COMP3161/9164 Concepts of Programming Languages

# Concurrency and Session Types

Rob Sison
UNSW

Term 3 2025

# Definitions

# Definitions

## Definition

*Concurrency* is an abstraction for the programmer, allowing programs to be structured as multiple threads of control, called *processes*. These processes may communicate in various ways.

**Example Applications**: Servers, OS Kernels, GUI applications.

# Definitions

## Definition

*Concurrency* is an abstraction for the programmer, allowing programs to be structured as multiple threads of control, called *processes*. These processes may communicate in various ways.

**Example Applications**: Servers, OS Kernels, GUI applications.

## Anti-definition

Concurrency is **not** *parallelism*, which is a means to exploit multiprocessing hardware in order to improve performance.

# Sequential vs Concurrent

We could consider a *sequential* program as a sequence (or *total order*) of *actions*:

$$\bullet \longrightarrow \bullet \longrightarrow \bullet \longrightarrow \bullet \longrightarrow \bullet \longrightarrow \bullet \rightarrow \cdots$$

# Sequential vs Concurrent

We could consider a *sequential* program as a sequence (or *total order*) of *actions*:

$$\bullet \longrightarrow \bullet \longrightarrow \bullet \longrightarrow \bullet \longrightarrow \bullet \longrightarrow \bullet \rightarrow \cdots$$

The ordering here is "happens before". For example, processor instructions:

$$\texttt{LD R0,X} \longrightarrow \texttt{LDI R1,5} \longrightarrow \texttt{ADD R0,R1} \longrightarrow \texttt{ST X,R0}$$

# Sequential vs Concurrent

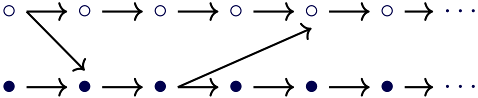
We could consider a *sequential* program as a sequence (or *total order*) of *actions*:

$$\bullet \longrightarrow \bullet \longrightarrow \bullet \longrightarrow \bullet \longrightarrow \bullet \longrightarrow \bullet \longrightarrow \cdots$$

The ordering here is "happens before". For example, processor instructions:

$$\texttt{LD R0,X} \longrightarrow \texttt{LDI R1,5} \longrightarrow \texttt{ADD R0,R1} \longrightarrow \texttt{ST X,R0}$$

A concurrent program is not a total order but a partial order.



This means that there are now multiple possible *interleavings* of these actions — our program is non-deterministic where the interleaving is selected by the scheduler.

# A Sobering Realisation

How many scenarios are there for a program with $n$ processes consisting of $m$ steps each?

# A Sobering Realisation

How many scenarios are there for a program with $n$ processes consisting of $m$ steps each?

| | $n = 2$ | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| $m = 2$ | 6 | 90 | 2520 | 113400 | $2^{22.8}$ |
| 3 | 20 | 1680 | $2^{18.4}$ | $2^{27.3}$ | $2^{36.9}$ |
| 4 | 70 | 34650 | $2^{25.9}$ | $2^{38.1}$ | $2^{51.5}$ |
| 5 | 252 | $2^{19.5}$ | $2^{33.4}$ | $2^{49.1}$ | $2^{66.2}$ |
| 6 | 924 | $2^{24.0}$ | $2^{41.0}$ | $2^{60.2}$ | $2^{81.1}$ |

# A Sobering Realisation

How many scenarios are there for a program with $n$ processes consisting of $m$ steps each?

| | $n = 2$ | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| $m = 2$ | 6 | 90 | 2520 | 113400 | $2^{22.8}$ |
| 3 | 20 | 1680 | $2^{18.4}$ | $2^{27.3}$ | $2^{36.9}$ |
| 4 | 70 | 34650 | $2^{25.9}$ | $2^{38.1}$ | $2^{51.5}$ |
| 5 | 252 | $2^{19.5}$ | $2^{33.4}$ | $2^{49.1}$ | $2^{66.2}$ |
| 6 | 924 | $2^{24.0}$ | $2^{41.0}$ | $2^{60.2}$ | $2^{81.1}$ |

$$\frac{(nm)!}{m!^n}$$

# Shared-variable vs Message-passing

Explicit communication between concurrent processes falls broadly into two classes:

- Shared-variable (or shared-memory) concurrency: Communication occurs by reading/writing shared state.

**Question**

Why would you see this more in imperative programming languages rather than functional ones?

# Shared-variable vs Message-passing

Explicit communication between concurrent processes falls broadly into two classes:

- Shared-variable (or shared-memory) concurrency:
  Communication occurs by reading/writing shared state.

> **Question**
>
> Why would you see this more in imperative programming languages rather than functional ones?

- Message-passing concurrency:
  Communication occurs by sending/receiving on channels.

# Shared-variable vs Message-passing

Explicit communication between concurrent processes falls broadly into two classes:

- Shared-variable (or shared-memory) concurrency:
  Communication occurs by reading/writing shared state.

> **Question**
>
> Why would you see this more in imperative programming languages rather than functional ones?

- Message-passing concurrency:
  Communication occurs by sending/receiving on channels.

  That said, message passing can be:

  - Synchronous: Sending has to wait for the receiver.
  - Asynchronous: Sending "leaves a message" for the receiver.

# Shared Variables and Synchronisation

If you don't synchronise different threads' accesses to shared variables, you can end up with unwanted behaviour.

| **var** $x := 0$ | |
|---|---|
| **while** $x < 20$ **do** | **while** $x > -20$ **do** |
| **var** $p := x;$ | **var** $q := x;$ |
| $x := p + 1;$ | $x := q - 1;$ |

## Question

How many loop iterations?

# Shared Variables and Synchronisation

If you don't synchronise different threads' accesses to shared variables, you can end up with unwanted behaviour.

| **var** $x := 0$ | |
|---|---|
| **while** $x < 20$ **do** | **while** $x > -20$ **do** |
| **var** $p := x$; | **var** $q := x$; |
| $x := p + 1$; | $x := q - 1$; |

**Question**

How many loop iterations?

Who knows!

# Shared Variables and Synchronisation

If you don't synchronise different threads' accesses to shared variables, you can end up with unwanted behaviour.

| **var** $x := 0$ | |
|---|---|
| **while** $x < 20$ **do** | **while** $x > -20$ **do** |
| **var** $p := x;$ | **var** $q := x;$ |
| $x := p + 1;$ | $x := q - 1;$ |

## Question

How many loop iterations?

Who knows! Plus, data races are undefined behaviour in C!

# Shared Variables and Synchronisation

If you don't synchronise different threads' accesses to shared variables, you can end up with unwanted behaviour.

| **var** $x := 0$ | |
|---|---|
| **while** $x < 20$ **do** | **while** $x > -20$ **do** |
| **var** $p := x;$ | **var** $q := x;$ |
| $x := p + 1;$ | $x := q - 1;$ |

**Question**

How many loop iterations?

Who knows! Plus, data races are undefined behaviour in C!

*Data races* are unsynchronised concurrent accesses to shared variables by different threads.

# Atomicity of Critical Sections

The basic unit of synchronisation we would like to implement is to group multiple steps into one atomic step, called a *critical section*.

# Atomicity of Critical Sections

The basic unit of synchronisation we would like to implement is to group multiple steps into one atomic step, called a *critical section*. A sketch of the problem can be outlined as follows:

| | |
|---|---|
| **forever do** | **forever do** |
| *non-critical section* | *non-critical section* |
| *pre-protocol* | *pre-protocol* |
| **critical section** | **critical section** |
| *post-protocol* | *post-protocol* |

# Atomicity of Critical Sections

The basic unit of synchronisation we would like to implement is to group multiple steps into one atomic step, called a *critical section*. A sketch of the problem can be outlined as follows:

| | |
|---|---|
| **forever do**<br>    *non-critical section*<br>    *pre-protocol*<br>    **critical section**<br>    *post-protocol* | **forever do**<br>    *non-critical section*<br>    *pre-protocol*<br>    **critical section**<br>    *post-protocol* |

The non-critical section models the possibility that a process may do something else. It can take any amount of time (even infinite).

# Atomicity of Critical Sections

The basic unit of synchronisation we would like to implement is to group multiple steps into one atomic step, called a *critical section*. A sketch of the problem can be outlined as follows:

| | |
|---|---|
| **forever do** | **forever do** |
| *non-critical section* | *non-critical section* |
| *pre-protocol* | *pre-protocol* |
| **critical section** | **critical section** |
| *post-protocol* | *post-protocol* |

The non-critical section models the possibility that a process may do something else. It can take any amount of time (even infinite).

Our task is to find a pre- and post-protocol such that certain atomicity properties are satisfied.

# Desiderata for Critical Sections

We want to ensure two main properties for critical sections:

- **Mutual Exclusion** No two processes are in their critical section at the same time.
- **Eventual Entry** (or *starvation-freedom*) Once it enters its pre-protocol, a process will eventually be able to execute its critical section.

**Question**

Which is safety and which is liveness?

# Desiderata for Critical Sections

We want to ensure two main properties for critical sections:

- **Mutual Exclusion** No two processes are in their critical section at the same time.
- **Eventual Entry** (or *starvation-freedom*) Once it enters its pre-protocol, a process will eventually be able to execute its critical section.

**Question**

Which is safety and which is liveness?
Mutex is safety, Eventual Entry is liveness.

# Locks

The most common abstraction to ensure mutual exclusion of entry to critical sections is *locks*. Typically a lock is abstracted into an abstract data type, with two operations:

- *Taking* the lock — the first exchange (step $p_2/q_2$)
- *Releasing* the lock — the second exchange (step $p_4/q_4$)

| **var** *lock* | |
|---|---|
| **forever do** | **forever do** |
| $p_1$ *non-critical section* | $q_1$ *non-critical section* |
| $p_2$ **take** (*lock*) | $q_2$ **take** (*lock*); |
| $p_3$ **critical section** | $q_3$ **critical section** |
| $p_4$ **release** (*lock*) | $q_4$ **release** (*lock*); |

# Locks – Implementation Concerns (C)

C11 specifies extensions for mutex locking primitives.

- These should be implemented with the help of architecture-dependent hardware support, at assembly level.
- Don't try to home-roll mutex implementations by trying to implement them with racy reads/writes in C itself!
  - Declaring variables `volatile` doesn't help you!
  - `volatile` just means the compiler won't optimise reads/writes away, but racy reads/writes to `volatile` variables are still undefined behaviour!

# Locks – Implementation Concerns (LCR)

When reasoning about the design of a synchronisation primitive, we typically require that each statement only accesses (reads from or writes to) at most one shared variable at a time. Otherwise, we cannot guarantee that each statement is one atomic step.

This is called the *limited critical reference* restriction.

For example, for shared variable $x$ and non-shared variable $p$:

- $x := x + 1$ *does not* satisfy the LCR restriction.
- $p := x; x := p + 1$ does satisfy it.
  (But it wouldn't if $p$ was also shared.)

# Locks – Implementation Concerns (primitives)

Then, to implement a suitable pre- and post-protocol to ensure mutual exclusion and eventual entry, read the documentation on atomic primitives for your target hardware.[1]

| | |
|---|---|
| **forever do** | **forever do** |
| *non-critical section* | *non-critical section* |
| *pre-protocol* | *pre-protocol* |
| **critical section** | **critical section** |
| *post-protocol* | *post-protocol* |

[1] And take COMP3151/9154 Foundations of Concurrency when it starts being offered again.

# Locks – Implementation Concerns (primitives)

Then, to implement a suitable pre- and post-protocol to ensure mutual exclusion and eventual entry, read the documentation on atomic primitives for your target hardware.[1]

| | |
|---|---|
| **forever do**<br>    *non-critical section*<br>    *pre-protocol*<br>    **critical section**<br>    *post-protocol* | **forever do**<br>    *non-critical section*<br>    *pre-protocol*<br>    **critical section**<br>    *post-protocol* |

Useful atomic primitives may include: `CAS` (compare-and-swap), `FAA` (fetch-and-add), `XCHG` (exchange register/memory with register), test-and-set instructions, etc.

[1] And take COMP3151/9154 Foundations of Concurrency when it starts being offered again.

# Locks and Deadlock

But even with locks, you have to be careful because taking multiple locks in a certain order can cause a concurrent program to get stuck – in concurrent contexts, often called *deadlock*.

Example:

| **var** $A, B$ | | | |
|---|---|---|---|
| **forever do** | | **forever do** | |
| $p_1$ | *non-critical section* | $q_1$ | *non-critical section* |
| $p_2$ | **take** $(A)$; | $q_2$ | **take** $(B)$; |
| $p_3$ | **take** $(B)$; | $q_3$ | **take** $(A)$; |
| $p_4$ | **critical section** | $q_4$ | **critical section** |
| $p_5$ | **release** $(B)$; | $q_5$ | **release** $(A)$; |
| $p_6$ | **release** $(A)$; | $q_6$ | **release** $(B)$; |

# Message Passing and Deadlock

In message passing concurrency, ensuring atomicity is less of an issue because each thread chooses when it will interact with others – communication is via send/receive APIs rather than shared state.

# Message Passing and Deadlock

In message passing concurrency, ensuring atomicity is less of an issue because each thread chooses when it will interact with others – communication is via send/receive APIs rather than shared state.

But getting stuck is still a concern.

# Message Passing and Deadlock

In message passing concurrency, ensuring atomicity is less of an issue because each thread chooses when it will interact with others – communication is via send/receive APIs rather than shared state.

But getting stuck is still a concern.

**Example**

P waits to receive Q's message; Q waits to receive P's message

# Message Passing and Deadlock

In message passing concurrency, ensuring atomicity is less of an issue because each thread chooses when it will interact with others – communication is via send/receive APIs rather than shared state.

But getting stuck is still a concern.

**Example**

P waits to receive Q's message; Q waits to receive P's message
Deadlock!

# Session Types

A type system for processes that establish and communicate along channels.

Dual types reflect behaviour on either side of the channel.

- $\oplus$ "plus" (w/ unit 0) vs $\&$ "with" (w/ unit $\top$)
  – selecting vs offering a choice
- $\otimes$ "times" (w/ unit 1) vs $⅋$ "par" (w/ unit $\bot$)
  – outputting vs inputting a process, then continuing
- $\exists$ (existential) vs $\forall$ (universal)
  – sending vs receiving a type
- ? (client request) vs ! (server accept)
  – requesting vs offering a process repeatedly

# Session Types

A type system for processes that establish and communicate along channels.

Dual types reflect behaviour on either side of the channel.

- $\oplus$ "plus" (w/ unit 0) vs $\&$ "with" (w/ unit $\top$)
  – selecting vs offering a choice
- $\otimes$ "times" (w/ unit 1) vs $\parr$ "par" (w/ unit $\bot$)
  – outputting vs inputting a process, then continuing
- $\exists$ (existential) vs $\forall$ (universal)
  – sending vs receiving a type
- ? (client request) vs ! (server accept)
  – requesting vs offering a process repeatedly

Correspondence to linear logic ensures freedom from deadlock.
(Threads can't get stuck!)

# Session Types

A type system for processes that establish and communicate along channels.

Dual types reflect behaviour on either side of the channel.

- $\oplus$ "plus" (w/ unit 0) vs $\&$ "with" (w/ unit $\top$)
  – selecting vs offering a choice
- $\otimes$ "times" (w/ unit 1) vs $⅋$ "par" (w/ unit $\bot$)
  – outputting vs inputting a process, then continuing
- $\exists$ (existential) vs $\forall$ (universal)
  – sending vs receiving a type
- ? (client request) vs ! (server accept)
  – requesting vs offering a process repeatedly

Correspondence to linear logic ensures freedom from deadlock.
(Threads can't get stuck!)

# Duals and Units

Dual types reflect behaviour on either side of the channel.
We'll use notation $A^\perp$ for the dual of type $A$.

First we have that $(A)^\perp = A^\perp$ and $(A^\perp)^\perp = A$.

# Duals and Units

Dual types reflect behaviour on either side of the channel.
We'll use notation $A^\perp$ for the dual of type $A$.

First we have that $(A)^\perp = A^\perp$ and $(A^\perp)^\perp = A$. Then:

- $(A \oplus B)^\perp = A^\perp \,\&\, B^\perp$
  – select (vs offer) a choice between $A$ and $B$
- $(A \otimes B)^\perp = A^\perp \parr B^\perp$
  – output (vs input) a process of type $A$, then continue as $B$

# Duals and Units

Dual types reflect behaviour on either side of the channel.
We'll use notation $A^\perp$ for the dual of type $A$.

First we have that $(A)^\perp = A^\perp$ and $(A^\perp)^\perp = A$. Then:

- $(A \oplus B)^\perp = A^\perp \mathbin{\&} B^\perp$
  – select (vs offer) a choice between $A$ and $B$
- $(A \otimes B)^\perp = A^\perp \mathbin{⅋} B^\perp$
  – output (vs input) a process of type $A$, then continue as $B$

These four type operators have their unit values, also related in two pairs by dualities:

- $A \oplus 0 = A$ and $A \mathbin{\&} \top = A$ and $0^\perp = \top$ and $\top^\perp = 0$
- $A \otimes 1 = A$ and $A \mathbin{⅋} \bot = A$ and $1^\perp = \bot$ and $\bot^\perp = 1$
  (unit $\bot$ not to be confused with dual notation $^\perp$)

# Typing and Reduction of Units

For $\oplus$ and $\&$, respectively, as (impossible) empty selection and (trivial) empty choice:

(no rule for 0) $$\frac{}{x.case() \vdash \Gamma, x : \top}\ \top$$

For $\otimes$ and $\invamp$, respectively, as empty output and empty input:

$$\frac{}{x[].0 \vdash x : 1}\ 1 \qquad \frac{P \vdash \Gamma}{x().P \vdash \Gamma, x : \bot}\ \bot$$

# Typing and Reduction of Units

For $\oplus$ and $\&$, respectively, as (impossible) empty selection and (trivial) empty choice:

(no rule for 0) $$\frac{}{x.case() \vdash \Gamma, x : \top}\ \top$$

For $\otimes$ and $⅋$, respectively, as empty output and empty input:

$$\frac{}{x[].0 \vdash x : 1}\ 1 \qquad \frac{P \vdash \Gamma}{x().P \vdash \Gamma, x : \bot}\ \bot$$

Note: We'll use $\Gamma, \Delta, \Theta$ for typing environments; ordering of entries in session typing environments is ignored.

# Typing and Reduction of Units

For $\oplus$ and $\&$, respectively, as (impossible) empty selection and (trivial) empty choice:

$$\text{(no rule for 0)} \qquad \frac{}{x.case() \vdash \Gamma, x : \top}\ \top$$

For $\otimes$ and $⅋$, respectively, as empty output and empty input:

$$\frac{}{x[].0 \vdash x : 1}\ 1 \qquad \frac{P \vdash \Gamma}{x().P \vdash \Gamma, x : \bot}\ \bot$$

Note: We'll use $\Gamma, \Delta, \Theta$ for typing environments; ordering of entries in session typing environments is ignored.

There is only a reduction rule for 1 with $\bot$ (none for 0 with $\top$):

- **ch** $x.(x[].0 \mid x().P) \implies P$ ($\beta_{1\bot}$)

# Selection and Choice

To type a process $P$ that first transmits a request along channel $x$ to select from type $A \oplus B$, we invoke one of two rules:

$$\frac{P \vdash \Gamma, x : A}{x[\mathsf{inl}].P \vdash \Gamma, x : A \oplus B}\ \oplus_1 \qquad \frac{P \vdash \Gamma, x : B}{x[\mathsf{inr}].P \vdash \Gamma, x : A \oplus B}\ \oplus_2$$

# Selection and Choice

To type a process $P$ that first transmits a request along channel $x$ to select from type $A \oplus B$, we invoke one of two rules:

$$\frac{P \vdash \Gamma, x : A}{x[\mathsf{inl}].P \vdash \Gamma, x : A \oplus B}\ \oplus_1 \qquad \frac{P \vdash \Gamma, x : B}{x[\mathsf{inr}].P \vdash \Gamma, x : A \oplus B}\ \oplus_2$$

The process offering the choice on channel $x$ can then branch between $Q$ or $R$ based on whether inl or inr was chosen:

$$\frac{Q \vdash \Delta, x : A^{\perp} \qquad R \vdash \Delta, x : B^{\perp}}{x.\mathsf{case}(Q, R) \vdash \Delta, x : A^{\perp} \mathbin{\&} B^{\perp}}\ \&$$

# Selection and Choice

To type a process $P$ that first transmits a request along channel $x$ to select from type $A \oplus B$, we invoke one of two rules:

$$\frac{P \vdash \Gamma, x : A}{x[\mathsf{inl}].P \vdash \Gamma, x : A \oplus B}\ \oplus_1 \qquad \frac{P \vdash \Gamma, x : B}{x[\mathsf{inr}].P \vdash \Gamma, x : A \oplus B}\ \oplus_2$$

The process offering the choice on channel $x$ can then branch between $Q$ or $R$ based on whether inl or inr was chosen:

$$\frac{Q \vdash \Delta, x : A^\perp \qquad R \vdash \Delta, x : B^\perp}{x.\mathsf{case}(Q, R) \vdash \Delta, x : A^\perp \,\&\, B^\perp}\ \&$$

Reduction rules:

- $\mathbf{ch}\ x.(x[\mathsf{inl}].P \mid x.\mathsf{case}(Q,R)) \Longrightarrow \mathbf{ch}\ x.(P \mid Q)$ $(\beta_{\oplus\&1})$
- $\mathbf{ch}\ x.(x[\mathsf{inr}].P \mid x.\mathsf{case}(Q,R)) \Longrightarrow \mathbf{ch}\ x.(P \mid R)$ $(\beta_{\oplus\&2})$

# Output and Input

To type a process that (1) outputs a request along channel $x$ to open a new channel $y$ for process $P : A$, then (2) continues to behave as process $Q : B$, we invoke this rule:

$$\frac{P \vdash \Gamma, y : A \qquad Q \vdash \Delta, x : B}{x[y].(P \mid Q) \vdash \Gamma, \Delta, x : A \otimes B} \otimes$$

# Output and Input

To type a process that (1) outputs a request along channel $x$ to open a new channel $y$ for process $P : A$, then (2) continues to behave as process $Q : B$, we invoke this rule:

$$\frac{P \vdash \Gamma, y : A \qquad Q \vdash \Delta, x : B}{x[y].(P \mid Q) \vdash \Gamma, \Delta, x : A \otimes B} \otimes$$

The process that receives input channel name $y$ along $x$ then executes $R$, which is allowed to communicate on both channels:

$$\frac{R \vdash \Theta, y : A^{\perp}, x : B^{\perp}}{x(y).R \vdash \Theta, x : A^{\perp} \parr B^{\perp}} \parr$$

# Output and Input

To type a process that (1) outputs a request along channel $x$ to open a new channel $y$ for process $P : A$, then (2) continues to behave as process $Q : B$, we invoke this rule:

$$\frac{P \vdash \Gamma, y : A \qquad Q \vdash \Delta, x : B}{x[y].(P \mid Q) \vdash \Gamma, \Delta, x : A \otimes B} \otimes$$

The process that receives input channel name $y$ along $x$ then executes $R$, which is allowed to communicate on both channels:

$$\frac{R \vdash \Theta, y : A^{\perp}, x : B^{\perp}}{x(y).R \vdash \Theta, x : A^{\perp} \mathbin{⅋} B^{\perp}} ⅋$$

Reduction rule:

- $\textbf{ch}\ x.(x[y].(P \mid Q) \mid x(y).R) \Longrightarrow \textbf{ch}\ y.(P \mid \textbf{ch}\ x.(Q \mid R))\ (\beta_{\otimes ⅋})$

# Parallel Composition as Cut

To type the parallel composition of two processes $P$ and $Q$ communicating along channel $x$, we require their types are dual.

$$\frac{P \vdash \Gamma, x : A \qquad Q \vdash \Delta, x : A^{\perp}}{\mathbf{ch}\ x : A.\ (P \mid Q) \vdash \Gamma, \Delta}\ \text{Cut}$$

This rule is so named because it corresponds to the cut rule in linear logic. (The blue parts are classical linear logic propositions.)

# Parallel Composition as Cut

To type the parallel composition of two processes $P$ and $Q$ communicating along channel $x$, we require their types are dual.

$$\dfrac{P \vdash \Gamma, x : A \qquad Q \vdash \Delta, x : A^{\perp}}{\mathbf{ch}\ x : A.\ (P \mid Q) \vdash \Gamma, \Delta}\ \text{Cut}$$

This rule is so named because it corresponds to the cut rule in linear logic. (The blue parts are classical linear logic propositions.)

In general, cut rules in such logics are a way of composing proofs.

# Process Reduction as Cut Elimination

The dynamic semantics of session-typed processes then corresponds to cut elimination: the simplification of linear logic proofs so they don't use the Cut rule as their final step.

# Process Reduction as Cut Elimination

The dynamic semantics of session-typed processes then corresponds to cut elimination: the simplification of linear logic proofs so they don't use the Cut rule as their final step.

Here are some resulting equivalences and simplifications:

- $\mathbf{ch}\ x : A.\ (P \mid Q) \vdash \Gamma, \Delta \equiv \mathbf{ch}\ x : A^{\perp}.\ (Q \mid P) \vdash \Gamma, \Delta$ (Swap)

# Process Reduction as Cut Elimination

The dynamic semantics of session-typed processes then corresponds to cut elimination: the simplification of linear logic proofs so they don't use the Cut rule as their final step.

Here are some resulting equivalences and simplifications:

- $\textbf{ch}\ x : A.\ (P \mid Q) \vdash \Gamma, \Delta \equiv \textbf{ch}\ x : A^{\perp}.\ (Q \mid P) \vdash \Gamma, \Delta$ (Swap)
- $\textbf{ch}\ y.(\textbf{ch}\ x.(P \mid Q) \mid R) \vdash \Gamma, \Delta, \Theta \equiv$
  $\textbf{ch}\ x.(P \mid \textbf{ch}\ y.(Q \mid R)) \vdash \Gamma, \Delta, \Theta$ (Assoc)

# Process Reduction as Cut Elimination

The dynamic semantics of session-typed processes then corresponds to cut elimination: the simplification of linear logic proofs so they don't use the Cut rule as their final step.

Here are some resulting equivalences and simplifications:

- $\textbf{ch}\ x : A.\ (P \mid Q) \vdash \Gamma, \Delta \equiv \textbf{ch}\ x : A^{\perp}.\ (Q \mid P) \vdash \Gamma, \Delta$ (Swap)
- $\textbf{ch}\ y.(\textbf{ch}\ x.(P \mid Q) \mid R) \vdash \Gamma, \Delta, \Theta \equiv$
  $\textbf{ch}\ x.(P \mid \textbf{ch}\ y.(Q \mid R)) \vdash \Gamma, \Delta, \Theta$ (Assoc)

(Omitting types now, as all this leaves them unchanged:)

- $\textbf{ch}\ x.(x[\mathsf{inl}].P \mid x.\mathsf{case}(Q, R)) \Longrightarrow \textbf{ch}\ x.(P \mid Q)$ $(\beta_{\oplus\&1})$
- $\textbf{ch}\ x.(x[\mathsf{inr}].P \mid x.\mathsf{case}(Q, R)) \Longrightarrow \textbf{ch}\ x.(P \mid R)$ $(\beta_{\oplus\&2})$
- $\textbf{ch}\ x.(x[y].(P \mid Q) \mid x(y).R) \Longrightarrow \textbf{ch}\ y.(P \mid \textbf{ch}\ x.(Q \mid R))$ $(\beta_{\otimes\parr})$

# Dynamic Semantics Summary

For our chosen subset, we have equivalences:

- $\textbf{ch}\ x : A.\ (P \mid Q) \equiv \textbf{ch}\ x : A^{\perp}.\ (Q \mid P)$ (Swap)
- $\textbf{ch}\ y.(\textbf{ch}\ x.(P \mid Q) \mid R) \equiv \textbf{ch}\ x.(P \mid \textbf{ch}\ y.(Q \mid R))$ (Assoc)

In addition, we have the reduction rules for:

- Selection and Choice
  - $\textbf{ch}\ x.(x[\text{inl}].P \mid x.\text{case}(Q, R)) \Longrightarrow \textbf{ch}\ x.(P \mid Q)$ $(\beta_{\oplus \& 1})$
  - $\textbf{ch}\ x.(x[\text{inr}].P \mid x.\text{case}(Q, R)) \Longrightarrow \textbf{ch}\ x.(P \mid R)$ $(\beta_{\oplus \& 2})$
- Output and Input
  - $\textbf{ch}\ x.(x[y].(P \mid Q) \mid x(y).R) \Longrightarrow \textbf{ch}\ y.(P \mid \textbf{ch}\ x.(Q \mid R))$ $(\beta_{\otimes ⅋})$
- Empty Output and Input
  - $\textbf{ch}\ x.(x[].0 \mid x().P) \Longrightarrow P$ $(\beta_{1\perp})$

We now have enough to inspect some examples. (**Demo**)

# Deadlock Freedom of Session-Typed Processes

Let's look at the Cut rule again:

$$\frac{P \vdash \Gamma, x : A \qquad Q \vdash \Delta, x : A^{\perp}}{\mathbf{ch}\ x : A.\ (P \mid Q) \vdash \Gamma, \Delta}\ \text{Cut}$$

## Question

Would $\mathbf{ch}\ x, y.\ (x(u).\texttt{wait\_meal} \mid y(v).\texttt{wait\_payment})$ be typeable using session types?

# Deadlock Freedom of Session-Typed Processes

Let's look at the Cut rule again:

$$\frac{P \vdash \Gamma, x : A \qquad Q \vdash \Delta, x : A^{\perp}}{\mathbf{ch}\ x : A.\ (P \mid Q) \vdash \Gamma, \Delta}\ \text{Cut}$$

**Question**

Would $\mathbf{ch}\ x, y.\ (x(u).\mathtt{wait\_meal} \mid y(v).\mathtt{wait\_payment})$ be typeable using session types?

No.

Cut is the only rule that types parallel composition, and it only permits processes $P$ and $Q$ to have *one* channel $x$ between them.

This prevents any loops of communication like in the example above, that can lead to deadlock.

# A Typing Rule that Allows Deadlock

Conversely, suppose we were to extend session types with a "BiCut" rule that permits two channels between $P$ and $Q$.

$$\frac{P \vdash \Gamma, x : A, y : B \qquad Q \vdash \Delta, x : A^{\perp}\ y : B^{\perp}}{\textbf{ch}\ x : A, y : B.\ (P \mid Q) \vdash \Gamma, \Delta}\ \text{BiCut}$$

Then $\textbf{ch}\ x, y.\ (x(u).\texttt{wait\_meal} \mid y(v).\texttt{wait\_payment})$ would be typeable using this rule, but the processes would immediately get stuck waiting for each other.

# Concurrency Summary

- Shared-variable concurrency is most often synchronised to avoid data races to critical sections, using locks.
  - Providing locks requires careful implementation with:
    - operations that obey limited critical reference restrictions.
    - thorough knowledge of atomic hardware primitives.
  - Using locks carelessly is prone to deadlock.
- Message-passing concurrency limits communication to explicit APIs between processes, but is still prone to deadlock.
- Session types for message-passing processes have a correspondence to classical linear logic.
  - They have dual types to support, between processes communicating over a channel:
    - selection/choice and output/input (covered today)
    - type polymorphism and service repetition (not covered today)
  - Processes prone to deadlock-causing communication loops via multiple channels between them fail to typecheck.

# Acknowledgements for Session Types

Today's presentation was based on "Propositions as Sessions" by Philip Wadler (2012, 2014), but session types go back to the 90s (Honda, 1993), and linear logic back to the 80s (Girard, 1987).

Wadler continues a line of work by the above authors as well as

- Abramsky (1994), Bellin and Scott (1994), Caires and Pfenning (2010) and others on the correspondence between classical linear logic and session types.
- Honda, Kubo and Vasconcelos (1998), Gay and Vasconcelos (2010) and others on a functional language with session types.

For more information including references to the others, see Wadler's conference paper and journal article of the same name.

# MyExperience

Please fill out the survey. It helps tremendously.

`https://myexperience.unsw.edu.au`