Asynchronous Session-based Concurrency: Deadlock Freedom by Typing

Jorge A. Pérez
www.jperez.nl

University of Groningen, The Netherlands

Dutch Winter School 2026
(Part 3)

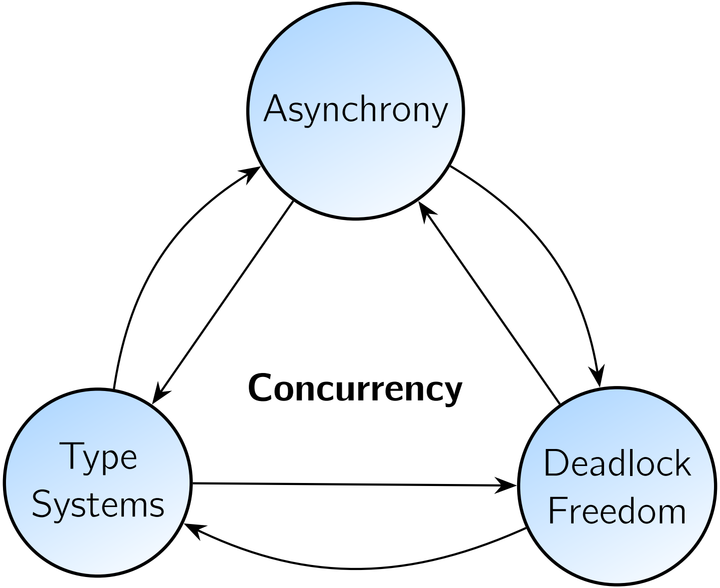
Taking Stock

Up to here:

- Concurrent interpretation of LL: statics and dynamics
- Left and right rules per connective - rely and guarantee interactive behaviors
- Cut reductions and process synchronizations
- A look at the correctness properties ensured by the logic-based type system
- The computational interpretation of proof transformations
- Deadlock freedom (DF) and progress

Today:

- Asynchronous processes, based on classical linear logic, and DF


Asynchrony
Concurrency
Type Systems
Deadlock Freedom

The Deadlock Freedom Property (DF)

- Informally, the guarantee that processes "never get stuck".
- Most desirable for the message-passing programs that enable today's concurrent and distributed software systems.
- Even a single component that becomes permanently blocked waiting for a message can impact system reliability.
- Difficulty: Non-local component analysis.

Example

```
peter c1 c2 =
  let (x, c1) = receive c1 in
  let c2 = send (41) c2 in
  wait c1;
  close c2;
  ()
```

```
miles c1 c2 =
  let (x, c2) = receive c2 in
  let c1 = send (42) c1 in
  wait c2;
  close c1;
  ()
```

Example

```
peter c1 c2 =
  let (x, c1) = receive c1 in
  let c2 = send (41) c2 in
  wait c1;
  close c2;
  ()
```

This program follows its protocol ✔

```
miles c1 c2 =
  let (x, c2) = receive c2 in
  let c1 = send (42) c1 in
  wait c2;
  close c1;
  ()
```

Example

```
peter c1 c2 =
  let (x, c1) = receive c1 in
  let c2 = send (41) c2 in
  wait c1;
  close c2;
  ()
```

This program follows its protocol ✔

```
miles c1 c2 =
  let (x, c2) = receive c2 in
  let c1 = send (42) c1 in
  wait c2;
  close c1;
  ()
```

This program follows its protocol ✔

Example, with Concurrency

```
peter c1 c2 =
  let (x, c1) = receive c1 in
  let c2 = send (41) c2 in
  wait c1;
  close c2;
  ()
```

```
miles c1 c2 =
  let (x, c2) = receive c2 in
  let c1 = send (42) c1 in
  wait c2;
  close c1;
  ()
```

```
main =
  let (c1, c1dual) = new @T () in
  let (c2, c2dual) = new @T () in
  fork (miles c1dual c2);
       peter c1 c2dual
```

Is ‘main’ deadlock-free?

Using Processes to Enforce DF

Program (Rust, Go, ABS...)

Processes (π-calculus)

Using Processes to Enforce DF

Program (Rust, Go, ABS...)

Processes (π-calculus)

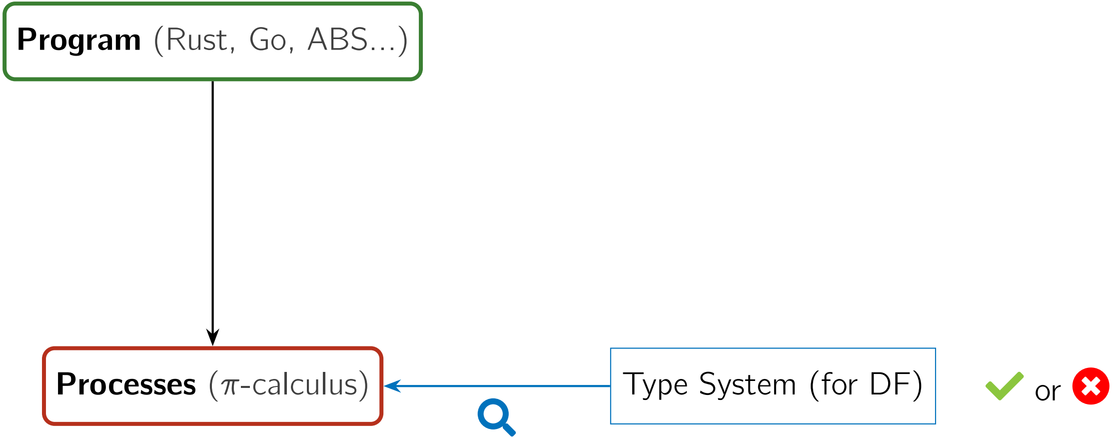
Type System (for DF)

Using Processes to Enforce DF

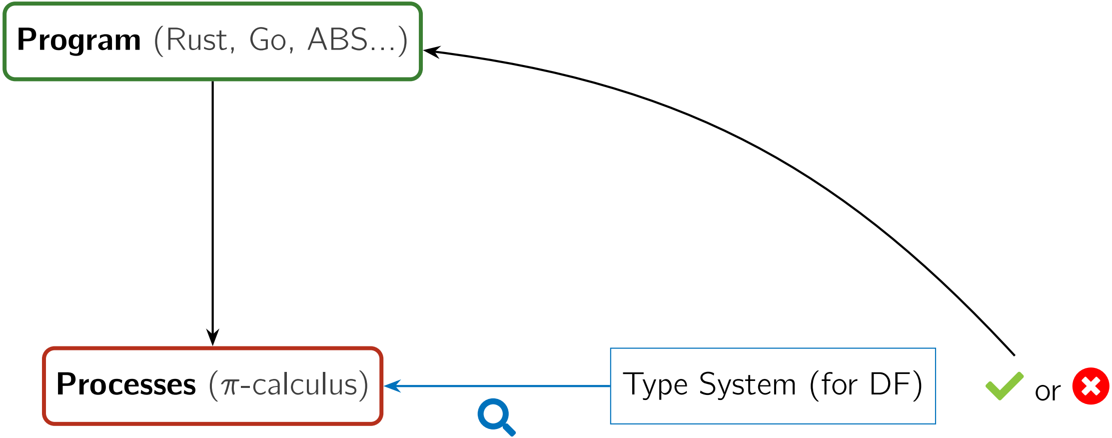
Program (Rust, Go, ABS...)

Processes (π-calculus) ← Type System (for DF) ✓ or ✗

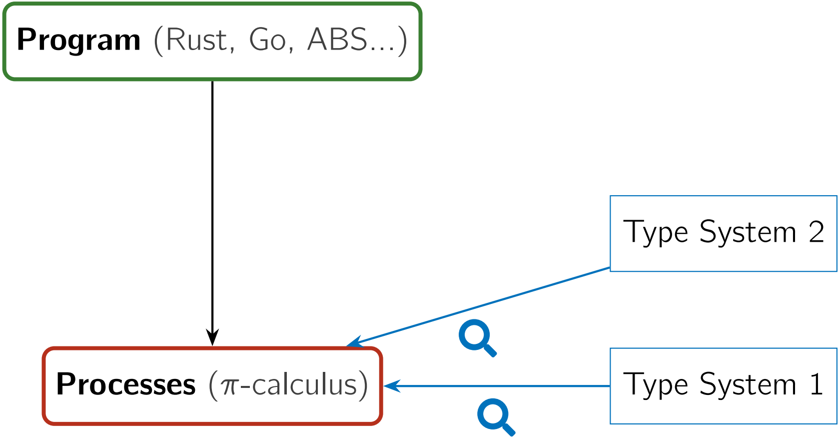
Using Processes to Enforce DF



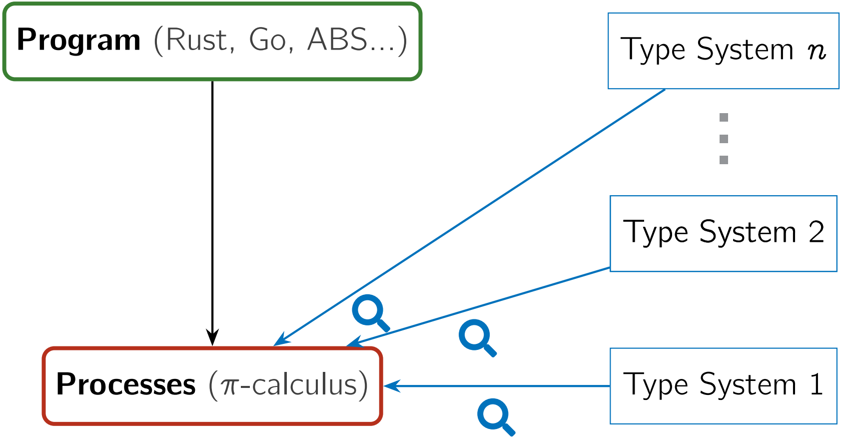
Using Processes to Enforce DF



Using Processes to Enforce DF: Many Different Type Systems!



Using Processes to Enforce DF: Many Different Type Systems!



Challenges	
Contrast type systems that enforce DF	**Design** more advanced type systems for DF

Challenges		
Contrast type systems that enforce DF	⟷	**Design** more advanced type systems for DF

Challenges		
Contrast type systems that enforce DF	$\longleftrightarrow$	**Design** more advanced type systems for DF

Our Insight: Session type systems derived from Linear Logic ('Propositions as Sessions') offer a guiding reference for both!

Some Observations

- Asynchronous communication is the cleanest setting to study DF by typing

Some Observations

- Asynchronous communication is the cleanest setting to study DF by typing
- Asynchrony unblocks behaviors / session types constrain behaviors

Some Observations

- Asynchronous communication is the cleanest setting to study DF by typing
- Asynchrony unblocks behaviors / session types constrain behaviors
- Asynchronous sessions “in between” untyped synchrony and asynchrony:

Some Observations

- Asynchronous communication is the cleanest setting to study DF by typing
- Asynchrony unblocks behaviors / session types constrain behaviors
- Asynchronous sessions "in between" untyped synchrony and asynchrony: (output) actions from different sessions are independent, but (output) actions within a session should respect their session type.

Some Observations

- Asynchronous communication is the cleanest setting to study DF by typing
- Asynchrony unblocks behaviors / session types constrain behaviors
- Asynchronous sessions “in between” untyped synchrony and asynchrony: (output) actions from different sessions are independent, but (output) actions within a session should respect their session type.

Today We overview DF enforcement in three typed asynchronous π-calculi:

Some Observations

- Asynchronous communication is the cleanest setting to study DF by typing
- Asynchrony unblocks behaviors / session types constrain behaviors
- Asynchronous sessions “in between” untyped synchrony and asynchrony: (output) actions from different sessions are independent, but (output) actions within a session should respect their session type.

Today We overview DF enforcement in three typed asynchronous π-calculi:

1. AP: processes follow protocols but ‘happily deadlock’ (**no DF enforcement**)

Some Observations

- Asynchronous communication is the cleanest setting to study DF by typing
- Asynchrony unblocks behaviors / session types constrain behaviors
- Asynchronous sessions "in between" untyped synchrony and asynchrony: (output) actions from different sessions are independent, but (output) actions within a session should respect their session type.

Today We overview DF enforcement in three typed asynchronous π-calculi:

1. AP: processes follow protocols but 'happily deadlock' (**no DF enforcement**)
2. ACP: DF by typing for tree-like process topologies (**basic enforcement**)

Some Observations

- Asynchronous communication is the cleanest setting to study DF by typing
- Asynchrony unblocks behaviors / session types constrain behaviors
- Asynchronous sessions "in between" untyped synchrony and asynchrony: (output) actions from different sessions are independent, but (output) actions within a session should respect their session type.

Today We overview DF enforcement in three typed asynchronous π-calculi:

1. AP: processes follow protocols but 'happily deadlock' (**no DF enforcement**)
2. ACP: DF by typing for tree-like process topologies (**basic enforcement**)
3. APCP: DF even for circular process topologies (**advanced enforcement**)

Some Observations

- Asynchronous communication is the cleanest setting to study DF by typing
- Asynchrony unblocks behaviors / session types constrain behaviors
- Asynchronous sessions "in between" untyped synchrony and asynchrony: (output) actions from different sessions are independent, but (output) actions within a session should respect their session type.

Today We overview DF enforcement in three typed asynchronous π-calculi:

1. AP: processes follow protocols but 'happily deadlock' (**no DF enforcement**)
2. ACP: DF by typing for tree-like process topologies (**basic enforcement**)
3. APCP: DF even for circular process topologies (**advanced enforcement**)

Full details: Our LMCS and ICE'24 papers.

Session Processes and Deadlocks

Process Syntax

$$
\begin{array}{rllcrlr}
P, Q ::= & x[a, b] & \text{send} & \mid & x(y, z); P & \text{receive} \\
\mid & x[b] \triangleleft j & \text{selection} & \mid & x(z) \triangleright \{i : P\}_{i \in I} & \text{branch} \\
\mid & P \mid Q & \text{parallel} & \mid & (\nu xy) P & \text{restriction} \\
\mid & 0 & \text{inaction} & \mid & [x \leftrightarrow y] & \text{forwarder}
\end{array}
$$

Process Syntax

$$
\begin{array}{rllcll}
P, Q ::= & x[a, b] & \text{send} & \mid & x(y, z); P & \text{receive} \\
\mid & x[b] \triangleleft j & \text{selection} & \mid & x(z) \triangleright \{i : P\}_{i \in I} & \text{branch} \\
\mid & P \mid Q & \text{parallel} & \mid & (\nu xy)P & \text{restriction} \\
\mid & 0 & \text{inaction} & \mid & [x \leftrightarrow y] & \text{forwarder}
\end{array}
$$

Conventions:

- We write $a, b, c, \ldots, x, y, z, \ldots$ to denote **names** (or **endpoints**)
- Early letters of the alphabet denote **objects** of output-like constructs.
- We write $\tilde{x}, \tilde{y}, \tilde{z}, \ldots$ to denote finite sequences of names.

Reduction

$$\frac{\text{[red-send-recv]}}{(\nu xy)(x[a, b] \mid y(a', b'); Q) \longrightarrow Q\{a/a', b/b'\}}$$

Reduction

[red-send-recv]

$$\frac{}{(\nu xy)(x[a,b] \mid y(a',b');Q) \longrightarrow Q\{a/a', b/b'\}}$$

[red-sel-bra]

$$\frac{j \in I}{(\nu xy)(x[b] \triangleleft j \mid y(b') \triangleright \{i : Q_i\}_{i \in I}) \longrightarrow Q_j\{b/b'\}}$$

Reduction

[red-send-recv]

$$\frac{}{(\nu xy)(x[a,b] \mid y(a',b');Q) \longrightarrow Q\{a/a',b/b'\}}$$

[red-sel-bra]

$$\frac{j \in I}{(\nu xy)(x[b] \triangleleft j \mid y(b') \triangleright \{i : Q_i\}_{i \in I}) \longrightarrow Q_j\{b/b'\}}$$

[red-fwd]

$$\frac{y \neq z}{(\nu xy)([x \leftrightarrow z] \mid P) \longrightarrow P\{z/y\}}$$

Reduction

[red-send-recv]

$$\frac{}{(\nu xy)(x[a,b] \mid y(a',b');Q) \longrightarrow Q\{a/a', b/b'\}}$$

[red-sel-bra]

$$\frac{j \in I}{(\nu xy)(x[b] \triangleleft j \mid y(b') \triangleright \{i : Q_i\}_{i\in I}) \longrightarrow Q_j\{b/b'\}}$$

[red-fwd]

$$\frac{y \neq z}{(\nu xy)([x \leftrightarrow z] \mid P) \longrightarrow P\{z/y\}}$$

[red-sc]

$$\frac{P \equiv P' \qquad P' \longrightarrow P' \qquad Q' \equiv Q}{P \longrightarrow Q}$$

[red-res]

$$\frac{P \longrightarrow Q}{(\nu xy)P \longrightarrow (\nu xy)Q}$$

[red-par]

$$\frac{P \longrightarrow Q}{P \mid R \longrightarrow Q \mid R}$$

Derived Constructs / Syntactic Sugar

$$
\begin{aligned}
\overline{x}[y] \cdot P &\triangleq (\nu ya)(\nu zb)(x[a, b] \mid P\{z/x\}) \\
x(y); P &\triangleq x(y, z); P\{z/x\} \\
\overline{x} \triangleleft \ell \cdot P &\triangleq (\nu zb)(x[b] \triangleleft \ell \mid P\{z/x\}) \\
x \triangleright \{i : P_i\}_{i \in I} &\triangleq x(z) \triangleright \{i : P_i\{z/x\}\}_{i \in I}
\end{aligned}
$$

Note: We use '·' to signify that output-like operations are non blocking.

Continuations Induce Protocols

$$
\begin{aligned}
P \triangleq (\nu zu)\Big((\nu xy)\Big(&(\nu a x')(x[v_1, a] \mid x'[v_2, b]) \\
&\mid (\nu c z')(z[v_3, c] \mid y(w_1, y'); y'(w_2, y''); Q)\Big) \\
&\mid u(w_3, u'); R\Big)
\end{aligned}
$$

Continuations Induce Protocols

$$
\begin{aligned}
P \triangleq (\nu zu)\Big((\nu xy)\Big(&(\nu a x')(x[v_1, a] \mid x'[v_2, b]) \\
&\mid (\nu c z')(z[v_3, c] \mid y(w_1, y'); y'(w_2, y''); Q)\Big) \\
&\mid u(w_3, u'); R\Big)
\end{aligned}
$$

Or, using the sugared syntax:

$$
P = (\nu zu)((\nu xy)(\overline{x}[v_1] \cdot \overline{x}[v_2] \cdot 0 \mid \overline{z}[v_3] \cdot y(w_1); y(w_2); Q') \mid u(w_3); R')
$$

where $Q' \triangleq Q\{y/y''\}$ and $R' \triangleq R\{u/u'\}$.

Continuations Induce Protocols

$$\begin{aligned} P \triangleq (\nu zu)\Big((\nu xy)\Big(&(\nu a x')(x[v_1, a] \mid x'[v_2, b]) \\ &\mid (\nu c z')(z[v_3, c] \mid y(w_1, y'); y'(w_2, y''); Q)\Big) \\ &\mid u(w_3, u'); R\Big) \end{aligned}$$

Or, using the sugared syntax:

$$P = (\nu zu)((\nu xy)(\overline{x}[v_1] \cdot \overline{x}[v_2] \cdot 0 \mid \overline{z}[v_3] \cdot y(w_1); y(w_2); Q') \mid u(w_3); R')$$

where $Q' \triangleq Q\{y/y''\}$ and $R' \triangleq R\{u/u'\}$.

We have:

$$P \longrightarrow (\nu zu)((\nu xy)(\overline{x}[v_2] \cdot 0 \mid \overline{z}[v_3] \cdot y(w_2); Q'\{v_1/w_1\}) \mid u(w_3); R')$$
$$P \longrightarrow (\nu xy)(\overline{x}[v_1] \cdot \overline{x}[v_2] \cdot 0 \mid y(w_1); y(w_2); Q') \mid R'\{v_3/w_3\}$$

Note: There is no reduction involving the send on x': because x' is connected to the continuation of the send on x, it is thus not (yet) paired with a dual receive.

A Deadlocked Process

The “hello world” of deadlocked message-passing processes:

$$(\nu xy)(\nu uw)(x(v, x'); u[a, b] \mid w(z, w'); y[c, d])$$

Process is stuck:

- The input on x is waiting for the right output on y...
- ...the output on y is blocked by a receive (on w) waiting for the left output on u, which is blocked by the input on x.

A Deadlocked Process

The "hello world" of deadlocked message-passing processes:

$$(\nu xy)(\nu uw)(x(v, x'); u[a, b] \mid w(z, w'); y[c, d])$$

Process is stuck:

- The input on x is waiting for the right output on y...
- ...the output on y is blocked by a receive (on w) waiting for the left output on u, which is blocked by the input on x.

In other words, there is a **cyclic dependency** between the two threads.

Session Types

Types correspond to propositions in Linear Logic:

$$
\begin{array}{rlrlr}
A, B ::= & A \otimes B & \text{send} & \mid\ A \parr B & \text{receive} \\
\mid & \oplus\{i : A\}_{i\in I} & \text{selection} & \mid\ \&\{i : A\}_{i\in I} & \text{branching} \\
\mid & \bullet & \text{closed protocol} & &
\end{array}
$$

Session Types

Types correspond to propositions in Linear Logic:

$$
\begin{array}{rlrlr}
A, B ::= & A \otimes B & \text{send} & \mid \; A \mathbin{⅋} B & \text{receive} \\
\mid & \oplus\{i : A\}_{i \in I} & \text{selection} & \mid \; \&\{i : A\}_{i \in I} & \text{branching} \\
\mid & \bullet & \text{closed protocol} & &
\end{array}
$$

The **dual** of session type A, denoted $\overline{A}$:

$$
\begin{array}{lll}
\overline{A \otimes B} \triangleq \overline{A} \mathbin{⅋} \overline{B} & \overline{\oplus\{i : A_i\}_{i \in I}} \triangleq \&\{i : \overline{A_i}\}_{i \in I} & \overline{\bullet} \triangleq \bullet \\
\overline{A \mathbin{⅋} B} \triangleq \overline{A} \otimes \overline{B} & \overline{\&\{i : A_i\}_{i \in I}} \triangleq \oplus\{i : \overline{A_i}\}_{i \in I} &
\end{array}
$$

Typing Judgments

Judgments are of the form

$$P \vdash \Gamma$$

where

- P is a process
- Γ records endpoint/protocol assignments of the form $x : A$.

Typing Judgments

Judgments are of the form

$$P \vdash \Gamma$$

where

- P is a process
- Γ records endpoint/protocol assignments of the form $x : A$.

Γ disallows

- *weakening* (all assignments must be used, except names typed with $\bullet$)
- *contraction* (assignments may not be duplicated).

but obeys *exchange* (assignments may be silently reordered).

The empty context is written $\emptyset$.

AP: Typing Rules

[typ-send]

$$\frac{}{x[a, b] \vdash x : A \otimes B, a : \overline{A}, b : \overline{B}}$$

[typ-recv]

$$\frac{P \vdash \Gamma, y : A, z : B}{x(y, z); P \vdash \Gamma, x : A \parr B}$$

AP: Typing Rules

[typ-send]

$$\frac{}{x[a, b] \vdash x : A \otimes B, a : \overline{A}, b : \overline{B}}$$

[typ-recv]

$$\frac{P \vdash \Gamma, y : A, z : B}{x(y, z); P \vdash \Gamma, x : A \parr B}$$

[typ-par]

$$\frac{P \vdash \Gamma \qquad Q \vdash \Delta}{P \mid Q \vdash \Gamma, \Delta}$$

[typ-res]

$$\frac{P \vdash \Gamma, x : A, y : \overline{A}}{(\nu xy) P \vdash \Gamma}$$

AP: Typing Rules

[typ-send]

$$\frac{}{x[a, b] \vdash x : A \otimes B, a : \overline{A}, b : \overline{B}}$$

[typ-recv]

$$\frac{P \vdash \Gamma, y : A, z : B}{x(y, z); P \vdash \Gamma, x : A \parr B}$$

[typ-par]

$$\frac{P \vdash \Gamma \qquad Q \vdash \Delta}{P \mid Q \vdash \Gamma, \Delta}$$

[typ-res]

$$\frac{P \vdash \Gamma, x : A, y : \overline{A}}{(\nu xy)P \vdash \Gamma}$$

[typ-sel]

$$\frac{j \in I}{x[b] \triangleleft j \vdash x : \oplus\{i : A_i\}_{i \in I}, b : \overline{A_j}}$$

[typ-bra]

$$\frac{\forall i \in I : P_i \vdash \Gamma, z : A_i}{x(z) \triangleright \{i : P_i\}_{i \in I} \vdash \Gamma, x : \&\{i : A_i\}_{i \in I}}$$

[typ-fwd]

$$\frac{}{[x \leftrightarrow y] \vdash x : \overline{A}, y : A}$$

[typ-end]

$$\frac{P \vdash \Gamma}{P \vdash \Gamma, x : \bullet}$$

[typ-inact]

$$\frac{}{0 \vdash \varnothing}$$

Properties of AP

Well-typed processes respect their session protocols under reduction:

Theorem (Type Preservation for AP)

Given $P \vdash \Gamma$ *and* Q *such that* $P \equiv Q$ *or* $P \longrightarrow Q$*, we have* $Q \vdash \Gamma$*.*

Deadlocks are Typable in AP

Recall the process

$$(\nu xy)(\nu uw)(x(v, x'); u[a, b] \mid w(z, w'); y[c, d])$$

It can be typed as follows:

$$
\dfrac{
  \dfrac{
    \dfrac{
      \dfrac{}{u[a, b] \vdash u : \bullet \otimes \bullet, a : \bullet, b : \bullet}
    }{
      \begin{array}{l} u[a, b] \vdash u : \bullet \otimes \bullet, \\ \qquad v : \bullet, x' : \bullet, a : \bullet, b : \bullet \end{array}
    } [\text{typ-end}]^2
  }{
    \begin{array}{l} x(v, x'); u[a, b] \vdash x : \bullet \parr \bullet, \\ \qquad u : \bullet \otimes \bullet, \\ \qquad a : \bullet, b : \bullet \end{array}
  } [\text{typ-recv}]
  \qquad
  \dfrac{\vdots}{
    \begin{array}{l} w(z, w'); y[c, d] \vdash w : \bullet \parr \bullet, \\ \qquad y : \bullet \otimes \bullet, \\ \qquad c : \bullet, d : \bullet \end{array}
  }
}{
  \dfrac{
    \begin{array}{l} x(v, x'); u[a, b] \mid w(z, w'); y[c, d] \vdash x : \bullet \parr \bullet, u : \bullet \otimes \bullet, a : \bullet, b : \bullet, \\ \qquad w : \bullet \parr \bullet, y : \bullet \otimes \bullet, c : \bullet, d : \bullet \end{array}
  }{
    (\nu xy)(\nu uw)(x(v, x'); u[a, b] \mid w(z, w'); y[c, d]) \vdash a : \bullet, b : \bullet, c : \bullet, d : \bullet
  } [\text{typ-res}]^2
} [\text{typ-par}]
$$

Enforcing Deadlock Freedom

ACP: Asynchronous CP

- An asynchronous variant of Wadler's Classical Processes (CP).

ACP: Asynchronous CP

- An asynchronous variant of Wadler's Classical Processes (CP).
- Obtained from AP by a minor yet crucial modification.
  - Replace:

$$\frac{P \vdash \Gamma \qquad Q \vdash \Delta}{P \mid Q \vdash \Gamma, \Delta}\ \text{[typ-par]} \qquad \frac{P \vdash \Gamma, x : A, y : \overline{A}}{(\nu xy)P \vdash \Gamma}\ \text{[typ-res]}$$

  + Add:

$$\boxed{\frac{P \vdash \Gamma, x : A \qquad Q \vdash \Delta, y : \overline{A}}{(\nu xy)(P \mid Q) \vdash \Gamma, \Delta}\ \text{[typ-cut]}}$$

ACP: Asynchronous CP

- An asynchronous variant of Wadler's Classical Processes (CP).
- Obtained from AP by a minor yet crucial modification.
  − Replace:

$$\frac{P \vdash \Gamma \qquad Q \vdash \Delta}{P \mid Q \vdash \Gamma, \Delta}\ \text{[typ-par]} \qquad \frac{P \vdash \Gamma, x : A, y : \overline{A}}{(\nu xy)P \vdash \Gamma}\ \text{[typ-res]}$$

  + Add:

$$\boxed{\frac{P \vdash \Gamma, x : A \qquad Q \vdash \Delta, y : \overline{A}}{(\nu xy)(P \mid Q) \vdash \Gamma, \Delta}\ \text{[typ-cut]}}$$

- ACP guarantees DF by ruling out all possible cyclic dependencies: sub-processes can connect along **exactly one pair of names**.

ACP: Asynchronous CP

- An asynchronous variant of Wadler's Classical Processes (CP).
- Obtained from AP by a minor yet crucial modification.

  – Replace:

$$\frac{P \vdash \Gamma \qquad Q \vdash \Delta}{P \mid Q \vdash \Gamma, \Delta}\ [\text{typ-par}] \qquad \frac{P \vdash \Gamma, x : A, y : \overline{A}}{(\nu x y) P \vdash \Gamma}\ [\text{typ-res}]$$

  + Add:

$$\boxed{\frac{P \vdash \Gamma, x : A \qquad Q \vdash \Delta, y : \overline{A}}{(\nu x y)(P \mid Q) \vdash \Gamma, \Delta}\ [\text{typ-cut}]}$$

- ACP guarantees DF by ruling out all possible cyclic dependencies: sub-processes can connect along **exactly one pair of names**.
- We write $^{\mathtt{C}}\vdash$ instead of $\vdash$ to distinguish the two type systems.

The Deadlocked Process, Revisited

- The deadlocked process

$$(\nu xy)(\nu uw)(x(v, x'); u[a, b] \mid w(z, w'); y[c, d])$$

is not typable under ${}^{\mathtt{C}}\!\vdash$:
its parallel sub-processes are connected on two pairs of names.

The Deadlocked Process, Revisited

- The deadlocked process

$$(\nu xy)(\nu uw)(x(v, x'); u[a, b] \mid w(z, w'); y[c, d])$$

is not typable under $\overset{\mathsf{C}}{\vdash}$:
its parallel sub-processes are connected on two pairs of names.

- A well-typed variant, obtained by 'parallelizing' the right sub-process:

$$(\nu xy)\Big((\nu uw)(x(v, x'); u[a, b] \mid w(z, w'); 0) \mid y[c, d]\Big) \overset{\mathsf{C}}{\vdash} \begin{array}{l} a : \bullet, b : \bullet, \\ c : \bullet, d : \bullet \end{array}$$

Properties of ACP

As in AP, well-typed processes respect their protocols:

Theorem (Type Preservation for ACP)

Given $P \overset{\mathtt{C}}{\vdash} \Gamma$ *and* Q *such that* $P \overset{\mathtt{C}}{\equiv} Q$ *or* $P \overset{\mathtt{C}}{\longrightarrow} Q$*, we have* $Q \overset{\mathtt{C}}{\vdash} \Gamma$*.*

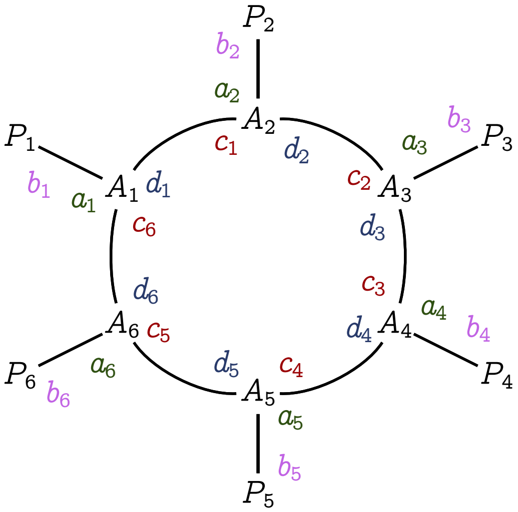
Thanks to the cut rule, we now also have:

Theorem (Deadlock Freedom for ACP)

Given $P \overset{\mathtt{C}}{\vdash} \emptyset$*, if* $P \overset{\mathtt{C}}{\not\longrightarrow}$*, then* $P \overset{\mathtt{C}}{\equiv} \mathbf{0}$*.*

Not Typable in ACP: Milner's Cyclic Scheduler

$$
\begin{aligned}
(\nu\, c_1\, d_1) \cdots (\nu\, c_6\, d_6)\Big((\nu\, a_1\, b_1)(A_1 \mid P_1) \mid\\
(\nu\, a_2\, b_2)(A_2 \mid P_2) \mid\\
\vdots\qquad\qquad\\
(\nu\, a_6\, b_6)(A_6 \mid P_6)\Big)
\end{aligned}
$$



Can We Get It All?
DF in Circular Topologies

APCP: Priority-Based DF Enforcement

Asynchronous **Priority-Based** Classical Processes:

- Back to **separate rules** for parallel composition and restriction (as in AP)
- Excluding cyclic dependencies using **priorities**

APCP: Priority-Based DF Enforcement

- We write $\pi, \rho, \ldots$ to denote priorities (integers).
  We write ω to denote the 'ultimate priority': it is greater than all other priorities and cannot be increased further.

APCP: Priority-Based DF Enforcement

- We write $\pi, \rho, \ldots$ to denote priorities (integers).
  We write ω to denote the 'ultimate priority': it is greater than all other priorities and cannot be increased further.
- Types are as before, now annotated with priorities:

$$A, B ::= A \otimes^{\pi} B \mid A \parr^{\pi} B \mid \oplus^{\pi}\{i : A\}_{i \in I} \mid \&^{\pi}\{i : A\}_{i \in I} \mid \bullet$$

APCP: Priority-Based DF Enforcement

- We write $\pi, \rho, \ldots$ to denote priorities (integers).
  We write ω to denote the 'ultimate priority': it is greater than all other priorities and cannot be increased further.
- Types are as before, now annotated with priorities:

$$A, B ::= A \otimes^{\pi} B \mid A \parr^{\pi} B \mid \oplus^{\pi}\{i : A\}_{i \in I} \mid \&^{\pi}\{i : A\}_{i \in I} \mid \bullet$$

- For session type A, $pr(A)$ denotes its *priority*:

$$pr(A \otimes^{\pi} B) \triangleq pr(A \parr^{\pi} B) \triangleq pr(\oplus^{\pi}\{i : A_i\}_{i \in I}) \triangleq pr(\&^{\pi}\{i : A_i\}_{i \in I}) \triangleq \pi$$

$$pr(\bullet) \triangleq \omega$$

APCP: Priority-Based DF Enforcement

- We write $\pi, \rho, \ldots$ to denote priorities (integers).
  We write ω to denote the 'ultimate priority': it is greater than all other priorities and cannot be increased further.
- Types are as before, now annotated with priorities:

$$A, B ::= A \otimes^{\pi} B \mid A \invamp^{\pi} B \mid \oplus^{\pi}\{i : A\}_{i \in I} \mid \&^{\pi}\{i : A\}_{i \in I} \mid \bullet$$

- For session type A, $pr(A)$ denotes its *priority*:

$$pr(A \otimes^{\pi} B) \triangleq pr(A \invamp^{\pi} B) \triangleq pr(\oplus^{\pi}\{i : A_i\}_{i \in I}) \triangleq pr(\&^{\pi}\{i : A_i\}_{i \in I}) \triangleq \pi$$
$$pr(\bullet) \triangleq \omega$$

- Duality: $A = \overline{B}$ iff (i) actions in A and B are complementary (as before) and (ii) their priority annotations coincide.

The Laws of Priorities

Key Idea

Prefixes with **lower priority** must not be blocked by prefixes with **higher priority**.

The Laws of Priorities

Key Idea
Prefixes with **lower priority** must not be blocked by prefixes with **higher priority**.

Typing enforces the following laws:

1. Outputs with priority π must have messages and continuations with priority **strictly larger** than π;
2. A prefix typed with priority π must be prefixed only by inputs with priority **strictly smaller** than π;
3. Dual prefixes leading to a synchronization must have **equal priorities**.

Typing rules of APCP: AP with Priorities

[typ-send]

$$\frac{\pi < pr(A), pr(B)}{x[y, z] \vdash x : A \otimes^{\pi} B, y : \overline{A}, z : \overline{B}}$$

[typ-recv]

$$\frac{P \vdash \Gamma, y : A, z : B \qquad \pi < pr(\Gamma)}{x(y, z); P \vdash \Gamma, x : A \parr^{\pi} B}$$

Typing rules of APCP: AP with Priorities

[typ-send]

$$\frac{\pi < pr(A), pr(B)}{x[y,z] \vdash x : A \otimes^{\pi} B, y : \overline{A}, z : \overline{B}}$$

[typ-recv]

$$\frac{P \vdash \Gamma, y : A, z : B \qquad \pi < pr(\Gamma)}{x(y,z); P \vdash \Gamma, x : A \parr^{\pi} B}$$

[typ-sel]

$$\frac{j \in I \qquad \pi < pr(A_j)}{x[z] \triangleleft j \vdash x : \oplus^{\pi}\{i : A_i\}_{i \in I}, z : \overline{A_j}}$$

[typ-bra]

$$\frac{\forall i \in I : P_i \vdash \Gamma, z : A_i \qquad \pi < pr(\Gamma)}{x(z) \triangleright \{i : P_i\}_{i \in I} \vdash \Gamma, x : \&^{\pi}\{i : A_i\}_{i \in I}}$$

(Other rules unchanged, with the extended duality)

Properties of APCP

Theorem (Type Preservation for APCP)

Given $P \vdash^{\mathrm{P}} \Gamma$ *and* Q *such that* $P \equiv Q$ *or* $P \longrightarrow Q$, *we have* $Q \vdash^{\mathrm{P}} \Gamma$.

Theorem (Deadlock Freedom for APCP)

Given $P \vdash^{\mathrm{P}} \emptyset$, *if* $P \not\longrightarrow$, *then* $P \equiv 0$.

Properties of APCP

Theorem (Type Preservation for APCP)

Given $P \ {}^{\mathtt{P}}\!\vdash \Gamma$ *and* Q *such that* $P \equiv Q$ *or* $P \longrightarrow Q$*, we have* $Q \ {}^{\mathtt{P}}\!\vdash \Gamma$*.*

Theorem (Deadlock Freedom for APCP)

Given $P \ {}^{\mathtt{P}}\!\vdash \emptyset$*, if* $P \not\longrightarrow$*, then* $P \equiv 0$*.*

Let $\vdash$, ${}^{\mathtt{C}}\!\vdash$, and ${}^{\mathtt{P}}\!\vdash$ denote **sets of well-typed processes** in AP, ACP, and APCP, respectively. We immediately have:

Theorem (Comparative Expressiveness)

We have ${}^{\mathtt{C}}\!\vdash \ \subset \ {}^{\mathtt{P}}\!\vdash \ \subset \ \vdash$*.*

Examples

- Recall the deadlocked process; it is rejected under $^{\mathtt{P}}\vdash$:

$$(\nu xy)(\nu uw)(x(v, x'); u[a, b] \mid w(z, w'); y[c, d])$$

Suppose the actions on x, y have priority π, and that the actions on u, w have priority ρ. Using Rule [typ-recv] we require $\pi < \rho$ and $\rho < \pi$.

Examples

- Recall the deadlocked process; it is rejected under ${}^{\mathrm{P}}\!\vdash$:

$$(\nu xy)(\nu uw)(x(v, x'); u[a, b] \mid w(z, w'); y[c, d])$$

  Suppose the actions on x, y have priority π, and that the actions on u, w have priority ρ. Using Rule [typ-recv] we require $\pi < \rho$ and $\rho < \pi$.

- Recall the non-deadlocked variant:

$$(\nu xy)\Big((\nu uw)(x(v, x'); u[a, b] \mid w(z, w'); 0) \mid y[c, d]\Big)$$

  In this case, ${}^{\mathrm{P}}\!\vdash$ only requires $\pi < \rho$ but not $\rho < \pi$, so no cyclic dependency is detected—the process is considered well typed.

Application: DF in Concurrent Functional Sessions

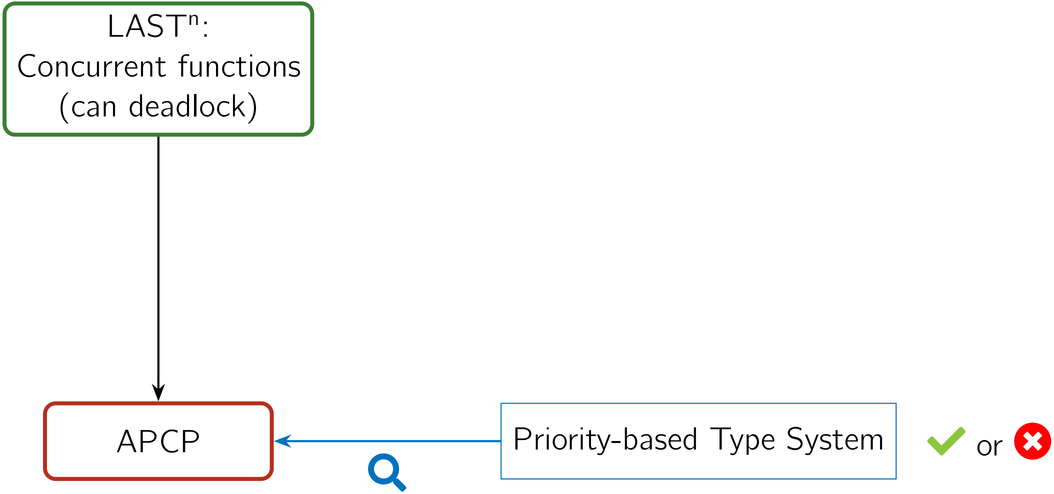
LAST^n:
Concurrent functions
(can deadlock)

Application: DF in Concurrent Functional Sessions

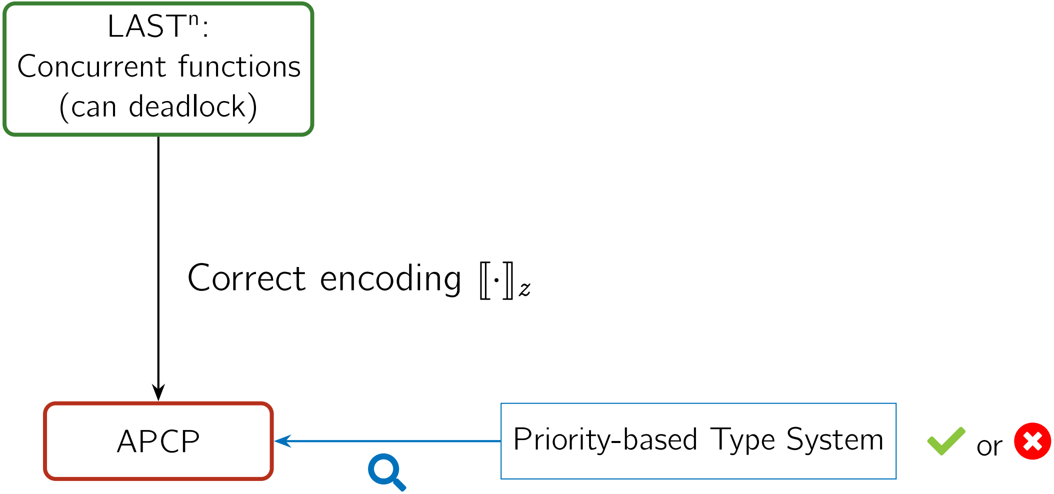
LAST^n:
Concurrent functions
(can deadlock)

APCP ← Priority-based Type System ✓ or ✗

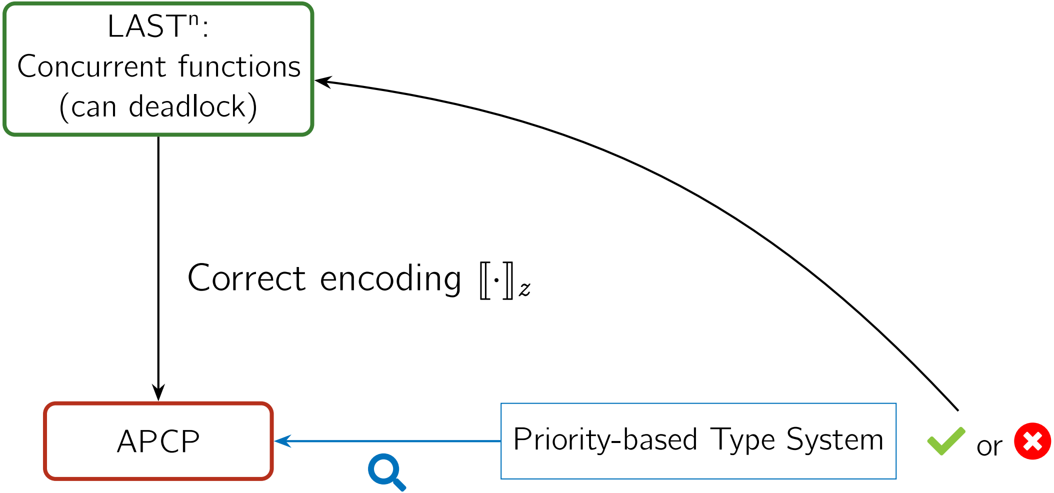
Application: DF in Concurrent Functional Sessions



Application: DF in Concurrent Functional Sessions



Application: DF in Concurrent Functional Sessions



Summing Up

Summary Three typed asynchronous π-calculi based on Linear Logic:

1. AP: processes correctly follow protocols but deadlock
2. ACP: AP with logic-based composition, DF for tree-like topologies
3. APCP: extension of AP with priorities, DF for circular topologies

Summing Up

Summary Three typed asynchronous π-calculi based on Linear Logic:

1. AP: processes correctly follow protocols but deadlock
2. ACP: AP with logic-based composition, DF for tree-like topologies
3. APCP: extension of AP with priorities, DF for circular topologies

Further Results

- Typed specification of the cyclic scheduler
- Processes with recursion and recursive session types (tail-recursive)
- Full treatment of LAST^n and its encoding into APCP
- Analysis of multiparty protocols with APCP

Summing Up

Summary Three typed asynchronous π-calculi based on Linear Logic:

1. AP: processes correctly follow protocols but deadlock
2. ACP: AP with logic-based composition, DF for tree-like topologies
3. APCP: extension of AP with priorities, DF for circular topologies

Further Results

- Typed specification of the cyclic scheduler
- Processes with recursion and recursive session types (tail-recursive)
- Full treatment of LAST^n and its encoding into APCP
- Analysis of multiparty protocols with APCP

Ongoing Work

- APCP extended with PCF-like servers (with J. Jaramillo and D. Mazza)
- Priority-based DF in FreeST (with A. Mordido)

Asynchronous Session-based Concurrency: Deadlock Freedom by Typing

Jorge A. Pérez
www.jperez.nl

University of Groningen, The Netherlands

Dutch Winter School 2026
(Part 3)