# Intersections and Unions of Session Types

Coşku Acay    Frank Pfenning

Carnegie Mellon University
School of Computer Science

ITRS 2016

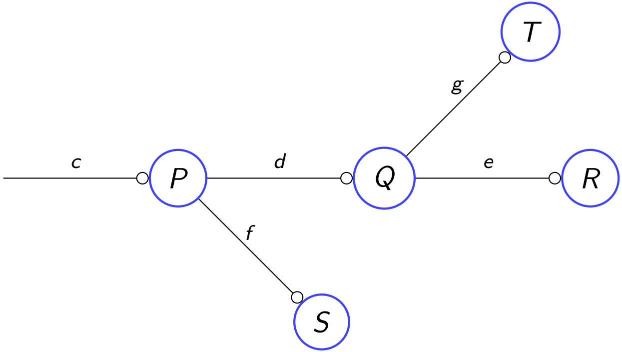
# Overview

# Plan

1. Background
   - Message-passing Concurrency
   - Session Types
   - Subtyping
   - Configurations and Reduction
2. Intersections and Unions
   - Intersection Types
   - Union Types
   - Reinterpreting Choice
3. Algorithmic System
4. Metatheory

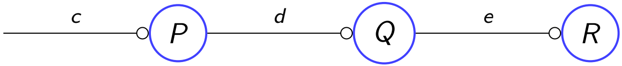
# Setting

- Processes represented as nodes
- Channels go between processes and represented as edges
- Each channel is "provided" by a specific process (e.g. $P$ provides $c$, $Q$ provides $d$ etc.): one-to-one correspondence between channels and processes



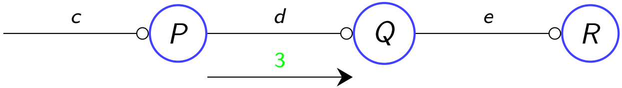
# Communication

- Processes compute internally
- Exchange messages along channels



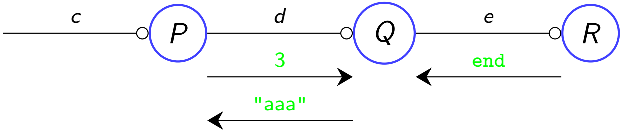
# Communication

- Processes compute internally
- Exchange messages along channels



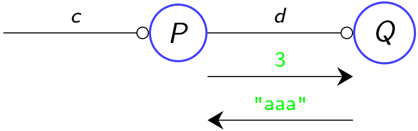
# Communication

- Processes compute internally
- Exchange messages along channels



# Communication

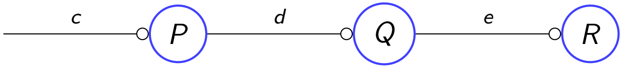
- Processes compute internally
- Exchange messages along channels




*Note that communication is synchronous.

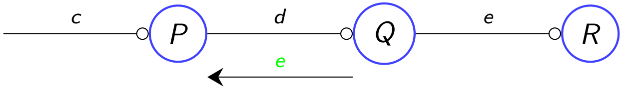
# Higher-order Messages

- Processes can also send channels they own



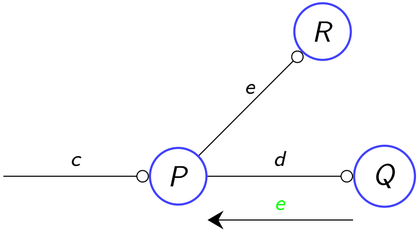
# Higher-order Messages

- Processes can also send channels they own



# Higher-order Messages

- Processes can also send channels they own



# Session Types

- Don't want to send `int` if expecting `string`
- Don't try to receive if other process is not sending

# Session Types

- Don't want to send `int` if expecting `string`
- Don't try to receive if other process is not sending

**Solution: Assign types to each channel**

# Session Types

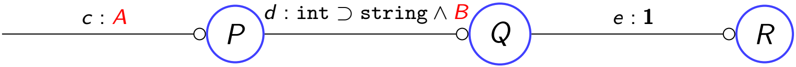
- Don't want to send `int` if expecting `string`
- Don't try to receive if other process is not sending

**Solution: Assign types to each channel** (from provider's perspective).

# Session Types

- Don't want to send `int` if expecting `string`
- Don't try to receive if other process is not sending

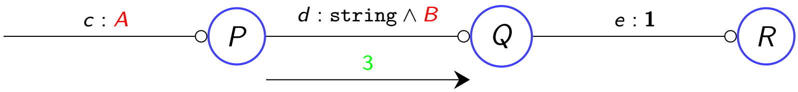
**Solution: Assign types to each channel** (from provider's perspective).



# Session Types

- Don't want to send `int` if expecting `string`
- Don't try to receive if other process is not sending

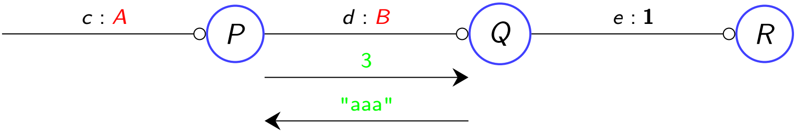
**Solution: Assign types to each channel** (from provider's perspective).



# Session Types

- Don't want to send `int` if expecting `string`
- Don't try to receive if other process is not sending

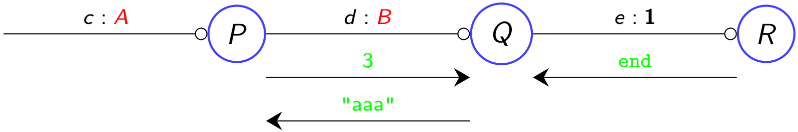
**Solution: Assign types to each channel** (from provider's perspective).



# Session Types

- Don't want to send `int` if expecting `string`
- Don't try to receive if other process is not sending

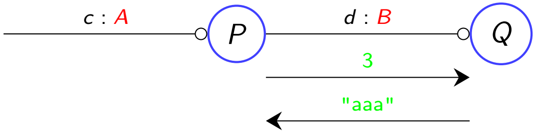
**Solution: Assign types to each channel** (from provider's perspective).



# Session Types

- Don't want to send `int` if expecting `string`
- Don't try to receive if other process is not sending

**Solution: Assign types to each channel** (from provider's perspective).



# Why linear?

- Sessions are resources: communicating along a channel consumes the old type
- Contraction would violate type safety
- Weakening would work, but we keep things simple

# Linear Propositions as Session Types

| | |
|---|---|
| $\mathbf{1}$ | send `end` and terminate |
| $A \otimes B$ | send channel of type $A$ and continue as $B$ |
| $\tau \wedge B$ | send value of type $\tau$ and continue as $B$ |
| $\oplus\{lab_k : A_k\}_{k \in I}$ | send $lab_i$ and continue as $A_i$ for some $i \in I$ |
| $A \multimap B$ | receive channel of type $A$ and continue as $B$ |
| $\tau \supset B$ | receive value of type $\tau$ and continue as $B$ |
| $\&\{lab_k : A_k\}_{k \in I}$ | receive $lab_i$ and continue as $A_i$ for some $i \in I$ |
| $\mu t.A_t$ | (equi-)recursive type |

# Linear Propositions as Session Types

| | |
|---|---|
| $\mathbf{1}$ | send `end` and terminate |
| $A \otimes B$ | send channel of type $A$ and continue as $B$ |
| $\tau \wedge B$ | send value of type $\tau$ and continue as $B$ |
| $\oplus\{lab_k : A_k\}_{k \in I}$ | send $lab_i$ and continue as $A_i$ for some $i \in I$ |
| $A \multimap B$ | receive channel of type $A$ and continue as $B$ |
| $\tau \supset B$ | receive value of type $\tau$ and continue as $B$ |
| $\&\{lab_k : A_k\}_{k \in I}$ | receive $lab_i$ and continue as $A_i$ for some $i \in I$ |
| $\mu t.A_t$ | (equi-)recursive type |

**Example: Queue Interface**

```
type queue = &{ enq : A -o queue
              , deg : +{none : 1, some : A * queue}
              }
```

# Proof Terms as Concurrent Processes

$P, Q, R ::=$

| | |
|---|---|
| $x \leftarrow P_x \,;\, Q_x$ | cut (spawn) |
| $c \leftarrow d$ | id (forward) |
| $\mathrm{close}\ c \mid \mathrm{wait}\ c \,;\, P$ | $\mathbf{1}$ |
| $\mathrm{send}\ c\ (y \leftarrow P_y) \,;\, Q \mid x \leftarrow \mathrm{recv}\ c \,;\, R_x$ | $A \otimes B$, $A \multimap B$ |
| $c.lab \,;\, P \mid \mathrm{case}\ c\ \mathrm{of}\ \{lab_k \to Q_k\}_{k \in I}$ | $\&\{lab_k : A_k\}_{k \in I}$, $\oplus\{lab_k : A_k\}_{k \in I}$ |

# Example: An Implementation of Queues

```
type queue = &{ enq : A -o queue
              , deq : +{none : 1, some : A * queue}
              }



              empty : queue
              q <- empty = case q of
                enq -> x <- recv q;
                       e <- empty;
                       q <- elem x e
                deq -> q.none; close q

              elem : A -o queue -o queue
              q <- elem x r = case q of
                enq -> y <- recv q;
                       r.enq; send r y;
                       q <- elem x r
                deq -> q.some; send q x;
                       q <- r
```

# Process Typing

Typing judgement has the form $\Psi \vdash_\eta P :: (c : A)$ meaning "process $P$ offers along channel $c$ the session $A$ under the context $\Psi$." $\eta$ tracks recursive variables.

# Process Typing

Typing judgement has the form $\Psi \vdash_\eta P :: (c : A)$ meaning "process $P$ offers along channel $c$ the session $A$ under the context $\Psi$." $\eta$ tracks recursive variables.

Some examples:

$$\frac{}{c : A \vdash d \leftarrow c :: (d : A)}\ \mathtt{id}$$

$$\frac{\Psi \vdash P_c :: (c : A) \quad \Psi', c : A \vdash Q_c :: (d : D)}{\Psi, \Psi' \vdash c \leftarrow P_c\, ;\ Q_c :: (d : D)}\ \mathtt{cut}$$

$$\frac{}{\emptyset \vdash \mathrm{close}\ c :: (c : \mathbf{1})}\ \mathbf{1}\mathtt{R} \qquad \frac{\Psi \vdash P :: (d : A)}{\Psi, c : \mathbf{1} \vdash \mathrm{wait}\ c\, ;\ P :: (d : A)}\ \mathbf{1}\mathtt{L}$$

$$\frac{\Psi \vdash P :: (d : A) \quad \Psi' \vdash Q :: (c : B)}{\Psi, \Psi' \vdash \mathrm{send}\ c\ (d \leftarrow P_d)\, ;\ Q :: (c : A \otimes B)}\ \otimes\mathtt{R}$$

# Subtyping

- Width and depth subtyping for *n*-ary choices
- Width: $\&\{lab_k : A_k\}_{k \in I} \leq \&\{lab_k : A_k\}_{k \in J}$ whenever $J \subseteq I$
- Depth: $\&\{lab_k : A_k\}_{k \in I} \leq \&\{lab_k : A'_k\}_{k \in I}$ whenever $A_i \leq A'_i$

# Subtyping

- Width and depth subtyping for *n*-ary choices
- Width: $\&\{lab_k : A_k\}_{k \in I} \leq \&\{lab_k : A_k\}_{k \in J}$ whenever $J \subseteq I$
- Depth: $\&\{lab_k : A_k\}_{k \in I} \leq \&\{lab_k : A'_k\}_{k \in I}$ whenever $A_i \leq A'_i$
- Defined coinductively because of recursive types

# Subtyping

- Width and depth subtyping for *n*-ary choices
- Width: $\&\{lab_k : A_k\}_{k \in I} \leq \&\{lab_k : A_k\}_{k \in J}$ whenever $J \subseteq I$
- Depth: $\&\{lab_k : A_k\}_{k \in I} \leq \&\{lab_k : A'_k\}_{k \in I}$ whenever $A_i \leq A'_i$
- Defined coinductively because of recursive types

Introduced to process typing using subsumption:

$$\frac{\Psi \vdash_\eta P :: (c : A') \quad A' \leq A}{\Psi \vdash_\eta P :: (c : A)}\ \texttt{SubR}$$

$$\frac{\Psi, c : A' \vdash_\eta P :: (d : B) \quad A \leq A'}{\Psi, c : A \vdash_\eta P :: (d : B)}\ \texttt{SubL}$$

# Configurations

- A processes by itself is not very useful in concurrent setting
- Need to be able to talk about interactions

# Configurations

- A processes by itself is not very useful in concurrent setting
- Need to be able to talk about interactions
- Use a process configuration, which is simply a set of labelled processes: $\Omega = \texttt{proc}_{c_1}(P_1), \ldots, \texttt{proc}_{c_n}(P_n)$.

# Configurations

- A processes by itself is not very useful in concurrent setting
- Need to be able to talk about interactions
- Use a process configuration, which is simply a set of labelled processes: $\Omega = \texttt{proc}_{c_1}(P_1), \ldots, \texttt{proc}_{c_n}(P_n)$.
- Typing judgement, $\models \Omega :: \Psi$, imposes a tree structure (ensures linearity)

# Reduction

Configurations reduce by interaction. Some examples:

$$
\begin{aligned}
\texttt{id} \quad & : \mathtt{proc}_c(c \leftarrow d) \multimap \{c = d\} \\
\texttt{cut} \quad & : \mathtt{proc}_c(x \leftarrow P_x \,;\, Q_x) \multimap \{\exists a.\mathtt{proc}_a(P_a) \otimes \mathtt{proc}_c(Q_a)\} \\
\texttt{one} \quad & : \mathtt{proc}_c(\mathrm{close}\ c) \otimes \mathtt{proc}_d(\mathrm{wait}\ c \,;\, P) \multimap \{\mathtt{proc}_d(P)\}
\end{aligned}
$$

# Plan

# Intersections and Unions

What if we want to track more properties of queues?

# Intersections and Unions

What if we want to track more properties of queues? Empty, non-empty, even length?

# Intersections and Unions

What if we want to track more properties of queues? Empty, non-empty, even length?

These can be defined in the base system:

```
type empty-queue = &{ enq : A -o queue
                    , deq : +{none : 1}
                    }

type nonempty-queue = &{ enq : A -o queue
                       , deq : +{some : A * queue}
                       }
```

# Intersections and Unions

What if we want to track more properties of queues? Empty, non-empty, even length?

These can be defined in the base system:

```
type empty-queue = &{ enq : A -o queue
                    , deg : +{none : 1}
                    }

type nonempty-queue = &{ enq : A -o queue
                       , deg : +{some : A * queue}
                       }
```

**However, there is no way to properly track them!**

**We cannot track multiple refinements.**

**We cannot track multiple refinements.** Consider `concat : queue -o queue -o queue` that concatenates two queues. It has many types but no most general type:

```
concat : empty-queue -o empty-queue -o empty-queue
concat : queue -o nonempty-queue -o nonempty-queue
concat : nonempty-queue -o queue -o nonempty-queue
```

# Intersection Types

- Intersection of two types: $A \sqcap B$
- $c : A \sqcap B$ if channel c offers both behaviors simultaneously

# Intersection Types

- Intersection of two types: $A \sqcap B$
- $c : A \sqcap B$ if channel c offers both behaviors simultaneously

$$\frac{\Psi \vdash_\eta P :: (c : A) \quad \Psi \vdash_\eta P :: (c : B)}{\Psi \vdash_\eta P :: (c : A \sqcap B)} \sqcap\mathrm{R}$$

$$\frac{\Psi, c : A \vdash_\eta P :: (d : D)}{\Psi, c : A \sqcap B \vdash_\eta P :: (d : D)} \sqcap\mathrm{L}_1 \qquad \frac{\Psi, c : B \vdash_\eta P :: (d : D)}{\Psi, c : A \sqcap B \vdash_\eta P :: (d : D)} \sqcap\mathrm{L}_2$$

# Intersections Solve the Previous Problem

We can now specify multiple behavioral properties:

```
concat : empty-queue -o empty-queue -o empty-queue
     and queue -o nonempty-queue -o nonempty-queue
     and nonempty-queue -o queue -o nonempty-queue
```

# Union Types

- Union of two types: $A \sqcup B$
- $c : A \sqcup B$ if channel c offers either behavior

# Union Types

- Union of two types: $A \sqcup B$
- $c : A \sqcup B$ if channel c offers either behavior
- Dual to intersections

# Union Types

- Union of two types: $A \sqcup B$
- $c : A \sqcup B$ if channel c offers either behavior
- Dual to intersections

$$\frac{\Psi \vdash_\eta P :: (c : A)}{\Psi \vdash_\eta P :: (c : A \sqcup B)} \sqcup\mathrm{R}_1 \qquad \frac{\Psi \vdash_\eta P :: (c : B)}{\Psi \vdash_\eta P :: (c : A \sqcup B)} \sqcup\mathrm{R}_2$$

$$\frac{\Psi, c : A \vdash_\eta P :: (d : D) \quad \Psi, c : B \vdash_\eta P :: (d : D)}{\Psi, c : A \sqcup B \vdash_\eta P :: (d : D)} \sqcup\mathrm{L}$$

# Reasons for Adding Unions

- Maintain the symmetry of the system
- Makes working with internal choice more convenient
- Interpretation of internal choice (we will explain later)

# Reasons for Adding Unions

- Maintain the symmetry of the system
- Makes working with internal choice more convenient
- Interpretation of internal choice (we will explain later)

We can also write things like:

```
type queue = empty-queue or nonempty-queue
```

# Reinterpreting Choice

Consider $\&\{\texttt{inl} : A, \texttt{inr} : B\}$

# Reinterpreting Choice

Consider $\&\{\texttt{inl} : A, \texttt{inr} : B\}$

This type says: "I will act as $A$ if you send me `inl` *and* I will act as $B$ if you send me `inr`."

# Reinterpreting Choice

Consider $\&\{\texttt{inl} : A, \texttt{inr} : B\}$

This type says: "I will act as $A$ if you send me `inl` *and* I will act as $B$ if you send me `inr`."

Interpreting *and* as $\sqcap$ gives
$\&\{\texttt{inl} : A, \texttt{inr} : B\} \approx \&\{\texttt{inl} : A\} \sqcap \&\{\texttt{inr} : B\}$.

# Reinterpreting Choice - General Case

Generalizing to *n*-ary choice and dualising gives:

$$\&\{lab_k : A_k\}_{k \in I} \triangleq \bigsqcap_{k \in I} \&\{lab_k : A_k\}$$

$$\oplus\{lab_k : A_k\}_{k \in I} \triangleq \bigsqcup_{k \in I} \oplus\{lab_k : A_k\}$$

Easy to verify these definitions satisfy the typing rules.

# Reinterpreting Choice - General Case

Generalizing to *n*-ary choice and dualising gives:

$$\&\{lab_k : A_k\}_{k \in I} \triangleq \bigsqcap_{k \in I} \&\{lab_k : A_k\}$$

$$\oplus\{lab_k : A_k\}_{k \in I} \triangleq \bigsqcup_{k \in I} \oplus\{lab_k : A_k\}$$

Easy to verify these definitions satisfy the typing rules.

Suggests treating intersections and unions as implicit choice.

# Plan

# Algorithmic Subtyping

Idea: make $\leq \sqcap\mathtt{L}_{\{1,2\}}$ and $\leq \sqcup\mathtt{R}_{\{1,2\}}$ invertible so we can apply eagerly.

$$\frac{A_{\{1,2\}} \leq B}{A_1 \sqcap A_2 \leq B} \leq \sqcap\mathtt{L}_{\{1,2\}} \quad \longrightarrow \quad \frac{\alpha, A_1, A_2 \Rightarrow \beta}{\alpha, A_1 \sqcap A_2 \Rightarrow \beta} \Rightarrow \sqcap\mathtt{L}$$

$$\frac{A \leq B_{\{1,2\}}}{A \leq B_1 \sqcup B_2} \leq \sqcup\mathtt{R}_{\{1,2\}} \quad \longrightarrow \quad \frac{\alpha \Rightarrow \beta, A_1, A_2}{\alpha \Rightarrow \beta, A_1 \sqcup A_2} \Rightarrow \sqcup\mathtt{R}$$

# Algorithmic Subtyping

Idea: make $\leq \sqcap\mathtt{L}_{\{1,2\}}$ and $\leq \sqcup\mathtt{R}_{\{1,2\}}$ invertible so we can apply eagerly.

$$\frac{A_{\{1,2\}} \leq B}{A_1 \sqcap A_2 \leq B} \leq \sqcap\mathtt{L}_{\{1,2\}} \quad \longrightarrow \quad \frac{\alpha, A_1, A_2 \Rightarrow \beta}{\alpha, A_1 \sqcap A_2 \Rightarrow \beta} \Rightarrow \sqcap\mathtt{L}$$

$$\frac{A \leq B_{\{1,2\}}}{A \leq B_1 \sqcup B_2} \leq \sqcup\mathtt{R}_{\{1,2\}} \quad \longrightarrow \quad \frac{\alpha \Rightarrow \beta, A_1, A_2}{\alpha \Rightarrow \beta, A_1 \sqcup A_2} \Rightarrow \sqcup\mathtt{R}$$

Also admits distributivity:

$$(A_1 \sqcup B) \sqcap (A_2 \sqcup B) \equiv (A_1 \sqcap A_2) \sqcup B$$

$$(A_1 \sqcup A_2) \sqcap B \equiv (A_1 \sqcap B) \sqcup (A_2 \sqcap B)$$

Turns out to be necessary for soundness of algorithmic typing.

# Algorithmic Type Checking

- Make $\sqcap L_{\{1,2\}}$ and $\sqcup R_{\{1,2\}}$ invertible so we can apply eagerly.

# Algorithmic Type Checking

- Make $\sqcap\mathrm{L}_{\{1,2\}}$ and $\sqcup\mathrm{R}_{\{1,2\}}$ invertible so we can apply eagerly.
- Delay subtyping to `id`
- Label `cut` with it's type

# Algorithmic Type Checking

- Make $\sqcap\mathtt{L}_{\{1,2\}}$ and $\sqcup\mathtt{R}_{\{1,2\}}$ invertible so we can apply eagerly.
- Delay subtyping to `id`
- Label `cut` with it's type

$$\frac{\Psi, c : A_{1,2} \vdash_\eta P :: (d : D)}{\Psi, c : A_1 \sqcap A_2 \vdash_\eta P :: (d : D)}\ \sqcap\mathtt{L}_{1,2} \quad \rightarrow \quad \frac{\Psi, c : (\alpha, A, B) \Vdash_\eta P :: (d : \beta)}{\Psi, c : (\alpha, A \sqcap B) \Vdash_\eta P :: (d : \beta)}\ \sqcap\mathtt{L}$$

$$\frac{\Psi \vdash_\eta P :: (c : A_{1,2})}{\Psi \vdash_\eta P :: (c : A_1 \sqcup A_2)}\ \sqcup\mathtt{R}_{1,2} \quad \rightarrow \quad \frac{\Psi \Vdash_\eta P :: (c : A, B, \alpha)}{\Psi \Vdash_\eta P :: (c : A \sqcup B, \alpha)}\ \sqcup\mathtt{R}$$

# Algorithmic Type Checking

- Make $\sqcap\mathtt{L}_{\{1,2\}}$ and $\sqcup\mathtt{R}_{\{1,2\}}$ invertible so we can apply eagerly.
- Delay subtyping to `id`
- Label `cut` with it's type

$$\frac{\Psi, c : A_{1,2} \vdash_\eta P :: (d : D)}{\Psi, c : A_1 \sqcap A_2 \vdash_\eta P :: (d : D)}\ \sqcap\mathtt{L}_{1,2} \quad \to \quad \frac{\Psi, c : (\alpha, A, B) \Vdash_\eta P :: (d : \beta)}{\Psi, c : (\alpha, A \sqcap B) \Vdash_\eta P :: (d : \beta)}\ \sqcap\mathtt{L}$$

$$\frac{\Psi \vdash_\eta P :: (c : A_{1,2})}{\Psi \vdash_\eta P :: (c : A_1 \sqcup A_2)}\ \sqcup\mathtt{R}_{1,2} \quad \to \quad \frac{\Psi \Vdash_\eta P :: (c : A, B, \alpha)}{\Psi \Vdash_\eta P :: (c : A \sqcup B, \alpha)}\ \sqcup\mathtt{R}$$

$$\frac{\alpha \Rightarrow \beta}{c : \alpha \Vdash_\eta d \leftarrow c :: (d : \beta)}\ \mathtt{id}$$

$$\frac{\Psi \Vdash_\eta P_c :: (c : A) \quad \Psi', c : A \Vdash_\eta Q_c :: (d : \alpha)}{\Psi, \Psi' \Vdash_\eta c : A \leftarrow P_c\, ;\, Q_c :: (d : \alpha)}\ \mathtt{cut}$$

# Results

**Theorem (Completeness of Algorithmic Subtyping)**

*Algorithmic subtyping is complete with respect to declarative subtyping.*

**Theorem (Equivalence of Algorithmic Typing)**

*Algorithmic typing is sound and complete with respect to declarative typing.*

# Plan

# Type Safety

We have proved progress and preservation for the system extended with intersections and unions.

# Type Safety

We have proved progress and preservation for the system extended with intersections and unions.

Progress $\rightarrow$ deadlock-freedom

Type preservation $\rightarrow$ session fidelity

# Progress

## Theorem (Progress)

*If* $\models \Omega :: \Psi$ *then either*

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

## Proof.

By induction on $\models \Omega :: \Psi$ followed by a nested induction on the typing of the root process. When two processes are involved, we also need inversion on client's typing. □

*A process is poised if it is waiting to communicate with its client.

# Type Preservation

## Theorem (Preservation)

*If* $\models \Omega :: \Psi$ *and* $\Omega \longrightarrow \Omega'$ *then* $\models \Omega' :: \Psi$.

## Proof.

By inversion on $\Omega \longrightarrow \Omega'$, followed by induction on the typing judgments of the involved processes. Each branch requires a hand-rolled induction hypothesis. $\square$

# Conclusion and Highlights

We introduced intersection and union types to a session-typed process calculus and demonstrated their usefulness.

- Unions work naturally. The elimination rule we give has been shown unsound in the presence of effects (even non-termination).

# Conclusion and Highlights

We introduced intersection and union types to a session-typed process calculus and demonstrated their usefulness.

- Unions work naturally. The elimination rule we give has been shown unsound in the presence of effects (even non-termination).
- More general than refinement system of Freeman and Pfenning

# Conclusion and Highlights

We introduced intersection and union types to a session-typed process calculus and demonstrated their usefulness.

- Unions work naturally. The elimination rule we give has been shown unsound in the presence of effects (even non-termination).
- More general than refinement system of Freeman and Pfenning
- Subtyping resembles Gentzen's multiple conclusion calculus

# Conclusion and Highlights

We introduced intersection and union types to a session-typed process calculus and demonstrated their usefulness.

- Unions work naturally. The elimination rule we give has been shown unsound in the presence of effects (even non-termination).
- More general than refinement system of Freeman and Pfenning
- Subtyping resembles Gentzen's multiple conclusion calculus
- Algorithmic typing mirrors subtyping

# Future Work

- Simple: integrate a functional language, extend to shared channels and asynchronous communication

# Future Work

- Simple: integrate a functional language, extend to shared channels and asynchronous communication
- More interesting: Add polymorphism and abstract types
  - Polymorphism is non-trivial with equirecursive types

# Future Work

- Simple: integrate a functional language, extend to shared channels and asynchronous communication
- More interesting: Add polymorphism and abstract types
  - Polymorphism is non-trivial with equirecursive types
- Applications other than refinements?

# The End