } x :: (x : \text{1})^{1_R} \]

\[ \Delta_1 \vdash P :: (x : A) \quad \Delta_2, x : A \vdash Q :: (z : C) \]

\[ \Delta_1, \Delta_2 \vdash x \leftarrow P; Q :: (z : C) \quad \text{Cut} \]

no resources duplicated
Benefits of linear logic for programming

Let’s identify absence of weakening and contraction in our rules:

\[
\cdot \vdash \text{close } x :: (x : 1) \quad 1_R
\]

no aliases created

\[
\Delta_1 \vdash P :: (x : A) \quad \Delta_2, x : A \vdash Q :: (z : C)
\]

\[
\Delta_1, \Delta_2 \vdash x \leftarrow P; Q :: (z : C)
\]

no resources duplicated

Cut
Benefits of linear logic for programming

Let’s identify absence of weakening and contraction in our rules:

\[
\vdash \text{close } x :: (x : 1) \quad 1_R
\]

- no aliases created
- no resources duplicated
- every provider has at most one client
Benefits of linear logic for programming
Benefits of linear logic for programming

linear logic session types turn run-time process graph into a tree
Benefits of linear logic for programming

1. Linear logic session types turn run-time process graph into a tree.
2. For intuitionistic linear logic session types, the tree is directed.
Benefits of linear logic for programming

- Linear logic session types turn run-time process graph into a tree.
- For intuitionistic linear logic session types, the tree is directed.

\[ x_1 : A_1, \ldots, x_n : A_n \vdash P :: (x : A) \]
Benefits of linear logic for programming

- Linear logic session types turn run-time process graph into a tree.
- For intuitionistic linear logic session types, tree is directed.

\[ x_1 : A_1, \ldots, x_n : A_n \vdash P :: (x : A) \]
Benefits of linear logic for programming

- Linear logic session types turn run-time process graph into a tree.
- For intuitionistic linear logic session types, the tree is directed.

\[ x_1 : A_1, \ldots, x_n : A_n \vdash P :: (x : A) \]

Parent: client
Benefits of linear logic for programming

linear logic session types turn run-time process graph into a tree

for intuitionistic linear logic session types, tree is directed

\[ x_1 : A_1, \ldots, x_n : A_n \vdash P :: (x : A) \]

- parent: client
- child: provider
Benefits of linear logic for programming

- Linear logic session types turn run-time process graph into a tree.
- For intuitionistic linear logic session types, tree is directed.

\[ x_1 : A_1, \ldots, x_n : A_n \vdash P :: (x : A) \]

Parent: client

Child: provider

We will use directedness for deadlock-freedom.
Benefits of linear logic for programming
Benefits of linear logic for programming

- type safety holds easily:
Benefits of linear logic for programming

type safety holds easily:
Benefits of linear logic for programming

type safety holds easily:

Preservation (a.k.a., session fidelity)
Benefits of linear logic for programming

Type safety holds easily:

Preservation (a.k.a., session fidelity)
- every provider has a unique client
Benefits of linear logic for programming

Type safety holds easily:

Preservation (a.k.a., session fidelity) ✓
- every provider has a unique client
Benefits of linear logic for programming

type safety holds easily:

Preservation (a.k.a., session fidelity) ✓
- every provider has a unique client

Progress (a.k.a., deadlock-freedom)
Benefits of linear logic for programming

- **Preservation (a.k.a., session fidelity)** ✓
  - every provider has a unique client

- **Progress (a.k.a., deadlock-freedom)**
  - 2 possible threats to progress:
    - provider ready to synchronize, client not
    - client ready to synchronize, provider not

type safety holds easily:
Benefits of linear logic for programming

Type safety holds easily:

Preservation (a.k.a., session fidelity)
- every provider has a unique client

Progress (a.k.a., deadlock-freedom)
- 2 possible threats to progress:
  - provider ready to synchronize, client not
  - client ready to synchronize, provider not

\[ \text{“a waits for b”} \]
Benefits of linear logic for programming

Type safety holds easily:

Preservation (a.k.a., session fidelity)
- every provider has a unique client

Progress (a.k.a., deadlock-freedom)
- 2 possible threats to progress:
  - provider ready to synchronize, client not
  - client ready to synchronize, provider not

"a waits for b"
Benefits of linear logic for programming

Type safety holds easily:

Preservation (a.k.a., session fidelity)
- every provider has a unique client

Progress (a.k.a., deadlock-freedom)
- 2 possible threats to progress:
  - provider ready to synchronize, client not
  - client ready to synchronize, provider not

Green arrows can only go along edges, thus cannot form a cycle
Benefits of linear logic for programming

- **Type safety holds easily:**
  - a waits for b

- **Preservation (a.k.a., session fidelity)**
  - every provider has a unique client

- **Progress (a.k.a., deadlock-freedom)**
  - 2 possible threats to progress:
    - provider ready to synchronize, client not
    - client ready to synchronize, provider not

- Green arrows can only go along edges, thus cannot form a cycle
Type safety formalized
Type safety formalized

Type safety expresses coherence between statics (type system) and dynamics of a language
Type safety formalized

Type safety expresses coherence between statics (type system) and dynamics of a language

let’s define the dynamics
Type safety formalized

Type safety expresses coherence between statics (type system) and dynamics of a language

Let’s define the dynamics

Multiset rewriting rules:
Type safety formalized

Type safety expresses coherence between statics (type system) and dynamics of a language

let's define the dynamics

Multiset rewriting rules:

\[ \text{proc}(a, P^a; P'), \text{proc}(c, Q^a; Q') \rightarrow \text{proc}(a, P'), \text{proc}(c, Q') \]
Type safety formalized

Type safety expresses coherence between statics (type system) and dynamics of a language

let’s define the dynamics

Multiset rewriting rules:

\[
\text{proc}(a, P\langle a \rangle; P'),\ \text{proc}(c, Q\langle a \rangle; Q') \\
\rightarrow \quad \text{proc}(a, P'),\ \text{proc}(c, Q')
\]

rewrite process tree only describing what changes
Type safety formalized

Type safety expresses coherence between statics (type system) and dynamics of a language

let’s define the dynamics

Multiset rewriting rules:

\[
\text{proc}(a, P\langle a \rangle; P'), \text{proc}(c, Q\langle a \rangle; Q') \\
\rightarrow \text{proc}(a, P'), \text{proc}(c, Q')
\]

rewrite process tree only describing what changes
Type safety formalized

Type safety expresses coherence between statics (type system) and dynamics of a language

let’s define the dynamics

Multiset rewriting rules:

\[
\text{proc}(a, P\langle a \rangle; P'), \text{proc}(c, Q\langle a \rangle; Q') \\
\text{\underrightarrow{}} \quad \text{proc}(a, P'), \text{proc}(c, Q')
\]

rewrite process tree only describing what changes
Type safety formalized

Type safety expresses coherence between statics (type system) and dynamics of a language

Let’s define the dynamics

Multiset rewriting rules:

\[ \text{proc}(a, P(a); P'), \text{proc}(c, Q(a); Q') \]

\[ \rightarrow \text{proc}(a, P'), \text{proc}(c, Q') \]

Rewrite process tree only describing what changes

After rewrite
Type safety formalized

Type safety expresses coherence between statics (type system) and dynamics of a language

let’s define the dynamics

Multiset rewriting rules:

\[
\text{proc}(a, P\langle a \rangle; P'), \text{proc}(c, Q\langle a \rangle; Q') \\
\quad \rightarrow \quad \text{proc}(a, P'), \text{proc}(c, Q')
\]

rewrite process tree only describing what changes
Type safety formalized

Type safety expresses coherence between statics (type system) and dynamics of a language

let’s define the dynamics

Multiset rewriting rules:

\[
\text{proc}(a, P\langle a \rangle; P'), \quad \text{proc}(c, Q\langle a \rangle; Q') \\
\rightarrow \quad \text{proc}(a, P'), \quad \text{proc}(c, Q')
\]

rewrite process tree only describing what changes
Type safety formalized

Type safety expresses coherence between statics (type system) and dynamics of a language

→ let’s define the dynamics

Multiset rewriting rules:

\[
\text{proc}(a, P<a>; P'), \text{proc}(c, Q<a>; Q') \\
\rightarrow \text{proc}(a, P'), \text{proc}(c, Q')
\]

→ rewrite process tree only describing what changes
Type safety formalized

Type safety expresses coherence between statics (type system) and dynamics of a language

Let’s define the dynamics

Multiset rewriting rules:

\[
\text{proc}(a, P\langle a \rangle; P'), \text{proc}(c, Q\langle a \rangle; Q') \quad \rightarrow \quad \text{proc}(a, P'), \text{proc}(c, Q')
\]

Rewrite process tree only describing what changes
Type safety formalized

Type safety expresses coherence between statics (type system) and dynamics of a language

Let's define the dynamics

Multiset rewriting rules:

\[
\text{proc}(a, P\langle a \rangle; P'), \text{proc}(c, Q\langle a \rangle; Q') \\
\rightarrow \quad \text{proc}(a, P'), \text{proc}(c, Q')
\]

Rewrite process tree only describing what changes
Type safety formalized

Type safety expresses coherence between statics (type system) and dynamics of a language

let’s define the dynamics

Multiset rewriting rules:

\[ \text{proc}(a, P(a); P'), \text{proc}(c, Q(a); Q') \rightarrow \text{proc}(a, P'), \text{proc}(c, Q') \]

offering channel

rewrite process tree only describing what changes
Type safety formalized

Type safety expresses coherence between statics (type system) and dynamics of a language

→ let’s define the dynamics

Multiset rewriting rules:

offering channel

\[ \text{proc}(a, P(a); P'), \text{proc}(c, Q(a); Q') \rightarrow \text{proc}(a, P'), \text{proc}(c, Q') \]

→ rewrite process tree only describing what changes
Dynamics

Selected rules:
Dynamics

Selected rules:

\[(D-\otimes) \quad \text{proc}(a, \text{send } a \ b; P), \ \text{proc}(c, y \leftarrow \text{recv } a; Q_y) \]
\[\rightarrow \quad \text{proc}(a, P), \ \text{proc}(c, [b/y] Q_y)\]
Selected rules:

\[(D-\otimes)\] \hspace{1cm} \text{proc}(a, \text{send } a \ b; P), \text{proc}(c, y \leftarrow \text{recv } a; Q_y) \\
\quad \rightarrow \text{proc}(a, P), \text{proc}(c, [b/y] Q_y) \\
\]

\[(D-\&)\] \hspace{1cm} \text{proc}(a, \text{case } a \text{ of } l \Rightarrow P), \text{proc}(c, a.l_k; Q) \\
\quad \rightarrow \text{proc}(a, P_k), \text{proc}(c, Q) \]
Dynamics

Selected rules:

(D-⊗) \quad \text{proc}(a, \text{send } a \ b; P), \text{proc}(c, y \leftarrow \text{recv } a; Q_y) \\
\quad \quad \rightarrow \text{proc}(a, P), \text{proc}(c, [b/y] Q_y)

(D-&) \quad \text{proc}(a, \text{case } a \text{ of } l \Rightarrow P), \text{proc}(c, a.l_k; Q) \\
\quad \quad \rightarrow \text{proc}(a, P_k), \text{proc}(c, Q)

(D-1) \quad \text{proc}(a, \text{close } a), \text{proc}(c, \text{wait } a; Q) \\
\quad \quad \rightarrow \text{proc}(c, Q)
Dynamics

Selected rules:

(D-⊗) \( \text{proc}(a, \text{send } a \ b; P), \text{proc}(c, y \leftarrow \text{recv } a; Q_y) \rightarrow \text{proc}(a, P), \text{proc}(c, [b/y] Q_y) \)

(D-&) \( \text{proc}(a, \text{case } a \text{ of } l \Rightarrow P), \text{proc}(c, a.l_k; Q) \rightarrow \text{proc}(a, P_k), \text{proc}(c, Q) \)

(D-1) \( \text{proc}(a, \text{close } a), \text{proc}(c, \text{wait } a; Q) \rightarrow \text{proc}(c, Q) \)

(D-Cut) \( \text{proc}(c, x \leftarrow P_x; Q_x) \rightarrow \text{proc}(a, [a/x] P_x), \text{proc}(c, [a/x] Q_x) \) (a fresh)
Dynamics

Selected rules:

\((D-\otimes)\) \quad \text{proc}(a, \text{send} \ a \ b; P), \ \text{proc}(c, y \leftarrow \text{recv} \ a; Q_y) \\
\rightarrow \ \text{proc}(a, P), \ \text{proc}(c, [b/y] Q_y) \\

\((D-\&\) \) \quad \text{proc}(a, \text{case} \ a \ \text{of} \ l \ \Rightarrow \ P), \ \text{proc}(c, a.l_k; Q) \\
\rightarrow \ \text{proc}(a, P_k), \ \text{proc}(c, Q) \\

\((D-1)\) \quad \text{proc}(a, \text{close} \ a), \ \text{proc}(c, \text{wait} \ a; Q) \\
\rightarrow \ \text{proc}(c, Q) \\

\((D-\text{Cut})\) \quad \text{proc}(c, x \leftarrow P_x; Q_x) \\
\rightarrow \ \text{proc}(a, [a/x] P_x), \ \text{proc}(c, [a/x] Q_x) \quad \text{(a fresh)} \\

\((D-\text{Id})\) \quad \text{proc}(a, \text{fwd} \ a \ b) \\
\rightarrow \ (a = b)
Dynamics

Selected rules:

\[(D-\otimes)\quad \text{proc}(a, \text{send } a \ b; P), \text{proc}(c, y \leftarrow \text{recv } a; Q_y) \rightarrow \text{proc}(a, P), \text{proc}(c, [b/y] Q_y)\]

\[(D-\&)\quad \text{proc}(a, \text{case } a \text{ of } \overline{l \Rightarrow P}), \text{proc}(c, a.l_k; Q) \rightarrow \text{proc}(a, P_k), \text{proc}(c, Q)\]

\[(D-1)\quad \text{proc}(a, \text{close } a), \text{proc}(c, \text{wait } a; Q) \rightarrow \text{proc}(c, Q)\]

\[(D-\text{Cut})\quad \text{proc}(c, x \leftarrow P_x; Q_x) \rightarrow \text{proc}(a, [a/x] P_x), \text{proc}(c, [a/x] Q_x) \quad \text{(a fresh)}\]

\[(D-\text{Id})\quad \text{proc}(a, \text{fwd } a \ b) \rightarrow (a = b)\]
Dynamics

Selected rules:

(D-⊗) \[\text{proc}(a, \text{send } a \ b; P), \text{proc}(c, y \leftarrow \text{recv } a; Q_y)\]
\[\rightarrow \text{proc}(a, P), \text{proc}(c, [b/y] Q_y)\]

(D-&) \[\text{proc}(a, \text{case } a \text{ of } l \Rightarrow P), \text{proc}(c, a.l_k; Q)\]
\[\rightarrow \text{proc}(a, P_k), \text{proc}(c, Q)\]

(D-1) \[\text{proc}(a, \text{close } a), \text{proc}(c, \text{wait } a; Q)\]
\[\rightarrow \text{proc}(c, Q)\]

(D-Cut) \[\text{proc}(c, x \leftarrow P_x; Q_x)\]
\[\rightarrow \text{proc}(a, [a/x] P_x), \text{proc}(c, [a/x] Q_x)\] (a fresh)

(D-Id) \[\text{proc}(a, \text{fwd } a \ b)\]
\[\rightarrow (a = b)\]
Dynamics

Selected rules:

(D-⊗) \quad \text{proc}(a, \text{send } a \ b; P), \text{proc}(c, y \leftarrow \text{recv } a; Q_y) \\
\quad \rightarrow \text{proc}(a, P), \text{proc}(c, [b/y] Q_y)

(D-&) \quad \text{proc}(a, \text{case } a \text{ of } l \Rightarrow P), \text{proc}(c, a.l_k; Q) \\
\quad \rightarrow \text{proc}(a, P_k), \text{proc}(c, Q)

(D-1) \quad \text{proc}(a, \text{close } a), \text{proc}(c, \text{wait } a; Q) \\
\quad \rightarrow \text{proc}(c, Q)

(D-Cut) \quad \text{proc}(c, x \leftarrow P_x; Q_x) \\
\quad \rightarrow \text{proc}(a, [a/x] P_x), \text{proc}(c, [a/x] Q_x) \quad \text{(a fresh)}

(D-Id) \quad \text{proc}(a, \text{fwd } a \ b) \\
\quad \rightarrow (a = b)

- synchronous dynamic
- both send and receive are blocking
- asynchronous semantics: spawns off messages and links them with forward
Configuration typing
Configuration typing

In addition to typing process terms, we must type the run-time configuration of processes (a.k.a. heap typing)
Configuration typing

In addition to typing process terms, we must type the run-time configuration of processes (a.k.a. heap typing)

\[ \vdash (\cdot) :: (\cdot) \]

\[ \Gamma \vdash \Omega :: \Delta_1, \Delta_2 \quad \Delta_1 \vdash P_a :: (a : A) \]
\[ \vdash \Omega, \text{proc}(a, P_a) :: (\Delta_2, a : A) \]
Configuration typing

In addition to typing process terms, we must type the run-time configuration of processes (a.k.a. heap typing)

\[
\Gamma \vdash \Omega :: \Delta_1, \Delta_2 \quad \Delta_1 \vdash P_a :: (a : A)
\]

\[
\vdash \Omega, \text{proc}(a, P_a) :: (\Delta_2, a : A)
\]
Configuration typing

In addition to typing process terms, we must type the run-time configuration of processes (a.k.a. heap typing)

\[
\Gamma \vdash \Omega :: \Delta_1, \Delta_2 \quad \Delta_1 \vdash P_a :: (a : A)
\]

\[
\vdash \Omega, \text{proc}(a, P_a) :: (\Delta_2, a : A)
\]
Configuration typing

In addition to typing process terms, we must type the run-time configuration of processes (a.k.a. heap typing)

\[
\Gamma \vdash \Omega :: \Delta_1, \Delta_2 \\
\Delta_1 \vdash P_a :: (a : A) \\
\Gamma \vdash \Omega, \text{proc}(a, P_a) :: (\Delta_2, a : A)
\]

\[\Omega\]

\[\Delta_1 = b, c, d\]
Configuration typing

In addition to typing process terms, we must type the run-time configuration of processes (a.k.a. heap typing)

\[
\Gamma \vdash \Omega :: \Delta_1, \Delta_2 \quad \Delta_1 \vdash P_a :: (a : A) \\
\Gamma \vdash \Omega, \mathit{proc}(a, P_a) :: (\Delta_2, a : A)
\]

\[\begin{align*}
\Omega \\
\Delta_1 &= b, c, d \\
\Delta_2 &= e
\end{align*}\]
Configuration typing

In addition to typing process terms, we must type the run-time configuration of processes (a.k.a. heap typing)

\[
\Gamma \vdash \Omega :: \Delta_1, \Delta_2 \\
\Delta_1 \vdash P_a :: (a : A) \\
\Gamma \vdash \Omega, \text{proc}(a, P_a) :: (\Delta_2, a : A)
\]

typing imposes forest structure and tree structure at top level
Configuration typing

In addition to typing process terms, we must type the run-time configuration of processes (a.k.a. heap typing)

\[
\Gamma \vdash \Omega :: \Delta_1, \Delta_2 \quad \Delta_1 \vdash P_a :: (a : A)
\]

\[
\vdash \Omega, \text{proc}(a, P_a) :: (\Delta_2, a : A)
\]

typing imposes forest structure and tree structure at top level

a closed program offers a session of type \(1\), the top-level “main” process
Preservation and progress
Preservation and progress

**Theorem (Preservation).** If $\models \Omega :: \Delta$ and $\Omega \rightarrow \Omega'$, then $\models \Omega' :: \Delta$. 
Preservation and progress

**Theorem** (Preservation). If $\models \Omega :: \Delta$ and $\Omega \rightarrow \Omega'$, then $\models \Omega' :: \Delta$. 
Preservation and progress

**Theorem (Preservation).** If $\models \Omega :: \Delta$ and $\Omega \rightarrow \Omega'$, then $\models \Omega' :: \Delta$. 
Preservation and progress

**Theorem** (Preservation). *If $\models \Omega :: \Delta$ and $\Omega \rightarrow \Omega'$, then $\models \Omega' :: \Delta$.*
Preservation and progress

**Theorem (Preservation).** If $\vdash \Omega :: \Delta$ and $\Omega \rightarrow \Omega'$, then $\vdash \Omega' :: \Delta$. 

The figure shows a tree-like structure with nodes labeled as follows: 

- **P** at the root
- **a**, **b**, **c**, **d**, **e** as leaves

The text indicates no communication along **a** and **e**, b/c no clients.

Diagram:

- Node **P** is at the top with **a** and **e** as leaf nodes.
- **b**, **c**, **d** are internal nodes leading to the leaves.
Preservation and progress

**Theorem (Preservation).** If $\vdash \Omega :: \Delta$ and $\Omega \rightarrow \Omega'$, then $\vdash \Omega' :: \Delta$.

**Theorem (Progress).** If $\vdash \Omega :: \Delta$, then either
1. $\Omega \rightarrow \Omega'$, for some $\Omega'$, or
2. $\Omega$ is poised.
Preservation and progress

**Theorem (Preservation).** If $\models \Omega :: \Delta$ and $\Omega \rightarrow \Omega'$, then $\models \Omega' :: \Delta$.

**Theorem (Progress).** If $\models \Omega :: \Delta$, then either

1. $\Omega \rightarrow \Omega'$, for some $\Omega'$, or
2. $\Omega$ is poised.
Preservation and progress

**Theorem (Preservation).** If $\models \Omega :: \Delta$ and $\Omega \rightarrow \Omega'$, then $\models \Omega' :: \Delta$.

**Theorem (Progress).** If $\models \Omega :: \Delta$, then either
1. $\Omega \rightarrow \Omega'$, for some $\Omega'$, or
2. $\Omega$ is poised.

![Diagram showing the process poised property](image)
Preservation and progress

**Theorem (Preservation).** If $\models \Omega :: \Delta$ and $\Omega \rightarrow \Omega'$, then $\models \Omega' :: \Delta$.

**Theorem (Progress).** If $\models \Omega :: \Delta$, then either

1. $\Omega \rightarrow \Omega'$, for some $\Omega'$, or
2. $\Omega$ is poised.

*every process poised*

* a poised process is ready to sync along offering channel*
Preservation and progress

**Theorem (Preservation).** If $\vdash \Omega :: \Delta$ and $\Omega \rightarrow \Omega'$, then $\vdash \Omega' :: \Delta$.

**Theorem (Progress).** If $\vdash \Omega :: \Delta$, then either

1. $\Omega \rightarrow \Omega'$, for some $\Omega'$, or
2. $\Omega$ is poised.

every process poised

a poised process is ready to sync along offering channel
Taking stock
Taking stock

linear session type language
Taking stock

linear session type language

guarantees session fidelity and deadlock-freedom
Taking stock

- linear session type language
- guarantees session fidelity and deadlock-freedom
- corresponds to intuitionistic linear logic
Taking stock

- linear session type language
- guarantees session fidelity and deadlock-freedom
- corresponds to intuitionistic linear logic
- one connective from linear logic still missing: persistent truth
Taking stock

linear session type language

guarantees session fidelity and deadlock-freedom

corresponds to intuitionistic linear logic

one connective from linear logic still missing: persistent truth

\[
A, B \triangleq A \otimes B \quad \text{multiplicative conjunction} \quad \text{“channel output”}
\]
\[
A \multimap B \quad \text{multiplicative implication} \quad \text{“channel input”}
\]
\[
A \& B \quad \text{additive conjunction} \quad \text{“external choice”}
\]
\[
A \oplus B \quad \text{additive disjunction} \quad \text{“internal choice”}
\]
\[
1 \quad \text{unit for } \otimes \quad \text{“termination”}
\]
Taking stock

- **linear session type language**
- guarantees session fidelity and deadlock-freedom
- corresponds to intuitionistic linear logic
- one connective from linear logic still missing: persistent truth

\[
A, B \triangleq A \otimes B \quad \text{multiplicative conjunction} \quad \text{“channel output”}
\]
\[
A \multimap B \quad \text{multiplicative implication} \quad \text{“channel input”}
\]
\[
A \& B \quad \text{additive conjunction} \quad \text{“external choice”}
\]
\[
A \oplus B \quad \text{additive disjunction} \quad \text{“internal choice”}
\]
\[
1 \quad \text{unit for } \otimes \quad \text{“termination”}
\]
\[
!A \quad \text{“of course”, persistent truth} \quad \text{“replication”}
\]