Session-Typed Concurrent Programming

Stephanie Balzer
Carnegie Mellon University

POPL 2019 TutorialFest

1 Supported by a Mozilla Research Grant and an NSF grant (No. CCF-1718267)
Content of tutorial

Message-passing programming and session types
• types for typing such programs

Linear logic and session types
• benefits and limitations

Manifest sharing
• controlled sharing for concurrency (non-determinism)

Manifest sharing and the pi-calculus
• expressiveness of untyped asynchronous pi-calculus recovered

Current & future research
Learning objectives of tutorial

• How to program with session types
• What linearity is good for in programming
• How logic can guide programming language design
• Expressiveness of session-typed programming
• Hands-on experience with Concurrent C0
Session-typed message-passing programming
Message-passing concurrent programming

Program: network of processes connected by channels
Computation: by message exchange along channels
N-ary channels: e.g., a connects P₁, P₂, and P₃

Legend: process ─ channel ─ message
Message-passing concurrent programming

Program: network of processes connected by channels

Computation: by message exchange along channels

N-ary channels: e.g., a connects P₁, P₂, and P₃

Legend:  process  channel  message
Message-passing concurrent programming

Program: network of processes connected by channels

Computation: by message exchange along channels

N-ary channels: e.g., a connects P_1, P_2, and P_3

Legend:  
- process 
- channel 
- message
Message-passing concurrent programming

Program: network of processes connected by channels

Computation: by message exchange along channels

N-ary channels: e.g., a connects P₁, P₂, and P₃

Legend:  process  channel  message
Message-passing concurrent programming

Program: network of processes connected by channels

Computation: by message exchange along channels

N-ary channels: e.g., a connects $P_1$, $P_2$, and $P_3$

Legend: process channel message
Message-passing concurrent programming

Program: network of processes connected by channels

Computation: by message exchange along channels

N-ary channels: e.g., a connects P₁, P₂, and P₃

Legend: process channel message
Message-passing concurrent programming

Program: network of processes connected by channels
Computation: by message exchange along channels
N-ary channels: e.g., channel a connects P₁, P₂, and P₃

Legend:
- process
- channel
- message

non-determinism
Message-passing concurrent programming

Program: network of processes connected by channels

Computation: by message exchange along channels

N-ary channels: e.g., a connects P₁, P₂, and P₃

→ non-determinism

→ formal model: pi-calculus

[Milner 1992]
Example: queue
Example: queue
Example: queue
Example: queue

enqueue: client sends “enq” followed by “L” along q
Example: queue

enqueue: client sends “enq” followed by “L” along q
Example: queue

enqueue: client sends “enq” followed by “L” along q

client queue

enqueue: client sends “enq” followed by “L” along q
Example: queue

enqueue: client sends “enq” followed by “L” along q

depqueue: client sends “deq”, then receives “P”
Example: queue

enqueue: client sends “enq” followed by “L” along q

decqueue: client sends “deq”, then receives “P”
Example: queue

- In our example, we’ve stored characters in the queue
- However, we can also store channel references:
Example: queue

- In our example, we’ve stored characters in the queue
- However, we can also store channel references:
Example: queue

- In our example, we’ve stored characters in the queue.
- However, we can also store channel references:

  “higher-order” channels (session types)
Example: queue

- In our example, we’ve stored characters in the queue.
- However, we can also store channel references:

  ![Diagram](image)

  - “higher-order” channels (session types)
  - “mobility” (pi-calculus)
Types for protocols of message exchange
Types for protocols of message exchange

Session types [Honda 1993]

\[ A \triangleq \text{?}[T].A' \mid ![T].A' \mid \&\{l_1 : A_1, \ldots, l_n : A_n\} \mid \oplus\{l_1 : A_1, \ldots, l_n : A_n\} \mid \text{end} \mid X \mid \mu X.A' \]

\[ T \triangleq A \mid \text{int} \mid \ldots \]
Types for protocols of message exchange

Session types [Honda 1993]

\[ A \triangleq \ ?[T].A' \mid ![T].A' \mid \&\{l_1 : A_1, \ldots, l_n : A_n\} \mid \oplus\{l_1 : A_1, \ldots, l_n : A_n\} \mid \text{end} \mid X \mid \mu X.A' \]

\[ T \triangleq A \mid \text{int} \mid \ldots \]

input: receive message of type T, continue as type A'
Types for protocols of message exchange

Session types [Honda 1993]

\[ A \triangleq ?[T].A' \mid ![T].A' \mid \]
\[ \&\{l_1 : A_1, \ldots, l_n : A_n\} \mid \oplus\{l_1 : A_1, \ldots, l_n : A_n\} \mid \]
\[ \text{end} \mid X \mid \mu X.A' \]

\[ T \triangleq A \mid \text{int} \mid \ldots \]
Types for protocols of message exchange

Session types [Honda 1993]

\[ A \triangleq ?[T].A' \mid ![T].A' \mid \&\{l_1 : A_1, \ldots, l_n : A_n\} \mid \oplus\{l_1 : A_1, \ldots, l_n : A_n\} \mid \text{end} \mid X \mid \mu X.A' \]

\[ T \triangleq A \mid \text{int} \mid \ldots \]

output: send message of type T, continue as type A'
Types for protocols of message exchange

Session types [Honda 1993]

\[ A \triangleq [T].A' \mid ![T].A' \mid \&\{l_1 : A_1, \ldots, l_n : A_n\} \mid \oplus\{l_1 : A_1, \ldots, l_n : A_n\} \mid \text{end} \mid X \mid \mu X.A' \]

\[ T \triangleq A \mid \text{int} \mid \ldots \]
Types for protocols of message exchange

Session types [Honda 1993]

\[ A \triangleq ?[T].A' \mid ![T].A' \mid \&\{l_1 : A_1, \ldots, l_n : A_n\} \mid \oplus\{l_1 : A_1, \ldots, l_n : A_n\} \mid \text{end} \mid X \mid \mu X.A' \]

\[ T \triangleq A \mid \text{int} \mid \ldots \]

external choice: receive label \( l_i \), continue as type \( A_i \)
Types for protocols of message exchange

Session types [Honda 1993]

\[ A \triangleq ?[T].A' \mid ![T].A' \mid \&\{l_1 : A_1, \ldots, l_n : A_n\} \mid \oplus\{l_1 : A_1, \ldots, l_n : A_n\} \mid \text{end} \mid X \mid \mu X.A' \]

\[ T \triangleq A \mid \text{int} \mid \ldots \]
Types for protocols of message exchange

Session types [Honda 1993]

\[ A \triangleq \ [ ?[T].A' | ![T].A' | \&\{l_1 : A_1, \ldots, l_n : A_n\} | \oplus\{l_1 : A_1, \ldots, l_n : A_n\} | \text{end} | X | \mu X.A' \] 

\[ T \triangleq A | \text{int} | \ldots \]

internal choice: send label l_i, continue as type A_i
Types for protocols of message exchange

Session types [Honda 1993]

\[ A \triangleq \ ?[T].A' \mid ![T].A' \mid \&\{l_1 : A_1, \ldots, l_n : A_n\} \mid \oplus\{l_1 : A_1, \ldots, l_n : A_n\} \mid \text{end} \mid X \mid \mu X.A' \]

\[ T \triangleq A \mid \text{int} \mid \ldots \]
Types for protocols of message exchange

Session types [Honda 1993]

$$A \triangleq \ ?[T].A' \mid ![T].A' \mid$$
$$\&\{l_1 : A_1, \ldots, l_n : A_n\} \mid \oplus\{l_1 : A_1, \ldots, l_n : A_n\} \mid$$
$$\text{end} \mid X \mid \mu X.A'$$

$$T \triangleq A \mid \text{int} \mid \ldots$$

termination: close session and terminate
Types for protocols of message exchange

Session types [Honda 1993]

\[ A \triangleq ?[T].A' \mid ![T].A' \mid \&\{l_1 : A_1, \ldots, l_n : A_n\} \mid \oplus\{l_1 : A_1, \ldots, l_n : A_n\} \mid \text{end} \mid X \mid \mu X.A' \]

\[ T \triangleq A \mid \text{int} \mid \ldots \]
Types for protocols of message exchange

Session types [Honda 1993]

\[ A \triangleq \ ?[T].A' \mid ![T].A' \mid \\
& \{l_1 : A_1, \ldots, l_n : A_n\} \mid \oplus\{l_1 : A_1, \ldots, l_n : A_n\} \mid \\
end \mid X \mid \mu X.A' \]

\[ T \triangleq A \mid \text{int} \mid \ldots \]

recursive session types
Types for protocols of message exchange

Session types [Honda 1993]

\[ A \triangleq ?[T].A' \mid ![T].A' \mid \&\{l_1 : A_1, \ldots, l_n : A_n\} \mid \oplus\{l_1 : A_1, \ldots, l_n : A_n\} \mid \text{end} \mid X \mid \mu X.A' \]

\[ T \triangleq A \mid \text{int} \mid \ldots \]
Types for protocols of message exchange

Session types [Honda 1993]

\[ A \triangleq ?[T].A' \mid ![T].A' \mid \&\{l_1 : A_1, \ldots, l_n : A_n\} \mid \oplus\{l_1 : A_1, \ldots, l_n : A_n\} \mid \text{end} \mid X \mid \mu X.A' \]

\[ T \triangleq A \mid \text{int} \mid \ldots \]

Example:

\[ \text{queue} = \&\{\text{enq} : ?[\text{char}].\text{queue}, \quad \text{deq} : \oplus\{\text{none} : \text{end}, \text{some} : ![\text{char}].\text{queue}\}\} \]
Session types change with protocol
Session types change with protocol

![Diagram showing a client connected to a queue through different session types labeled as "P", "O", and "P".]

client → q → "P" → "O" → "P" → queue
Session types change with protocol

What is the type of channel q?
Session types change with protocol

What is the type of channel q?

```
queue = \{enq : ?[char].queue,
    deq : \{none : end, some : ![char].queue\}\}
```
Session types change with protocol

queue = &{enq : ?[char].queue,
    deq : ⊕{none : end, some : ![char].queue}}
Session types change with protocol

queue = \{enq : ?[char].queue, 
       deq : \oplus\{none : end, some : ![char].queue\}\}

Action: 
Type:

q: queue
Session types change with protocol

\[ \text{queue} = \& \{ \text{enq : ?[char].queue,} \\
\quad \text{deq : } \oplus \{ \text{none : end, some : ![char].queue} \} \} \]

Action: send "enq" along q
Session types change with protocol

queue = &{enq : ?[char].queue,
    deq : ⊕{none : end, some : ![char].queue}}

Action:  
Type:
q: queue
send “enq” along q  
q: ?[char].queue
Session types change with protocol

\[
\text{queue} = \&\{\text{enq : } ?[\text{char}].\text{queue}, \\
\quad \text{deq : } \oplus\{\text{none : end}, \text{some : } ![\text{char}].\text{queue}\}\}\}
\]

**Action:**
- send “enq” along \(q\)
- send “L” along \(q\)

**Type:**
- \(q: \text{queue}\)
- \(q: ?[\text{char}].\text{queue}\)
- \(q:\)
Session types change with protocol

queue = &{enq : ?[char].queue,  
        deq : {none : end, some : ![char].queue}}

Action:  

send “enq” along q  
send “L” along q

Type:

q: queue  
q: ?[char].queue  
q: queue
Session types change with protocol

```plaintext
queue = &{enq : ?[char].queue,
          deq : ⊕{none : end, some : ![char].queue}}
```

**Action:**
- `send "enq" along q`
  - Type: `q: queue`
- `send "L" along q`
  - Type: `q: [char].queue`
- `send "deq" along q`
  - Type: `q: queue`

**Type:**
- `q: queue`
Session types change with protocol

\[
\text{queue} = \&\{ \text{enq} : ?[\text{char}].\text{queue}, \\
\quad \text{deq} : \oplus\{ \text{none} : \text{end}, \text{some} : ![\text{char}].\text{queue} \} \}
\]

<table>
<thead>
<tr>
<th>Action</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>send “enq” along q</td>
<td>q: ![char].queue</td>
</tr>
<tr>
<td>send “L” along q</td>
<td>q: ![char].queue</td>
</tr>
<tr>
<td>send “deq” along q</td>
<td>q: \oplus{\text{none} : \text{end}, \text{some} : ![\text{char}].\text{queue} }</td>
</tr>
</tbody>
</table>
Session types change with protocol

\[
\text{queue} = \&\{\text{enq} : ?[\text{char}].\text{queue}, \\
\quad \text{deq} : \oplus\{\text{none} : \text{end}, \text{some} : ![\text{char}].\text{queue}\}\}\}
\]

<table>
<thead>
<tr>
<th>Action:</th>
<th>Type:</th>
</tr>
</thead>
<tbody>
<tr>
<td>send “enq” along q</td>
<td>(q: \text{queue})</td>
</tr>
<tr>
<td>send “L” along q</td>
<td>(q: ?[\text{char}].\text{queue})</td>
</tr>
<tr>
<td>send “deq” along q</td>
<td>(q: \oplus{\text{none} : \text{end}, \text{some} : ![\text{char}].\text{queue}})</td>
</tr>
<tr>
<td>receive “some” along q</td>
<td>(q:)</td>
</tr>
</tbody>
</table>
Session types change with protocol

\[\text{queue} = \&\{ \text{enq} : ?[\text{char}].\text{queue}, \]
\[\quad \text{deq} : \oplus\{ \text{none} : \text{end}, \text{some} : ![\text{char}].\text{queue} \} \}\]

**Action:**

<table>
<thead>
<tr>
<th>Action</th>
<th>Type:</th>
</tr>
</thead>
<tbody>
<tr>
<td>send “enq” along q</td>
<td>q: queue</td>
</tr>
<tr>
<td>send “L” along q</td>
<td>q: [\text{char}.\text{queue} ]</td>
</tr>
<tr>
<td>send “deq” along q</td>
<td>q: \oplus{\text{none} : \text{end}, \text{some} : !\text{char}.\text{queue} }</td>
</tr>
<tr>
<td>receive “some” along q</td>
<td>q: !\text{char}.\text{queue}</td>
</tr>
</tbody>
</table>
Session types change with protocol
Session types change with protocol

What if?

client 2

q

“P” “O” “P”

client 1

queue
Session types change with protocol

What if?

client 2

q

client 1

queue

Action: q

Type: q: queue
Session types change with protocol

What if?

client 2

client 1

queue

Action: client 1 sends “enq” along q

Type: q: queue
Session types change with protocol

What if?

client 1

client 2

q

“P” “O” “P”

queue

Action:

client 1 sends “enq” along q

Type:

q: queue

q: ?[char].queue
Session types change with protocol

What if?

Action:
client 1 sends “enq” along q
client 2 sends “enq” along q

Type:
q: queue
q: ?[char].queue
Session types change with protocol

What if?

client 2

q

client 1

queue

Action:

client 1 sends “enq” along q

Type:

q: queue

client 2 sends “enq” along q

q: ?[char].queue

q is not at expected type!
Preservation (session fidelity)

- Expectations of client and provider match, if they do initially.
- How can we recover preservation?
Preservation (session fidelity)

- Expectations of client and provider match, if they do initially.
- How can we recover preservation?

Use linear logic as a foundation for session types
Preservation (session fidelity)

• Expectations of client and provider match, if they do initially.

• How can we recover preservation?

Use linear logic as a foundation for session types

Let’s view session types as linear propositions
Preservation (session fidelity)

• Expectations of client and provider match, if they do initially.

• How can we recover preservation?

Use linear logic as a foundation for session types

Let’s view session types as linear propositions

Linear logic allows us to treat channels as “resources”
Preservation (session fidelity)

• Expectations of client and provider match, if they do initially.

• How can we recover preservation?

Use linear logic as a foundation for session types

Let’s view session types as linear propositions

Linear logic allows us to treat channels as “resources”

What does that mean?
Linear session types
Linear logic

Rejects the following two structural rules:

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

“drop resource”

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

“duplicate resource”
Linear logic

Rejects the following two structural rules:

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

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

“drop resource”

“duplicate resource”

Presented work based on intuitionistic linear logic
Linear logic

Rejects the following two structural rules:

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

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

“drop resource”

“duplicate resource”

Presented work based on intuitionistic linear logic

Distinction of provider (right) from clients (left) of turnstile
Weakening

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

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

client → queue

q → “P” → “O” → “P”
Weakening

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

client \quad \text{queue}
Weakening

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

Terminating client without terminating or passing on queue
Weakening

\[ \Gamma \vdash C \]

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

- Client
- Queue

Terminating client without terminating or passing on queue
Weakening

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

Terminating client without terminating or passing on queue

Providing process without a client
Contraction

\[ \Gamma, A, A \vdash C \]

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

\[ \Gamma, A, A \vdash C \]

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

client → queue

q → “P” → “O” → “P”
Contraction

\[ \Gamma, A, A \vdash C \]
\[ \Gamma, A \vdash C \]

contract

Passing on channel reference and keeping it
Contraction

\[ \Gamma, A, A \vdash C \]

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

Passing on channel reference and keeping it

client

queue
Contraction

\[\Gamma, A, A \vdash C\]

contract

\[\Gamma, A \vdash C\]

Passing on channel reference and keeping it

Providing process has multiple clients
Tree structure

Without weakening and contraction, process graph forms a tree.
Tree structure

Without weakening and contraction, process graph forms a tree.

parent: client
Tree structure

Without weakening and contraction, process graph forms a tree.

parent: client

child: provider
Tree structure

Without weakening and contraction, process graph forms a tree.

Every providing process has exactly one client.
Tree structure

Without weakening and contraction, process graph forms a tree.

Every providing process has exactly one client.

Session fidelity restored.
Reconstructing session types from linear propositions [Caires & Pfenning 2010, Wadler 2012]

Curry-Howard correspondence: intuitionistic linear logic - session-typed pi-calculus [Caires & Pfenning 2010]

<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>
Reconstructing session types from linear propositions [Caires & Pfenning 2010, Wadler 2012]

Curry-Howard correspondence: intuitionistic linear logic - session-typed pi-calculus [Caires & Pfenning 2010]

Logic:
- Linear propositions
- Proofs
- Cut reduction

Type theory:
- Session types
- Programs
- Communication

Let’s discover this correspondence together!
Intuitionistic linear logic

Connectives:

\[ A, B \cong A \otimes B \quad \text{multiplicative conjunction} \]
\[ A \multimap B \quad \text{multiplicative implication} \]
\[ A \& B \quad \text{additive conjunction} \]
\[ A \oplus B \quad \text{additive disjunction} \]
\[ !A \quad \text{"of course", persistent truth} \]
Intuitionistic linear logic

Connectives:

\[ A, B \triangleq A \otimes B \quad \text{multiplicative conjunction} \]
\[ A \multimap B \quad \text{multiplicative implication} \]
\[ A \& B \quad \text{additive conjunction} \]
\[ A \oplus B \quad \text{additive disjunction} \]
\[ !A \quad \text{”of course”, persistent truth} \]

Judgment: \[ A_1, \ldots, A_n \vdash A \]
Intuitionistic linear logic

Connectives:

\[ A, B \triangleq A \otimes B \]  multiplicative conjunction
\[ A \multimap B \]  multiplicative implication
\[ A \& B \]  additive conjunction
\[ A \oplus B \]  additive disjunction
\[ !A \]  ”of course”, persistent truth

Judgment: \[ A_1, \ldots, A_n \vdash P :: A \]
Intuitionistic linear logic

Connectives:

\[ A, B \triangleq A \otimes B \] multiplicative conjunction
\[ A \rightarrow B \] multiplicative implication
\[ A \& B \] additive conjunction
\[ A \oplus B \] additive disjunction
\[ !A \] "of course", persistent truth

Judgment: \[ A_1, \ldots, A_n \vdash P :: (x : A) \]
Intuitionistic linear logic

Connectives:

\[ A, B \trianglelefteq A \otimes B \quad \text{multiplicative conjunction} \]
\[ A \multimap B \quad \text{multiplicative implication} \]
\[ A \& B \quad \text{additive conjunction} \]
\[ A \oplus B \quad \text{additive disjunction} \]
\[ !A \quad \text{”of course”, persistent truth} \]

Judgment: \( x_1 : A_1, \ldots, x_n : A_n \vdash P :: (x : A) \)
Intuitionistic linear logic

Connectives:

\[ A, B \triangleq A \otimes B \] multiplicative conjunction
\[ A \multimap B \] multiplicative implication
\[ A \& B \] additive conjunction
\[ A \oplus B \] additive disjunction
\[ !A \] ”of course”, persistent truth

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

“Process P offers a session of type A along channel x using the sessions \[ A_1, \ldots, A_n \] provided along channels \[ x_1, \ldots, x_n \].”
Intuitionistic linear logic

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

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

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

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

\[ \Delta \vdash :: ( : A) \quad \Delta \vdash :: ( : B) \]

\[ \frac{\Delta \vdash :: ( : A) \quad \Delta \vdash :: ( : B)}{\Delta \vdash :: ( : A \& B)} \text{ (T-\&R)} \]
Additive conjunction

\[
\begin{align*}
\Delta \vdash & \quad :: (x : A) \quad \Delta \vdash & \quad :: (x : B) \\
\hline
\Delta \vdash & \quad :: (x : A \& B)
\end{align*}
\] (T-\&R)
Additive conjunction

\[ \Delta \vdash :: (x : A) \quad \Delta \vdash :: (x : B) \]

\[ \Delta \vdash :: (x : A \& B) \quad (T-\&R) \]
Additive conjunction

\[ \Delta \vdash P_1 :: (x : A) \quad \Delta \vdash P_2 :: (x : B) \]

\[ \Delta \vdash :: (x : A \& B) \quad \text{(T-\&R)} \]
Additive conjunction

\[ \Delta \vdash P_1 :: (x : A) \quad \Delta \vdash P_2 :: (x : B) \]

\[ \frac{}{\Delta \vdash \text{case } x \text{ of } (P_1, P_2) :: (x : A \& B)} \quad (\text{T-}\&\text{R}) \]
Additive conjunction

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

\[ \Delta, A \vdash :: (\_ : C) \]
\[ \Delta, A\&B \vdash :: (\_ : C) \]
Additive conjunction

\[
\Delta \vdash P_1 :: (x : A) \quad \Delta \vdash P_2 :: (x : B) \\
\hline
\Delta \vdash \text{case } x \text{ of } (P_1, P_2) :: (x : A \& B) \\
\hline
\Delta, \quad A \vdash :: (z : C) \\
\hline
\Delta, \quad A \& B \vdash :: (z : C)
\]

(T-\&_R)

(T-\&_{L1})
Additive conjunction

\[
\Delta \vdash P_1 :: (x : A) \quad \Delta \vdash P_2 :: (x : B)
\]

\[
\frac{}{\Delta \vdash \text{case } x \text{ of } (P_1, P_2) :: (x : A\&B)} \quad (T-&R)
\]

\[
\Delta, x : A \vdash \quad :: (z : C)
\]

\[
\frac{}{\Delta, x : A\&B \vdash \quad :: (z : C)} \quad (T-&L_1)
\]
Additive conjunction

\[ \Delta \vdash P_1 \:: \: (x : A) \quad \Delta \vdash P_2 \:: \: (x : B) \]

\[ \Delta \vdash \text{case } x \text{ of } (P_1, P_2) \:: \: (x : A \& B) \]

\[ \Delta, x : A \vdash \:: \: (z : C) \]

\[ \Delta, x : A \& B \vdash \: x.\text{inl}; Q \:: \: (z : C) \]

(T-\&R)

(T-\&L1)
Additive conjunction

\[
\Delta \vdash P_1 :: (x : A) \quad \Delta \vdash P_2 :: (x : B)
\]

\[
\Delta \vdash \text{case } x \text{ of } (P_1, P_2) :: (x : A \& B)
\]

\[
\Delta, x : A \vdash Q :: (z : C)
\]

\[
\Delta, x : A \& B \vdash x.\text{inl}; Q :: (z : C)
\]
Additive conjunction

\[
\begin{align*}
\Delta & \vdash P_1 :: (x : A) \quad \Delta & \vdash P_2 :: (x : B) \\
\Delta & \vdash \text{case } x \text{ of } (P_1, P_2) :: (x : A \& B) \\
\Delta, x : A & \vdash Q :: (z : C) \\
\Delta, x : A \& B & \vdash x.\text{inl}; Q :: (z : C) \\
\Delta, x : B & \vdash Q :: (z : C) \\
\Delta, x : A \& B & \vdash x.\text{inr}; Q :: (z : C) \\
\end{align*}
\]
Additive conjunction

\[ \Delta \vdash P_1 :: (x : A) \quad \Delta \vdash P_2 :: (x : B) \]

\[ \Delta \vdash \text{case } x \text{ of } (P_1, P_2) :: (x : A \& B) \]

\[ \Delta, x : A \vdash Q :: (z : C) \]

\[ \Delta, x : A \& B \vdash x.\text{inl}; Q :: (z : C) \]  \quad (T-\&L_1)

\[ \Delta, x : B \vdash Q :: (z : C) \]

\[ \Delta, x : A \& B \vdash x.\text{inr}; Q :: (z : C) \]  \quad (T-\&L_2)

External choice: client chooses. Generalizes to \( \& \{ \overline{l} : A \} \).
Additive disjunction

\[ \Delta \vdash P :: (x : A) \]

\[ \Delta \vdash x.\text{inl}; P :: (x : A \oplus B) \quad (T-\oplus_{R1}) \]

\[ \Delta \vdash P :: (x : B) \]

\[ \Delta \vdash x.\text{inr}; P :: (x : A \oplus B) \quad (T-\oplus_{R2}) \]

\[ \Delta, x : A \vdash Q_1 :: (z : C) \quad \Delta, x : B \vdash Q_2 :: (z : C) \]

\[ \Delta, x : A \oplus B \vdash \text{case } x \text{ of } (Q_1, Q_2) :: (z : C) \quad (T-\oplus_L) \]

Internal choice: provider chooses. Generalizes to \( \oplus \{ \text{\overline{l}} : A \} \).
Multiplicative implication

\[
\frac{\Delta, \quad A \vdash :: (\quad : B)}{\Delta \vdash :: (\quad : A \rightarrow B)} \quad (T\rightarrow\circ_\mathbb{R})
\]
Multiplicative implication

\[ \frac{\Delta, A \vdash :: (\_ : B)}{\Delta \vdash :: (x : A \rightarrow B)} \quad (T_{\rightarrow R}) \]
Multiplicative implication

\[
\Delta, \quad A \vdash :: (x : B) \\
\hline
\Delta \vdash :: (x : A \rightarrow B) \\
\quad (T-\rightarrow_R)
\]
Multiplicative implication

\[ \Delta, y : A \vdash :: (x : B) \]

\[ \Delta \vdash :: (x : A \rightarrow B) \]

(T→⊙R)
Multiplicative implication

\[ \Delta, y : A \vdash :: (x : B) \]
\[ \Delta \vdash y \leftarrow \text{recv } x; P :: (x : A \rightarrow B) \]

(T→R)
Multiplicative implication

\[
\Delta, y : A \vdash P :: (x : B) \\
\hline
\Delta \vdash y \leftarrow \text{recv } x; P :: (x : A \rightarrow B)
\] 

(T→R)
Multiplicative implication

\[ \Delta, y : A \vdash P :: (x : B) \]
\[ \Delta \vdash y \leftarrow \text{recv } x; P :: (x : A \multimap B) \]

\[ \Delta \vdash :: ( : A) \quad \Delta', \quad B \vdash :: ( : C) \]
\[ \Delta, \Delta', \quad A \multimap B \vdash :: ( : C) \]

(T→R)

(T→L)

Channel input: provider receives a channel.
Multiplicative implication

\[ \Delta, y : A \vdash P :: (x : B) \]
\[ \frac{\Delta \vdash y \leftarrow \text{recv } x; P :: (x : A \rightarrow B)}{(T \rightarrow \circ_R)} \]

\[ \Delta \vdash :: (\_ : A) \quad \Delta', B \vdash :: (\_ : C) \]
\[ \frac{\Delta, \Delta', A \rightarrow B \vdash :: (\_ : C)}{(T \rightarrow \circ_L)} \]

Channel input: provider receives a channel.
Multiplicative implication

\[
\frac{\Delta, y : A \vdash P :: (x : B) \quad (T \rightarrow_R)}{\Delta \vdash y \leftarrow \text{recv } x; P :: (x : A \rightarrow B)}
\]

\[
\frac{\Delta \vdash \cdot :: (\cdot : A) \quad \Delta', x : B \vdash \cdot :: (z : C) \quad (T \rightarrow_L)}{\Delta, \Delta', x : A \rightarrow B \vdash \cdot :: (z : C)}
\]

Channel input: provider receives a channel.
Multiplicative implication

\[
\begin{align*}
\Delta, y : A &\vdash P :: (x : B) \\
\Delta &\vdash y \leftarrow \text{recv } x; P :: (x : A \rightarrow B)
\end{align*}
\]  
\[
(T-\rightarrow_R)
\]

\[
\begin{align*}
\Delta &\vdash :: (\_ : A) \\
\Delta', x : B &\vdash :: (z : C)
\end{align*}
\]  
\[
\Delta, \Delta', x : A \rightarrow B &\vdash \text{send } x (y \leftarrow Q); Q' :: (z : C)
\]  
\[
(T-\rightarrow_L)
\]

Channel input: provider receives a channel.
Multiplicative implication

\[
\begin{align*}
\Delta, y : A &\vdash P :: (x : B) \\
\hline
\Delta &\vdash y \leftarrow \text{recv } x; P :: (x : A \to B) & (T-\to^\circ_R)
\end{align*}
\]

\[
\begin{align*}
\Delta &\vdash :: (\cdot : A) \\
\hline
\Delta', x : B &\vdash Q' :: (z : C) & (T-\to^\circ_L)
\end{align*}
\]

\[
\begin{align*}
\Delta, \Delta', x : A \to B &\vdash \text{send } x (y \leftarrow Q); Q' :: (z : C)
\end{align*}
\]

Channel input: provider receives a channel.
Multiplicative implication

\[ \Delta, y : A \vdash P :: (x : B) \]
\[ \Delta \vdash y \leftarrow \text{recv } x; P :: (x : A \multimap B) \]

\[ \Delta \vdash :: (y : A) \quad \Delta', x : B \vdash Q' :: (z : C) \]
\[ \Delta, \Delta', x : A \multimap B \vdash \text{send } x (y \leftarrow Q); Q' :: (z : C') \]

Channel input: provider receives a channel.
Multiplicative implication

\[
\begin{align*}
\Delta, y : A &\vdash P :: (x : B) \\
\Delta &\vdash y \leftarrow \text{recv } x; P :: (x : A \rightarrow B) \quad (T-\rightarrow_R)
\end{align*}
\]

\[
\begin{align*}
\Delta &\vdash Q :: (y : A) \\
\Delta', x : B &\vdash Q' :: (z : C) \\
\Delta, \Delta', x : A \rightarrow B &\vdash \text{send } x (y \leftarrow Q); Q' :: (z : C) \quad (T-\rightarrow_L)
\end{align*}
\]

Channel input: provider receives a channel.
Multiplicative conjunction

\[
\Delta \vdash Q :: (y : A) \quad \Delta' \vdash P :: (x : B) \\
\Delta, \Delta' \vdash \text{send } x (y \leftarrow Q); P :: (x : A \otimes B)
\]  
\[(T-\otimes_R)\]

\[
\Delta, x : B, y : A \vdash Q' :: (z : C) \\
\Delta, x : A \otimes B \vdash y \leftarrow \text{recv } x; Q' :: (z : C)
\]  
\[(T-\otimes_L)\]

Channel output: provider sends a channel.
Unit of multiplicative conjunction

\[
\vdash \cdot :: (T - 1_R)
\]
Unit of multiplicative conjunction

\[
\vdash \quad :: (x : 1) \\
\]

\[
\text{(T-1}_R) 
\]
Unit of multiplicative conjunction

\[
\vdash \text{close } x :: (x : 1) \quad (T - 1_R)
\]
Unit of multiplicative conjunction

\[ \Gamma, x : 1 \vdash \text{close } x :: (x : 1) \quad (\text{T-1}_R) \]

\[ \Delta, : 1 \vdash :: (\_ : C) \quad (\text{T-1}_L) \]
Unit of multiplicative conjunction

\[
\begin{align*}
\Delta \vdash & \quad :: (z : C') \\
\Delta, \quad : 1 \vdash & \quad :: (z : C')
\end{align*}
\]

\[\text{(T-1}_R)\]

\[
\begin{align*}
\cdot \vdash & \quad \text{close } x :: (x : 1)
\end{align*}
\]

\[\text{(T-1}_L)\]
Unit of multiplicative conjunction

\[
\begin{align*}
\Delta & \vdash :: (z : C) \\
\Delta, x : 1 & \vdash :: (z : C) \\
\end{align*}
\]
\[(T-1_R)\]
\[
\cdot \vdash \text{close } x :: (x : 1)
\]
Unit of multiplicative conjunction

\[ \Delta \vdash \text{close } x :: (x : 1) \text{ (T-1}^R \text{)} \]

\[ \Delta \vdash :: (z : C) \]

\[ \Delta, x : 1 \vdash \text{wait } x; Q :: (z : C) \text{ (T-1}^L \text{)} \]
Unit of multiplicative conjunction

\[
\begin{align*}
\Delta &\vdash Q :: (z : C) \\
\hline
\Delta, x : 1 &\vdash \text{wait } x; Q :: (z : C)
\end{align*}
\]
Unit of multiplicative conjunction

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

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

Termination
Unit of multiplicative conjunction

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

\[(T-1_R)\]

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

\[(T-1_L)\]

Termination and ⊕ must consist of at least one label.
Recap
Recap

Connectives:

\[ A, B \triangleq A \otimes B \] channel output
\[ A \rightarrow B \] channel input
\[ \oplus \{ l : A \} \] internal choice
\[ \& \{ \overline{l} : A \} \] external choice
\[ 1 \] termination
Recap

Connectives:

\[ A, B \overset{\Delta}{=} A \otimes B \quad \text{channel output} \]
\[ A \to B \quad \text{channel input} \]
\[ \oplus \{l : A\} \quad \text{internal choice} \]
\[ \& \{ \overline{l} : A \} \quad \text{external choice} \]
\[ 1 \quad \text{termination} \]

Example:

\[
\text{queue } A = \& \{ \text{enq} : A \to \text{queue } A, \text{deq} : \oplus \{ \text{none} : 1, \text{some} : A \otimes \text{queue } A \} \}
\]
Judgmental rules
Judgmental rules

\[ \Delta \vdash P(x : A) \quad \Delta', x : A \vdash Q :: (z : C) \]
\[ \frac{}{\Delta, \Delta' \vdash x \leftarrow P; Q :: (z : C)} \quad \text{(T-cut)} \]
Judgmental rules

\[
\frac{\Delta \vdash P(x : A) \quad \Delta', x : A \vdash Q :: (z : C)}{
\Delta, \Delta' \vdash x \leftarrow P; Q :: (z : C) \quad \text{(T-CUT)}}
\]

Parallel composition (spawning new process)
Judgmental rules

\[ \Delta \vdash P(x : A) \quad \Delta', x : A \vdash Q ::= (z : C) \]

\[ \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quan
Judgmental rules

\[
\Delta \vdash P(x : A) \quad \Delta', x : A \vdash Q :: (z : C) \\
\frac{}{\Delta, \Delta' \vdash \text{fwd } x \leftarrow P; Q :: (z : C')} \quad \text{(T-CUT)}
\]

Parallel composition (spawning new process)

\[
\frac{}{y : A \vdash \text{fwd } x y :: (x : A)} \quad \text{(T-ID)}
\]

Forward: process offering along x terminates, client henceforth interacts with process offering along y.
Let’s program in Concurrent C0!

C0 [Pfenning 2010, Arnold 2010]
• safe subset of C supporting contracts
• teaching language developed at CMU
• http://c0.typesafety.net

Concurrent C0 [Willsey & Prabhu & Pfenning 2016]
• extends C0 with session types

Installing Concurrent C0
• see http://www.cs.cmu.edu/~balzers/popl_tutorial_2019
What about type safety?
What about type safety?

Preservation (aka session fidelity)
What about type safety?

Preservation (aka session fidelity)

- no weakening and contraction: process graph forms a tree
What about type safety?

Preservation (aka session fidelity)

- no weakening and contraction: process graph forms a tree
- every provider has a unique client
What about type safety?

Preservation (aka session fidelity) ✓

- no weakening and contraction: process graph forms a tree
- every provider has a unique client
What about type safety?

Preservation (aka session fidelity) ✓

• no weakening and contraction: process graph forms a tree
• every provider has a unique client

Progress
What about type safety?

Preservation (aka session fidelity)

- no weakening and contraction: process graph forms a tree
- every provider has a unique client

Progress

- What are the dangers to progress?
What about type safety?

Preservation (aka session fidelity) ✔

• no weakening and contraction: process graph forms a tree
• every provider has a unique client

Progress

• What are the dangers to progress?
• 2 scenarios:
What about type safety?

Preservation (aka session fidelity) ✓
- no weakening and contraction: process graph forms a tree
- every provider has a unique client

Progress
- What are the dangers to progress?
- 2 scenarios:
  - provider ready to synchronize, client not
What about type safety?

Preservation (aka session fidelity) ✔
- no weakening and contraction: process graph forms a tree
- every provider has a unique client

Progress
- What are the dangers to progress?
- 2 scenarios:
  - provider ready to synchronize, client not
  - client ready to synchronize, provider not
What about type safety?

Preservation (aka session fidelity) ✓
- no weakening and contraction: process graph forms a tree
- every provider has a unique client

Progress
- What are the dangers to progress?
- 2 scenarios:
  - provider ready to synchronize, client not
  - client ready to synchronize, provider not
What about type safety?

Preservation (aka session fidelity) ✓

- no weakening and contraction: process graph forms a tree
- every provider has a unique client

Progress

- What are the dangers to progress?
- 2 scenarios:
  - provider ready to synchronize, client not
  - client ready to synchronize, provider not

“a waits for b”
What about type safety?

Preservation (aka session fidelity) ✔

• no weakening and contraction: process graph forms a tree
• every provider has a unique client

Progress

• What are the dangers to progress?
• 2 scenarios:
  • provider ready to synchronize, client not
  • client ready to synchronize, provider not

“a waits for b”
What about type safety?

Preservation (aka session fidelity) ✓
- no weakening and contraction: process graph forms a tree
- every provider has a unique client

Progress
- What are the dangers to progress?
- 2 scenarios:
  - provider ready to synchronize, client not
  - client ready to synchronize, provider not
- green arrows can only go along edges, thus there are no cycles

“a waits for b”
What about type safety?

Preservation (aka session fidelity) ✓
- no weakening and contraction: process graph forms a tree
- every provider has a unique client

Progress ✓
- What are the dangers to progress?
- 2 scenarios:
  - provider ready to synchronize, client not
  - client ready to synchronize, provider not
- green arrows can only go along edges, thus there are no cycles
Of course!
Of course!

Connectives

\[ A, B \triangleq A \otimes B \quad \text{channel output} \]
\[ A \rightarrow B \quad \text{channel input} \]
\[ \oplus \{ l : A \} \quad \text{internal choice} \]
\[ \& \{ l : A \} \quad \text{external choice} \]
\[ 1 \quad \text{termination} \]
\[ !A \quad \text{persistence truth} \]
Of course!

Connectives

\( A, B \quad \triangleq \quad A \otimes B \quad \text{channel output} \)
\( A \rightarrow B \quad \text{channel input} \)
\( \oplus \{ l : A \} \quad \text{internal choice} \)
\( \& \{ l : A \} \quad \text{external choice} \)
\( 1 \quad \text{termination} \)
\( !A \quad \text{persistence truth} \)
Of course!

Connectives

\[ A, B \triangleq A \otimes B \]
\[ A \rightarrow B \quad \text{channel input} \]
\[ \oplus \{l : A\} \quad \text{internal choice} \]
\[ \& \{l : A\} \quad \text{external choice} \]
\[ 1 \quad \text{termination} \]
\[ !A \quad \text{persisten truth} \]

Unrestricted proposition, can be used arbitrarily often
Of course!

Connectives

\[ A, B \triangleq A \otimes B \quad \text{channel output} \]
\[ A \rightarrow B \quad \text{channel input} \]
\[ \oplus \{ l : A \} \quad \text{internal choice} \]
\[ \& \{ l : A \} \quad \text{external choice} \]
\[ 1 \quad \text{termination} \]
\[ !A \quad \text{persistent truth} \]

Unrestricted proposition, can be used arbitrarily often

\[ \Psi; \Delta \vdash P :: (x : A) \]
Of course!

Connectives

\[ A, B \triangleq A \otimes B \quad \text{channel output} \]
\[ A \overrightarrow{\implies} B \quad \text{channel input} \]
\[ \oplus\{l : A\} \quad \text{internal choice} \]
\[ \&\{l : A\} \quad \text{external choice} \]
\[ 1 \quad \text{termination} \]
\[ !A \quad \text{persistent truth} \]

Unrestricted proposition, can be used arbitrarily often

\[
\Psi; \Delta \vdash P :: (x : A)
\]

\(\Psi\) structural context (weakening and contraction)
Of course!

What is the computational meaning of “of course”?
What is the computational meaning of “of course”? 

Q → P
Of course!

What is the computational meaning of “of course”?

Q → P

copy
Of course!

What is the computational meaning of “of course”? 

![Diagram showing the relationship between Q and P, with an arrow indicating a copy process from Q to P']
Of course!

What is the computational meaning of “of course”? 

Copy: obtain linear copy from linear server process
Of course!

What is the computational meaning of “of course”? 

Copy: obtain linear copy from linear server process

Corresponds to replication in the pi-calculus
Of course!

What is the computational meaning of “of course”?

Copy: obtain linear copy from linear server process

Corresponds to replication in the pi-calculus

For some applications, a copying semantics is sufficient. For other applications, sharing is necessary.
Manifest sharing
Manifest sharing - key ideas

[Balzer & Pfenning 2017]
Manifest sharing - key ideas [Balzer & Pfenning 2017]

Acquire-release

- Multiple aliases (shared channels) to a process permitted, but communication requires exclusive access.
Manifest sharing - key ideas [Balzer & Pfenning 2017]

Acquire-release

• Multiple aliases (shared channels) to a process permitted, but communication requires exclusive access.

Manifestation in type structure

• Enrich type structure with adjoint modalities to prescribe acquire and release points.
Manifest sharing - key ideas [Balzer & Pfenning 2017]

Acquire-release

- Multiple aliases (shared channels) to a process permitted, but communication requires exclusive access.

Manifestation in type structure

- Enrich type structure with adjoint modalities to prescribe acquire and release points.

Equi-synchronizing session type

- Invariant guaranteeing that a process is released to the same type at which it was previously acquired, should it be released at all.
- Avoids run-time type checking upon acquire.
Manifest sharing - key ideas [Balzer & Pfenning 2017]

Acquire-release

- Multiple aliases *(shared channels)* to a process permitted, but communication requires exclusive access.

Manifestation in type structure

- Enrich type structure with *adjoint modalities* to prescribe acquire and release points.

Equi-synchronizing session type

- Invariant guaranteeing that a process is released to the same type at which it was previously acquired, should it be released at all.
- Avoids run-time type checking upon acquire.

let’s look at each in turn
Programmatic working of acquire-release
Programmatic working of acquire-release

Legend:

- **linear channel x**
- **shared channel y**
Programmatic working of acquire-release

producer and consumer contend for access to queue

Legend:  
- linear channel x  
- shared channel y
Programmatic working of acquire-release

Legend:  
- linear channel $x$  
- shared channel $y$
Programmatic working of acquire-release

consumer: c

producer: q2

Legend: x linear channel x y shared channel y
Programmatic working of acquire-release
Programmatic working of acquire-release

consumer: $q_1'$

consumer has exclusive access to queue

Legend:
- **linear channel x**
- **shared channel y**
Programmatic working of acquire-release

Legend: $x$ linear channel, $y$ shared channel

consumer: $c$ to $q_1'$

producer: $p$ to $q_2$

$rel$ to $elem$, $q$ to $elem$

$rel$ to $empty$, $a_2$ to $t_2$

consumer has exclusive access to queue
Programmatic working of acquire-release

Legend:  
- **x**: linear channel  
- **y**: shared channel
Programmatic working of acquire-release
Programmatic working of acquire-release

Note: acquire and release messages by client have matching actions by provider

Legend: linear channel x, shared channel y
Programmatic working of acquire-release

Note: acquire and release messages by client have matching actions by provider

What should be the type of q?

Legend:
- **linear channel x**
- **shared channel y**
Manifestation of acquire-release in typing
Manifestation of acquire-release in typing

\[
\text{queue } A = \&\{\text{enq} : A \rightarrow \text{queue } A, \\
\text{deq} : \oplus\{\text{none} : 1, \text{some} : A \otimes \text{queue } A\}\}
\]
Manifestation of acquire-release in typing

\[
\text{queue } A = \&\{\text{enq} : A \rightarrow \text{queue } A, \\
\text{deq} : \oplus\{\text{none} : \text{queue } A, \text{some} : A \otimes \text{queue } A\}\}
\]
Manifestation of acquire-release in typing

\[
\text{queue } A = \&\{\text{enq}: A \rightarrow \text{queue } A, \\
\quad \text{deq}: \oplus\{\text{none}: \text{queue } A, \text{some}: A \otimes \text{queue } A\}\}
\]
Manifestation of acquire-release in typing

\[
\text{acquire}
\]

\[
\text{queue } A = \&\{\text{enq} : A \rightarrow \text{queue } A, \\
\text{deq} : \oplus\{\text{none} : \text{queue } A, \text{some} : A \otimes \text{queue } A\}\}
\]
Manifestation of acquire-release in typing

\[
\text{queue } A = \&\{ \text{enq} : A \rightarrow \text{queue } A, \\
\text{deq} : \oplus\{\text{none} : \text{queue } A, \text{some} : A \otimes \text{queue } A\}\}
\]
Manifestation of acquire-release in typing

\[ \text{queue } A = \{ \text{enq} : A \rightarrow \text{queue } A, \text{deq} : \oplus \{ \text{none} : \text{queue } A, \text{some} : A \otimes \text{queue } A \} \} \]

Legend: 
- \text{process in linear phase} 
- \text{process in shared phase}
Adjoint stratification of session types\(^1\)

1 Generalization of [Benton 1994; Reed 2009; Pfenning and Griffith 2015]
Adjoint stratification of session types¹

stratify session types into a linear and shared layer, s.t. $S > L$

¹ Generalization of [Benton 1994; Reed 2009; Pfenning and Griffith 2015]
Adjoint stratification of session types\(^1\)

stratify session types into a linear and shared layer, s.t. \(S > L\)

\[
\begin{align*}
A_S & \triangleq \\
A_L, B_L & \triangleq \oplus \{ \overline{l : A_L} \} \mid A_L \otimes B_L \mid 1 \mid \\
& \quad \& \{ \overline{l : A_L} \} \mid A_L \to B_L
\end{align*}
\]

\(^1\) Generalization of [Benton 1994; Reed 2009; Pfenning and Griffith 2015]
Adjoint stratification of session types\textsuperscript{1}

stratify session types into a linear and shared layer, s.t. $S > L$

\[
\begin{align*}
A_S & \doteq \\
A_L, B_L & \doteq \oplus \{ \overline{l : A_L} \} \mid A_L \otimes B_L \mid 1 \mid \\
& \& \{ \overline{l : A_L} \} \mid A_L \rightarrow B_L
\end{align*}
\]

\textsuperscript{1} Generalization of [Benton 1994; Reed 2009; Pfenning and Griffith 2015]
Adjoint stratification of session types\(^1\)

Stratify session types into a linear and shared layer, s.t. \(S > L\)

Connect layers with modalities going back and forth

\[
\begin{align*}
A_S & \overset{\triangle}{=} \\
A_L, B_L & \overset{\triangle}{=} \oplus \{ \overline{l : A_L} \} \mid A_L \otimes B_L \mid \mathbf{1} \mid \\
& \quad \& \{ \overline{l : A_L} \} \mid A_L \rightsquigarrow B_L
\end{align*}
\]

\(^1\) Generalization of [Benton 1994; Reed 2009; Pfenning and Griffith 2015]
Adjoint stratification of session types

- stratify session types into a linear and shared layer, s.t. $S > L$
- connect layers with modalities going back and forth

$$A_S \triangleq \uparrow_L^S A_L$$
$$A_L, B_L \triangleq \oplus \{ \overline{l} : A_L \} \mid A_L \otimes B_L \mid 1 \mid \& \{ \overline{l} : A_L \} \mid A_L \rightarrow B_L \mid \downarrow_L^S A_S$$

1 Generalization of [Benton 1994; Reed 2009; Pfenning and Griffith 2015]
Adjoint stratification of session types\(^1\)

- stratify session types into a linear and shared layer, s.t. \(S > L\)
- connect layers with modalities going back and forth

\[
\begin{align*}
A_S & \triangleq \uparrow^S_L A_L \\
A_L, B_L & \triangleq \{l : A_L\} \mid A_L \otimes B_L \mid 1 \mid \\
& \quad \& \{l : A_L\} \mid A_L \rightarrow B_L \mid \downarrow^S_L A_S
\end{align*}
\]

- support of communication of shared channels, using \(\Pi\) for receive and \(\exists\) for send\(^2\)

---

1 Generalization of [Benton 1994; Reed 2009; Pfenning and Griffith 2015]
2 Based on [Cervesato et al. 2002; Watkins et al. 2001]
Adjoint stratification of session types

1 stratify session types into a linear and shared layer, s.t. $S > L$

2 connect layers with modalities going back and forth

$A_S \triangleq \uparrow^S_L A_L$

$A_L, B_L \triangleq \bigoplus \{ l : A_L \} \mid A_L \otimes B_L \mid \mathbf{1} \mid \exists x : A_S. B_L$

& $\bigwedge \{ l : A_L \} \mid A_L \rightarrow B_L \mid \downarrow^S_L A_S \mid \Pi x : A_S. B_L$

support of communication of shared channels, using $\Pi$ for receive and $\exists$ for send

1 Generalization of [Benton 1994; Reed 2009; Pfenning and Griffith 2015]

2 Based on [Cervesato et al. 2002; Watkins et al. 2001]
Adjoint stratification of session types

\[
A_S \quad \overset{\triangleq}{=} \quad \uparrow^S_L A_L
\]

\[
A_L, B_L \quad \overset{\triangleq}{=} \quad \oplus \{l : A_L\} \mid A_L \otimes B_L \mid 1 \mid \exists x : A_S . B_L \mid \\
\& \{l : A_L\} \mid A_L \to B_L \mid \downarrow^S_L A_S \mid \Pi x : A_S . B_L
\]
Shared queue

$$A_S \trianglerighteq \uparrow^S_{A_L} A_L$$

$$A_L, B_L \trianglerighteq \bigoplus \{ l : A_L \} \mid A_L \otimes B_L \mid 1 \mid \exists x : A_S . B_L$$

$$\& \{ l : A_L \} \mid A_L \rightarrow \downarrow^S_{B_L} B_L \mid \downarrow^S_{A_S} A_S \mid \Pi x : A_S . B_L$$
Shared queue

\[ A_S \triangleq \uparrow^S_L A_L \]

\[ A_L, B_L \triangleq \bigoplus \{ l : A_L \} \mid A_L \otimes B_L \mid 1 \mid \exists x : A_S. B_L \]
\[ \& \{ l : A_L \} \mid A_L \rightarrow B_L \mid \downarrow^S_L A_S \mid \Pi x : A_S. B_L \]

queue \( A_S = \& \{ \text{enq: } \Pi x : A_S. \text{ queue } A_S, \text{ deq: } \bigoplus \{ \text{none: } \text{ queue } A_S, \text{ some: } \exists x : A_S. \text{ queue } A_S \} \} \)
Shared queue

\[ A_S \triangleq \uparrow^S A_L \]

\[ A_L, B_L \triangleq \bigoplus \{ l : A_L \} \mid A_L \otimes B_L \mid 1 \mid \exists x : A_S. B_L \mid \]

\[ \& \{ l : A_L \} \mid A_L \rightarrow B_L \mid \downarrow^S \mathbb{A} \mid \Pi x : A_S. B_L \]

queue \( A_S \) = \&\{ \text{enq : } \Pi x : A_S. \}

deq : \bigoplus \{ \text{none : } \}
\quad \text{queue } A_S, \quad \text{queue } A_S, \quad \text{some : } \exists x : A_S. \quad \text{queue } A_S \} \}
Shared queue

\[
A_S \triangleq \uparrow^S_L A_L
\]

\[
A_L, B_L \triangleq \bigoplus \{ l : A_L \} \mid A_L \otimes B_L \mid 1 \mid \exists x : A_S. B_L \mid \\
\land \{ l : A_L \} \mid A_L \rightarrowtail B_L \mid \downarrow^S_L A_S \mid \Pi x : A_S. B_L
\]

\[
\text{queue } A_S = \uparrow^S_L \land \{ \text{enq} : \Pi x : A_S. \text{ deq} : \bigoplus \{ \text{none} : \text{ queue } A_S, \text{ some} : \exists x : A_S. \text{ queue } A_S \} \}
\]
Shared queue

\[
A_S \triangleq \uparrow^S A_L
\]

\[
A_L, B_L \triangleq \bigoplus \{ l : A_L \} \mid A_L \otimes B_L \mid 1 \mid \exists x : A_S. B_L \\
\& \{ l : A_L \} \mid A_L \rightarrow B_L \mid \downarrow^S A_S \mid \Pi x : A_S. B_L
\]

queue \( A_S \) = \uparrow^S \& \{ \text{enq} : \Pi x : A_S. \downarrow^S \text{queue} A_S, \text{deq} : \bigoplus \{ \text{none} : \downarrow^S \text{queue} A_S, \text{some} : \exists x : A_S. \downarrow^S \text{queue} A_S \} \}
Shared queue

\[A_S = \uparrow^S L A_L\]

\[A_L, B_L = \oplus \{l : A_L\} \mid A_L \otimes B_L \mid 1 \mid \exists x : A_S. B_L \]

\& \{l : A_L\} \mid A_L \rightarrow B_L \mid \downarrow^S L A_S \mid \Pi x : A_S. B_L\]

queue \(A_S\) = \uparrow^S L \& \{\text{enq} : \Pi x : A_S. \downarrow^S L \text{queue } A_S, \text{deq} : \oplus \{\text{none} : \downarrow^S L \text{queue } A_S, \text{some} : \exists x : A_S. \downarrow^S L \text{queue } A_S\}\}\

\[\uparrow^S L \text{ and } \downarrow^S L \text{ prescribe acquire-release points, resp.}\]
Equi-synchronizing session type
Equi-synchronizing session type

Invariant guaranteeing that a process is released to the same type at which it was previously acquired, should it be released at all.
Equi-synchronizing session type

Invariant guaranteeing that a process is released to the same type at which it was previously acquired, should it be released at all.

Guarantees session fidelity w/o run-time checking upon acquire.
Equi-synchronizing session type

Invariant guaranteeing that a process is released to the same type at which it was previously acquired, should it be released at all.

Guarantees session fidelity w/o run-time checking upon acquire

Examples of equi-synchronizing session types:

\[
\begin{align*}
\text{queue } A_S &= \uparrow^S_L \& \{ \text{enq : } \Pi x : A_S. \ \downarrow^S_L \text{queue } A_S, \\
&\quad \text{deq : } \oplus \{ \text{none : } \downarrow^S_L \text{queue } A_S, \ \text{some : } \exists x : A_S. \ \downarrow^S_L \text{queue } A_S \} \}
\end{align*}
\]
Equi-synchronizing session type

Invariant guaranteeing that a process is released to the same type at which it was previously acquired, should it be released at all.

Guarantees session fidelity w/o run-time checking upon acquire

Examples of equi-synchronizing session types:

\[
\text{queue } A_S = \uparrow^S_L \land \{ \text{enq : } \Pi x : A_S. \downarrow^S_L \text{ queue } A_S, \\
\text{deq : } \oplus \{ \text{none : } \downarrow^S_L \text{ queue } A_S, \text{some : } \exists x : A_S. \downarrow^S_L \text{ queue } A_S \} \}
\]

\[
\text{queue } A = \land \{ \text{enq : } A \rightarrow \text{ queue } A, \\
\text{deq : } \oplus \{ \text{none : } 1, \text{some : } A \otimes \text{ queue } A \} \}
\]
Equi-synchronizing session type

Invariant guaranteeing that a process is released to the same type at which it was previously acquired, should it be released at all.

Guarantees session fidelity w/o run-time checking upon acquire

Examples of equi-synchronizing session types:

\[
\text{queue } A_S = \uparrow^S_L \& \{ \text{enq} : \Pi x : A_S. \ \downarrow^S_L \text{queue } A_S, \\
\quad \text{deq} : \oplus \{ \text{none} : \downarrow^S_L \text{queue } A_S, \ \text{some} : \exists x : A_S. \ \downarrow^S_L \text{queue } A_S \} \}
\]

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

Non-equie-synchronizing:

\[
\text{queue } A_S = \uparrow^S_L \& \{ \text{enq} : \Pi x : A_S. \ \downarrow^S_L \text{queue } A_S, \\
\quad \text{deq} : \oplus \{ \text{none} : \downarrow^S_L \uparrow^S_L 1, \ \text{some} : \exists x : A_S. \ \downarrow^S_L \text{queue } A_S \} \}
\]
Typing judgments
### Typing judgments

<table>
<thead>
<tr>
<th>$A_S$</th>
<th>$\triangleq$</th>
<th>$\uparrow^S_L A_L$</th>
</tr>
</thead>
</table>
| $A_L, B_L$ | $\triangleq$ | $\oplus \{ l : A_L \} | A_L \otimes B_L | 1 | \exists x : A_S. B_L |$
| | | & \{ l : A_L \} | A_L \rightarrow B_L | \downarrow^S_L A_S | \Pi x : A_S. B_L |
Typing judgments

\[ A_S \triangleq \uparrow^S_L A_L \]

\[ A_L, B_L \triangleq \bigoplus \{ l : A_L \} \mid A_L \otimes B_L \mid 1 \mid \exists x : A_S. B_L \mid \]

\[ \& \{ l : A_L \} \mid A_L \twoheadrightarrow B_L \mid \downarrow^S_L A_S \mid \Pi x : A_S. B_L \]

\[ \Gamma \vdash \Sigma \; P :: (x_S : A_S) \quad \text{shared process } P, \text{ providing session of type } A_S \]

\[ \text{along } x_S, \text{ using channels in } \Gamma \]

\[ \Gamma; \; \Delta \vdash \Sigma \; P :: (x_L : A_L) \quad \text{linear process } P, \text{ providing session of type } A_L \]

\[ \text{along } x_L, \text{ using channels in } \Gamma \text{ and } \Delta \]

\[ \Gamma \quad \text{shared (structural) context} \]

\[ \Delta \quad \text{linear context} \]
Typing judgments

\[ A_S \triangleq \uparrow^S_{\text{L}} A_L \]

\[ A_L, B_L \triangleq \oplus \{ l : A_L \} \mid A_L \otimes B_L \mid 1 \mid \exists x : A_S. B_L \]
\[ \& \{ l : A_L \} \mid A_L \rightarrow B_L \mid \downarrow^S_{\text{L}} A_S \mid \Pi x : A_S. B_L \]

\[ \Gamma \vdash \Sigma \ P :: (x_S : A_S) \quad \text{shared process } P, \text{ providing session of type } A_S \]
\[ \text{along } x_S, \text{ using channels in } \Gamma \]

\[ \Gamma; \Delta \vdash \Sigma \ P :: (x_L : A_L) \quad \text{linear process } P, \text{ providing session of type } A_L \]
\[ \text{along } x_L, \text{ using channels in } \Gamma \text{ and } \Delta \]

\[ \Gamma \]
\[ \Delta \]

shared (structural) context

linear context
Typing judgments

<table>
<thead>
<tr>
<th>$A_S$</th>
<th>$\hat{=} \uparrow^S_L A_L$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_L, B_L$</td>
<td>$\hat{=} \bigoplus {l : A_L} \mid A_L \otimes B_L \mid 1 \mid \exists x : A_S. B_L \mid &amp;{l : A_L} \mid A_L \multimap B_L \mid \downarrow^S_L A_S \mid \Pi x : A_S. B_L$</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\Gamma \vdash P :: (x_S : A_S) & \quad \text{shared process } P, \text{ providing session of type } A_S \\
\Gamma ; \Delta \vdash P :: (x_L : A_L) & \quad \text{linear process } P, \text{ providing session of type } A_L \\
\Gamma & \quad \text{shared (structural) context} \\
\Delta & \quad \text{linear context}
\end{align*}
\]
Typing judgments

$$A_S \trianglerighteq \uparrow^S_A L$$

$$A_L, B_L \trianglerighteq \oplus \{ l : A_L \} \mid A_L \otimes B_L \mid 1 \mid \exists x : A_S. B_L \mid$$

$$\& \{ l : A_L \} \mid A_L \rightarrow B_L \mid \downarrow^S_A S \mid \Pi x : A_S. B_L$$

$$\Gamma \vdash \Sigma \boxed{P} :: (x_S : A_S)$$  
shared process $P$, providing session of type $A_S$ along $x_S$, using channels in $\Gamma$

$$\Gamma ; \Delta \vdash \Sigma \boxed{P} :: (x_L : A_L)$$  
linear process $P$, providing session of type $A_L$ along $x_L$, using channels in $\Gamma$ and $\Delta$

$$\Gamma$$  
shared (structural) context

$$\Delta$$  
linear context
Typing judgments

\[ A_S \triangleq \uparrow^S A_L \]

\[ A_L, B_L \triangleq \oplus \{ l : A_L \} \mid A_L \otimes B_L \mid 1 \mid \exists x : A_S. B_L \mid \]

\[ \& \{ l : A_L \} \mid A_L \rightarrow B_L \mid \downarrow^S A_S \mid \Pi x : A_S. B_L \]

\[ \Gamma \vdash P :: (x_S : A_S) \quad \text{shared process } P, \text{ providing session of type } A_S \]

\[ \text{along } x_S, \text{ using channels in } \Gamma \]

\[ \Gamma; \Delta \vdash P :: (x_L : A_L) \quad \text{linear process } P, \text{ providing session of type } A_L \]

\[ \text{along } x_L, \text{ using channels in } \Gamma \text{ and } \Delta \]

\[ \Gamma \quad \text{shared (structural) context} \]

\[ \Delta \quad \text{linear context} \]
Typing judgments

\[ A_S \trianglelefteq \uparrow^S L A_L \]

\[ A_L, B_L \trianglelefteq \bigoplus \{ \overline{l} \mid A_L \otimes B_L \} \mid 1 \mid \exists x : A_S. B_L \mid \& \{ \overline{l} \mid A_L \} \mid A_L \rightarrow B_L \mid \downarrow^S L A_S \mid \Pi x : A_S. B_L \]

\[ \Gamma \vdash \Sigma P :: (x_S : A_S) \quad \text{shared process } P, \text{ providing session of type } A_S \text{ along } x_S, \text{ using channels in } \Gamma \]

\[ \Gamma ; \Delta \vdash \Sigma P :: (x_L : A_L) \quad \text{linear process } P, \text{ providing session of type } A_L \text{ along } x_L, \text{ using channels in } \Gamma \text{ and } \Delta \]

\[ \Gamma \quad \text{shared (structural) context} \]

\[ \Delta \quad \text{linear context} \]
Typing of acquire
Typing of acquire

\[
\frac{\Gamma, x_S : \uparrow^s L A_L; \ \Delta, x_L : A_L \vdash \Sigma Q x_L : (z_L : C_L)}{
\Gamma, x_S : \uparrow^s L A_L; \ \Delta \vdash \Sigma x_L \leftarrow \text{acquire } x_S ; Q x_L : (z_L : C_L)}
\]
Typing of acquire

\[
\frac{\Gamma, x_S : \uparrow^s L A_L; \Delta, x_L : A_L \vdash \Sigma Q x_L : (z_L : C_L)}{\Gamma, x_S : \uparrow^s L A_L; \Delta \vdash \Sigma x_L \leftarrow \text{acquire} \; x_S ; Q x_L : (z_L : C_L)} (T-\uparrow^s_{LL})
\]
Typing of acquire

\[
\frac{
\Gamma, x_S : \uparrow^L A_L; \ \Delta, x_L : A_L \vdash \Sigma Q_{x_L} :: (z_L : C_L)
}{
\Gamma, x_S : \uparrow^L A_L; \ \Delta \vdash \Sigma x_L \leftarrow \text{acquire} \ x_S ; Q_{x_L} :: (z_L : C_L)
}\quad (T-\uparrow^L L)
\]

\[
\frac{
\Gamma; \ \cdot \vdash \Sigma P_{x_L} :: (x_L : A_L)
}{
\Gamma \vdash \Sigma x_L \leftarrow \text{accept} \ x_S ; P_{x_L} :: (x_S : \uparrow^L L A_L)
}\quad (T-\uparrow^R L)
\]
Typing of acquire

\[
\frac{\Gamma, x_S : \uparrow^s_L A_L; \Delta, x_L : A_L \vdash \Sigma Q_{x_L} :: (z_L : C_L)}{\Gamma, x_S : \uparrow^s_L A_L; \Delta \vdash \Sigma x_L \leftarrow \text{acquire} \; x_S ; Q_{x_L} :: (z_L : C_L)} (T-\uparrow^s_{LL})
\]

\[
\frac{\Gamma; \cdot \vdash \Sigma P_{x_L} :: (x_L : A_L)}{\Gamma \vdash \Sigma x_L \leftarrow \text{accept} \; x_S ; P_{x_L} :: (x_S : \uparrow^s_L A_L)} (T-\uparrow^s_{LR})
\]
Typing of release
Typing of release

\[ \Gamma, x_S : A_S; \Delta \vdash \Sigma Q_{x_S} :: (z_L : C_L) \]

\[ \Gamma; \Delta, x_L : \downarrow^S_L A_S \vdash \Sigma x_S \leftarrow \text{release } x_L ; Q_{x_S} :: (z_L : C_L) \]
Typing of release

\[
\begin{align*}
\Gamma, x_S : A_S; \quad \Delta &\vdash \Sigma Q_{x_S} : (z_L : C_L) \\
\Gamma; \quad \Delta, x_L : \downarrow_L A_S &\vdash \Sigma x_S \leftarrow \text{release } x_L ; Q_{x_S} : (z_L : C_L)
\end{align*}
\]
Typing of release

\[
\frac{\Gamma, x_S : A_S; \Delta \vdash_{\Sigma} Q_{x_S} :: (z_L : C_L)}{\Gamma; \Delta, x_L : \downarrow^L_{\downarrow} A_S \vdash_{\Sigma} x_S \leftarrow \text{release } x_L ; Q_{x_S} :: (z_L : C_L)} \quad (T-\downarrow^L_{\downarrow} L)
\]

\[
\frac{\Gamma \vdash_{\Sigma} P_{x_S} :: (x_S : A_S)}{\Gamma; \cdot \vdash_{\Sigma} x_S \leftarrow \text{detach } x_L ; P_{x_S} :: (x_L : \downarrow^L_{\downarrow} A_S)} \quad (T-\downarrow^L_{\downarrow} R)
\]
Typing of release

\[
\frac{\Gamma, x_S : A_S; \Delta \vdash \Sigma Q_{x_S} :: (z_L : C_L)}{\Gamma; \Delta, x_L : \downarrow^S A_S \vdash \Sigma x_S \leftarrow \text{release } x_L ; Q_{x_S} :: (z_L : C_L)} \quad (T-\downarrow^S_{LL})
\]

\[
\frac{\Gamma \vdash \Sigma P_{x_S} :: (x_S : A_S)}{\Gamma; \cdot \vdash \Sigma x_S \leftarrow \text{detach } x_L ; P_{x_S} :: (x_L : \downarrow^S A_S)} \quad (T-\downarrow^S_{LR})
\]
Let’s program in Concurrent C0!
What about type safety?
What about type safety?

Preservation (aka session fidelity)
What about type safety?

Preservation (aka session fidelity)

- acquire-release and equi-synchronizing invariant guarantee that every linear process has a unique client
What about type safety?

Preservation (aka session fidelity)

- acquire-release and equi-synchronizing invariant guarantee that every linear process has a unique client
- state-altering exchange only along linear channels
What about type safety?

Preservation (aka session fidelity) ✔

- acquire-release and equi-synchronizing invariant guarantee that every linear process has a unique client
- state-altering exchange only along linear channels
What about type safety?

Preservation (aka session fidelity) ✓
- acquire-release and equi-synchronizing invariant guarantee that every linear process has a unique client
- state-altering exchange only along linear channels

Progress
What about type safety?

Preservation (aka session fidelity) ✔

- acquire-release and equi-synchronizing invariant guarantee that every linear process has a unique client
- state-altering exchange only along linear channels

Progress

- acquire-release amounts to “locking”
What about type safety?

Preservation (aka session fidelity) ✔
- acquire-release and equi-synchronizing invariant guarantee that every linear process has a unique client
- state-altering exchange only along linear channels

Progress
- acquire-release amounts to “locking”
- possibility of cyclic dependencies
What about type safety?

Preservation (aka session fidelity) ✓

- acquire-release and equi-synchronizing invariant guarantee that every linear process has a unique client
- state-altering exchange only along linear channels

Progress

- acquire-release amounts to “locking”
- possibility of cyclic dependencies
What about type safety?

Preservation (aka session fidelity)

- acquire-release and equi-synchronizing invariant guarantee that every linear process has a unique client
- state-altering exchange only along linear channels

Progress

- acquire-release amounts to “locking”
- possibility of cyclic dependencies

“a waits for b to release resource”
What about type safety?

Preservation (aka session fidelity) ✓
- acquire-release and equi-synchronizing invariant guarantee that every linear process has a unique client
- state-altering exchange only along linear channels

Progress
- acquire-release amounts to “locking”
- possibility of cyclic dependencies

“a waits for b to release resource”
What about type safety?

Preservation (aka session fidelity) ✓
- acquire-release and equi-synchronizing invariant guarantee that every linear process has a unique client
- state-altering exchange only along linear channels

Progress
- acquire-release amounts to “locking”
- possibility of cyclic dependencies

progress must permit deadlock

“a waits for b to release resource”
Progress must permit deadlock
Progress must permit deadlock

For example, dining philosophers:
Progress must permit deadlock

For example, dining philosophers:

\[sfork = \uparrow^S_L \downarrow^S_L \text{sfork}\]
Progress must permit deadlock

For example, dining philosophers:

\[
\text{sfork} = \uparrow^S \downarrow_L \downarrow_L \uparrow^S \text{sfork}
\]

Both potentially deadlocking and non-deadlocking version (Dijkstra's solution) encodable in \( \text{SILLS} \)
Progress must permit deadlock

For example, dining philosophers:

\[ sfork = \uparrow^S_L \downarrow_L^S sfork \]

Both potentially deadlocking and non-deadlocking version (Dijkstra's solution) encodable in SILLs

Progress theorem:

- blocked process: linear process attempting to acquire
- configuration is stuck only if all processes are blocked
Curry-Howard isomorphism revisited
Curry-Howard isomorphism revisited

Linear session types w/o manifest sharing:

• linear propositions — session types
• proofs — processes
• cut reduction — communication
Curry-Howard isomorphism revisited

Linear session types w/o manifest sharing:
- linear propositions — session types
- proofs — processes
- cut reduction — communication

Session types with manifest sharing:
- linear propositions — session types
- proofs — processes
- interleaving of:
  - proof construction — acquire
  - proof reduction — communication
  - proof deconstruction — release
Curry-Howard isomorphism revisited

Linear session types w/o manifest sharing:

- linear propositions — session types
- proofs — processes
- cut reduction — communication

Session types with manifest sharing:

- linear propositions — session types
- proofs — processes
- interleaving of:
  - proof construction — acquire
  - proof reduction — communication
  - proof deconstruction — release

deadlock: failure of proof construction
Recovering expressiveness of pi-calculus for session-typed concurrency
Expressiveness of linear session types
Expressiveness of linear session types

- In the simply-typed $\lambda$-calculus, the addition of a recursive type $U = U \rightarrow U$ recovers full expressiveness of the untyped $\lambda$-calculus.
Expressiveness of linear session types

- In the simply-typed $\lambda$-calculus, the addition of a recursive type $\mathcal{U} = \mathcal{U} \rightarrow \mathcal{U}$ recovers full expressiveness of the untyped $\lambda$-calculus.

- Unfortunately, recursion alone is insufficient to recover the expressiveness of the untyped $\pi$-calculus for linear session-typed $\pi$-calculus.
Expressiveness of linear session types

- In the simply-typed $\lambda$-calculus, the addition of a recursive type $\mathcal{U} = \mathcal{U} \rightarrow \mathcal{U}$ recovers full expressiveness of the untyped $\lambda$-calculus.

- Unfortunately, recursion alone is insufficient to recover the expressiveness of the untyped $\pi$-calculus for linear session-typed $\pi$-calculus.

both recursion and sharing are necessary
[Balzer & Pfenning & Toninho 2018]
Expressiveness of linear session types

- In the simply-typed $\lambda$-calculus, the addition of a recursive type $\mathcal{U} = \mathcal{U} \rightarrow \mathcal{U}$ recovers full expressiveness of the untyped $\lambda$-calculus.

- Unfortunately, recursion alone is insufficient to recover the expressiveness of the untyped $\pi$-calculus for linear session-typed $\pi$-calculus.

both recursion and sharing are necessary
[Balzer & Pfenning & Toninho 2018]

key: sharing of channels and acquire-release discipline
Expressiveness of linear session types

- In the simply-typed $\lambda$-calculus, the addition of a recursive type $U = U \to U$ recovers full expressiveness of the untyped $\lambda$-calculus.

- Unfortunately, recursion alone is insufficient to recover the expressiveness of the untyped $\pi$-calculus for linear session-typed $\pi$-calculus.

  - both recursion and sharing are necessary
  - [Balzer & Pfenning & Toninho 2018]
  - key: sharing of channels and acquire-release discipline
  - encoding of untyped asynchronous $\pi$-calculus into SILL$_S$ and proof of its operational and observational correspondence
Acquire-release introduces non-determinism
Acquire-release introduces non-determinism

\[
\text{coin} = \uparrow^S_L \oplus \{ \text{head} : \downarrow^S_L \text{coin}, \quad \text{tail} : \downarrow^S_L \text{coin} \} \]

Acquire-release introduces non-determinism

\[
\text{coin} = \uparrow^S_L \oplus \{\text{head} : \downarrow^S_L \text{coin}, \text{tail} : \downarrow^S_L \text{coin}\}
\]

Processes:

- **head**: \text{coin} sends “head” and recurs as \text{tail} process
- **tail**: \text{coin} sends “tail” and recurs as \text{head} process
- **flipper**: \textbf{1} \leftarrow \text{coin} reads (“flips”) given coin once and terminates
- **ndchoice**: \oplus\{yes : \textbf{1}, no : \textbf{1}\} spawns new coin and flipper processes and then reads (“flips”) the coin and terminates
Acquire-release introduces non-determinism

\[ \text{coin} = \uparrow^S_L \oplus \{ \text{head} : \downarrow^S_L \text{coin}, \text{tail} : \downarrow^S_L \text{coin} \} \]

Processes:

- **head**: \( \text{coin} \) sends “head” and recurs as \( \text{tail} \) process
- **tail**: \( \text{coin} \) sends “tail” and recurs as \( \text{head} \) process
- **flipper**: \( 1 \leftarrow \text{coin} \) reads (“flips”) given coin once and terminates
- **ndchoice**: \( \oplus \{ \text{yes} : 1, \text{no} : 1 \} \) spawns new coin and flipper processes and then reads (“flips”) the coin and terminates

\[ 1 \leftarrow \text{coin} \]

**Note:** Coin flipping requires prior acquire.
Acquire-release introduces non-determinism

\[
\text{coin} = \uparrow^S_L \oplus \{ \text{head} : \downarrow^S_L \text{coin}, \text{tail} : \downarrow^S_L \text{coin} \}
\]

Processes:

- **head**: \( \text{coin} \) sends “head” and recurs as **tail** process
- **tail**: \( \text{coin} \) sends “tail” and recurs as **head** process
- **flipper**: \( 1 \leftarrow \text{coin} \) reads (“flips”) given coin once and terminates
- **ndchoice**: \( \oplus \{ \text{yes} : 1, \text{no} : 1 \} \) spawns new coin and flipper processes and then reads (“flips”) the coin and terminates

- **coin flipping requires prior acquire**
- **value read by ndchoice depends on order of two acquires**
Untyped, asynchronous $\pi$-calculus encoding
Untyped, asynchronous $\pi$-calculus encoding

Characteristics of untyped, asynchronous $\pi$-calculus:

- **Arity**: a $\pi$-calculus channel may connect arbitrarily many processes
- **Non-determinism**: e.g., $c(x).P \mid \overline{c}(a) \mid c(y).Q$
- **Untyped**: $\pi$-calculus channels are untyped
Untyped, asynchronous $\pi$-calculus encoding

Characteristics of untyped, asynchronous $\pi$-calculus:

- **Arity:** a $\pi$-calculus channel may connect arbitrarily many processes
- **Non-determinism:** e.g., $c(x).P \mid \overline{c}(a) \mid c(y).Q$
- **Untyped:** $\pi$-calculus channels are untyped

Basic idea:

- $\pi$-calculus process $\rightarrow$ linear SILL$_S$ process of type $\mathbf{1}$
- $\pi$-calculus channel $\rightarrow$ shared SILL$_S$ process of universal type $\mathcal{U}_S$
Untyped, asynchronous π-calculus encoding

Characteristics of untyped, asynchronous π-calculus:

• Arity: a π-calculus channel may connect arbitrarily many processes
• Non-determinism: e.g., \( c(x).P \mid \overline{c}(a) \mid c(y).Q \)
• Untyped: π-calculus channels are untyped

Basic idea:

• π-calculus process \( \rightarrow \) linear SILL\(_S\) process of type \( \mathbf{1} \)
• π-calculus channel \( \rightarrow \) shared SILL\(_S\) process of universal type \( \mathcal{U}_S \)

\[
\mathcal{U}_S = \uparrow^S_L \& \{ \text{ins} : \Pi x : \mathcal{U}_S. \downarrow^S_L \mathcal{U}_S, \\
\text{del} : \oplus \{ \text{none} : \downarrow^S_L \mathcal{U}_S, \\
\text{some} : \exists x : \mathcal{U}_S. \downarrow^S_L \mathcal{U}_S \} \}
\]
Untyped, asynchronous $\pi$-calculus encoding

Characteristics of untyped, asynchronous $\pi$-calculus:

- **Arity**: a $\pi$-calculus channel may connect arbitrarily many processes
- **Non-determinism**: e.g., $c(x).P | \overline{c(a)} | c(y).Q$
- **Untyped**: $\pi$-calculus channels are untyped

Basic idea:

- $\pi$-calculus process $\rightarrow$ linear SILL$_S$ process of type $1$
- $\pi$-calculus channel $\rightarrow$ shared SILL$_S$ process of universal type $U_S$

\[
U_S = \uparrow^S_L & \{ \text{ins} : \Pi x : U_S. \downarrow^S_L U_S, \\
\text{del} : \oplus \{ \text{none} : \downarrow^S_L U_S, \text{some} : \exists x : U_S. \downarrow^S_L U_S \} \}
\]

unordered “buffer”
Untyped, asynchronous π-calculus encoding

\[ U_S = \uparrow^S_L \& \{ \text{ins} : \Pi x : U_S. \downarrow^S_L U_S, \]
\[ \text{del} : \oplus \{ \text{none} : \downarrow^S_L U_S, \]
\[ \text{some} : \exists x : U_S. \downarrow^S_L U_S \} \} \]
Untyped, asynchronous $\pi$-calculus encoding

\[ U_S = \{ \uparrow^L_S \} & \{ \text{ins} : \Pi x : U_S. \downarrow^L_S U_S, \]
\[ \text{del} : \bigoplus \{ \text{none} : \downarrow^L_S U_S, \]
\[ \text{some} : \exists x : U_S. \downarrow^L_S U_S \} \}

Processes:

empty: $U_S$  \hspace{2cm} empty buffer

elem: $U_S$  \hspace{2cm} non-deterministically deletes channel from arbitrary point in buffer using \texttt{ndchoice}

send: $\Pi x : U_S. 1 \leftarrow U_S$  \hspace{2cm} attempts to acquire given buffer and inserts received channel

poll_receive: $\exists x : U_S. 1 \leftarrow U_S$  \hspace{2cm} perpetually attempts to acquire given buffer for deletion until it succeeds
Untyped, asynchronous $\pi$-calculus encoding

$$\mathcal{U}_s = \uparrow^i_s \downarrow^i_s \mathcal{U}_s \uparrow^i_s \downarrow^i_s \mathcal{U}_s,$$

\begin{align*}
\text{del} : & \oplus \{ 
\text{none}: \downarrow^a_s \mathcal{U}_s, \\
\text{some}: & \exists x: \mathcal{U}_s . \downarrow^a_s \mathcal{U}_s \}
\end{align*}

Processes:

empty: \mathcal{U}_s \quad \text{empty buffer}

elem: \mathcal{U}_s \quad \text{non-deterministically deletes channel from arbitrary point in buffer using ndchoice}

send: \Pi x: \mathcal{U}_s . 1 \leftarrow \mathcal{U}_s \quad \text{attempts to acquire given buffer and inserts received channel}

poll_receive: \exists x: \mathcal{U}_s . 1 \leftarrow \mathcal{U}_s \quad \text{perpetually attempts to acquire given buffer for deletion until it succeeds}

Untyped, asynchronous $\pi$-calculus:

$$P \triangleq 0 \mid \overline{x}(y) \mid x(y).P \mid \nu x P \mid P_1 \mid P_2 \mid !P$$
Untyped, asynchronous π-calculus encoding

\[ \mathcal{U}_S = \uparrow^S_L \mathcal{U}_S, \quad \downarrow^S_L \mathcal{U}_S, \quad \text{ins} : \Pi x : \mathcal{U}_S. \downarrow^S_L \mathcal{U}_S, \quad \text{del} : \oplus \{ \text{none} : \downarrow^S_L \mathcal{U}_S, \quad \text{some} : \exists x : \mathcal{U}_S. \downarrow^S_L \mathcal{U}_S \} \}

Processes:

empty: \( \mathcal{U}_S \)  
empty buffer

elem: \( \mathcal{U}_S \)  
non-deterministically deletes channel from arbitrary point in buffer using \text{ndchoice}

send: \( \Pi x : \mathcal{U}_S. 1 \leftarrow \mathcal{U}_S \)  
attempts to acquire given buffer and inserts received channel

poll_receive: \( \exists x : \mathcal{U}_S. 1 \leftarrow \mathcal{U}_S \)  
perpetually attempts for deletion until it succeeds

Untyped, asynchronous π-calculus:

\[ P \overset{\Delta}{=} 0 \mid \overline{x}(y) \mid x(y).P \mid \nu x P \mid P_1 \mid P_2 \mid !P \]
Ongoing and future work
Deadlock-freedom [Balzer et al. 2019]
Deadlock-freedom [Balzer et al. 2019]

Two forms of waiting:

• waiting to synchronize:
• waiting to release:
Deadlock-freedom [Balzer et al. 2019]

Two forms of waiting:

• waiting to synchronize: \( a \rightarrow b \) “a waits for b to synchronize”
• waiting to release:
Deadlock-freedom [Balzer et al. 2019]

Two forms of waiting:

• waiting to synchronize:  \( a \rightarrow b \)  “a waits for b to synchronize”
• waiting to release:  \( a \rightarrow b \)  “a waits for b to release resource”
Deadlock-freedom [Balzer et al. 2019]

Two forms of waiting:

- waiting to synchronize: "a waits for b to synchronize"
- waiting to release: "a waits for b to release resource"
Deadlock-freedom [Balzer et al. 2019]

Two forms of waiting:
- waiting to synchronize: \( a \rightarrow b \) “\( a \) waits for \( b \) to synchronize”
- waiting to release: \( a \rightarrow b \) “\( a \) waits for \( b \) to release resource”
Deadlock-freedom [Balzer et al. 2019]

Two forms of waiting:

- waiting to synchronize: $a \rightarrow b$ “a waits for b to synchronize”
- waiting to release: $a \rightarrow b$ “a waits for b to release resource”

plain “locking-up” is no longer sufficient
Deadlock-freedom [Balzer et al. 2019]

Two forms of waiting:

• waiting to synchronize:  
  a waits for b to synchronize

• waiting to release:  
  a waits for b to release resource

plain “locking-up” is no longer sufficient

see “manifest deadlock-freedom” [Balzer et al. 2019]
Session types for Rust (with Mozilla Research)

- Development of a library for Manifest Sharing in Rust
- Explore application to Servo
Digital contracts (with Das & Hoffmann & Pfenning)

- Development of Nomos, a new digital contract language
- Static guarantees:
  - protocol enforcement (shared session types)
  - control over resource usage (resource analysis)
  - tracking of assets (linear type system)
Summary

• Message-passing concurrent programming
• Session types are a natural fit for typing such programs
• Session types have logical foundation via Curry-Howard correspondence
  • linear session types
  • shared session types
• Manifest sharing recovers the expressiveness of the untyped asynchronous pi-calculus
Summary

• Message-passing concurrent programming
• Session types are a natural fit for typing such programs
• Session types have logical foundation via Curry-Howard correspondence
  • linear session types
  • shared session types
• Manifest sharing recovers the expressiveness of the untyped asynchronous pi-calculus

Thank you for your attention!