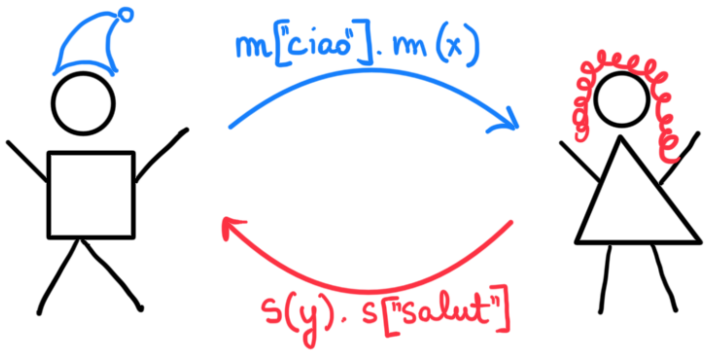
SESSION TYPES

Lecture 4: Extensions

MGS 2024

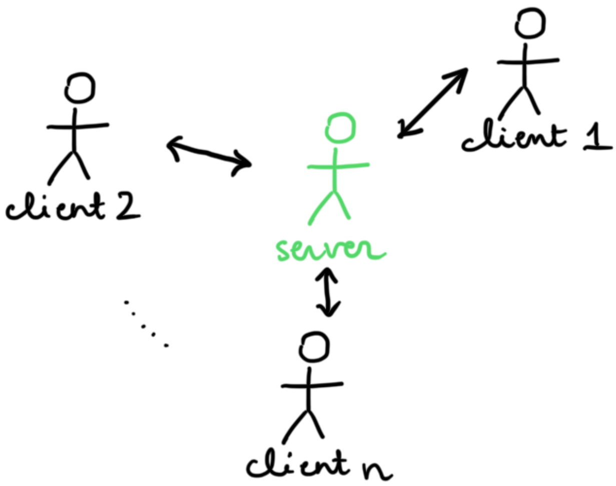
Matteo Acclavio ⊗ Sonia Marin



So far we have considered a finite, linear world where each channel is possessed by exactly one thread and computation cannot execute an unbounded number of steps.

for simplicity of exposition similar constructions can also be added to the multyparty system

Today we will extend the binary session type system with constructions that represent more realistic scenarios: shared channels and infinite behaviours.



Infinite Behaviour

① Processes

Example : $P = u \triangleright \{ init : u[1].\ u[\ ].\ inact$
$incr : u(x).u[x+1].\ u[\ ].\ inact$
$sum : u(x).\ u(y).\ u[x+y].\ u[\ ].\ inact \}$

$P^* = u \triangleright \{ init : u[1].$ repeat from start,
$incr : u(x).u[x+1].$ repeat from start,
$sum : u(x).\ u(y).\ u[x+y].$ repeat from start,
$exit : u[\ ].\ inact \}$

In order to write infinite processes in finite form we use a notation for processes based on recursive definitions :

$$A(u_1, \dots, u_n) =_{rec} P$$

where $\{u_1 \dots u_n\} = fv(P)$
and A can occur in P

process identifiers a new base set $(A, B, C \dots)$
to be added to our syntax of processes

Example To define P^* above we write :

$A(u) = u \triangleright \{ init : u[1].\ A(u),$
$incr : u(x).u[x+1].\ A(u),$
$sum : u(x).\ u(y).\ u[x+y].\ A(u),$
$exit : u[\ ].\ inact \}$

Additional condition : We want the unfolding of process equations to eventually exhibit a "proper" process constructor
(not an identifier

The easiest way to ensure it is to ask each equation to be guarded

i.e. on the right the identifier must occur under such a "proper" constructor

② Operational semantics

A call $A(v)$ to a process defined by equation $A(u) = P$ produces a new copy of P where bound variable u is replaced by free variable v.

That is, we extend the previously defined semantics with:

$$A(v_1, \dots, v_n) \longrightarrow P[v_1/u_1, \dots, v_n/u_n] \qquad \text{if } A(\tilde{u}) =_{rec} P$$

Recursive definitions do not necessarily lead to infinite computation

This process:

$$\begin{aligned} A(u) = u \triangleright \{ & \text{init} : u[1].A(u), \\ & \text{incr} : u(x).u[x+1].A(u), \\ & \text{sum} : u(x).u(y).u[x+y].A(u), \\ & \text{exit} : u[\,].\text{inact} \} \end{aligned}$$

offers infinite and finite paths.

The client chooses: any path containing exit is necessarily finite

③ Types

A common way of introducing infinite types is via the rec type.

(not specific to session types

rec $X.S$ represents an infinite type obtained by repeatedly substituting S for X in S.

This is taken to be equirecursive = any finite representations of the same type are considered equal (hence interchangeable in all contexts)

Example $$S = \mathrm{rec}\,X.\,\&\left\{\begin{array}{l}\mathrm{init} : [\mathrm{nat}] \triangleleft X\,, \\ \mathrm{incr} : (\mathrm{nat}) \triangleleft [\mathrm{nat}] \triangleleft X\,, \\ \mathrm{sum} : (\mathrm{nat}) \triangleleft (\mathrm{nat}) \triangleleft [\mathrm{nat}] \triangleleft X\,, \\ \mathrm{exit} : \mathrm{close} \end{array}\right\}$$

This requires us to extend the syntax of session types:

$$S ::= \ \dots \ \mid \ \mathrm{rec}\,X.\,S \ \mid \ X$$

from a set of type identifiers

This can of course define more complex types using nested recursion.

Again, we will impose for the type identifiers to be guarded so that the unfolding of a rec type is not trivial.

What does duality mean once types can be infinite?

In the finite case we had: $$\frac{S \perp S'}{[T] \triangleleft S \perp (T) \triangleleft S'} \quad \text{etc.}$$

same type

(just another way to write $([T] \triangleleft S)^{\perp} = (T) \triangleleft S^{\perp}$)

But now we read the rules: coinductively rather than inductively

And we add those for rec: $$\frac{S[\mathrm{rec}X.S/X] \perp S'}{\mathrm{rec}\,X.\,S \perp S'}$$ and vice-versa

④ Type system

We now look at the changes required to type recursive processes.

To simplify the presentation one can associate a type to each free variable in a recursive equation:

$$A(u_1 : S_1, \ldots, u_n : S_n) =_{rec} P$$

Then a process call $A(v_1, \ldots, v_n)$ is typable in a context containing an entry for each v_i whose type is taken from the equation.

↳ consequence of linearity

The typing rule is then simply

$$\frac{}{u_1 : S_1, \ldots, u_n : S_n \vdash A(u_1, \ldots, u_n)} \text{(CALL)}$$

assuming that $u_1 : S_1, \ldots, u_n : S_n \vdash P$

This means a suitable typing derivation needs to be given to show that the equation is well-formed.

Example : $B(u) =_{rec} \underbrace{u(x).\, u[x].\, B(u)}_{P}$

$$S = \text{rec } X.\, (\text{nat}) \triangleleft [\text{nat}] \triangleleft X$$

$$\dfrac{\dfrac{\dfrac{}{x : \text{nat} \Vdash x : \text{nat}} \qquad \dfrac{}{u : S \vdash B(u)}\text{(CALL)}}{u : [\text{nat}] \triangleleft S,\, x : \text{nat} \vdash u[x].\, B(u)}\text{(SEND)}}{u : S = (\text{nat}) \triangleleft [\text{nat}] \triangleleft S \vdash P}\text{(RECV)}$$

↑ equirecursion

Type safety properties are preserved by addition of recursive behaviour.

Shared Channels

So now a server can offer choices repeatedly to one client, we would also like to be able to represent a server expected to interact with any number of clients.

By adding shared channels, we might add <u>races</u> (processes can compete for ressources) and <u>non-determinism</u>.

We want to relax the condition that there is a <u>unique</u> process owning each of the endpoints.

<u>Example</u>: We already considered this process informally

$$(\nu uv)\big(u[n].P \mid u[n'].Q \mid v(x).R\big)$$

$$\swarrow \qquad\qquad \searrow$$

$$(\nu uv)\big(P \mid u[n'].Q \mid R[n/x]\big) \qquad\qquad (\nu uv)\big(u[n].P \mid Q \mid R[n'/x]\big)$$

This formally requires the more general reduction rules:

$$(\nu uv)\big(u(x).P \mid v[e].Q \mid R\big) \longrightarrow (\nu uv)\big(P[c/x] \mid Q \mid R\big) \quad \text{if } e \downarrow c$$

$$(\nu uv)\big(u \triangleright \{l : P_l\}_{l \in L} \mid v \triangleleft k : Q \mid R\big) \longrightarrow (\nu uv)\big(P_k \mid Q \mid R\big) \quad \text{if } k \in L$$

R is allowed to communicate on u or on v.

Channels meant to be shared require a certain <u>uniformity</u>:
if a shared channel endpoint receives an element of type T,
the subsequent behaviour can only receive an element of type T

<u>Example</u>

$$\begin{array}{l} a : \&\{one : close\} \triangleleft \&\{two : close\} \triangleleft close \\ u : \&\{one : close\},\quad w : \&\{two : close\} \end{array} \vdash \quad P = a[u].a[w]$$

$$\underbrace{(\nu tu)(\nu vw)(\nu ab)}_{\text{all channels are shared}}\big(t \triangleleft one \mid v \triangleleft two \mid P \mid P \mid b(x).b(y).y \triangleright \{two : inact\}\big)$$

$$\longrightarrow (\nu tu)(\nu vw)\big(t \triangleleft one \mid v \triangleleft two \mid a[w] \mid a[w] \mid u \triangleright \{two : inact\}\big)$$

$\hookrightarrow$ runtime error $\hookleftarrow$

All messages sent on a shared channel must have the same type to match the expectations of all receivers independently of how many messages have been already emitted.
(and same for the labels that are repeatedly offered)

There are different ways to capture this constraint, possibly the most restrictive (but also most common, I think) is to treat linear channels and shared channels completely independently

That requires a new base set of
shared channel names denoted by a, b ...

And an extension of the typing judgement to

$$\Gamma ; \underset{\substack{\uparrow \\ \text{shared context}}}{\Delta} \vdash P$$

shared context = NOT treated linearly

What does it mean for the typing rules?

To be changed during the lecture

$$\frac{}{\cdot \vdash \mathbf{inact}} \text{(INACT)}$$

$$\frac{\Gamma \vdash P}{\Gamma, u : \mathrm{wait} \vdash u().P} \text{(WAIT)} \qquad \frac{\Gamma, y : T, u : S \vdash P}{\Gamma, u : (T) \triangleleft S \vdash u(y).P} \text{(RECV)}$$

$$\frac{\Gamma \vdash P}{\Gamma, u : \mathrm{close} \vdash u[\,].P} \text{(CLOSE)} \qquad \frac{\Gamma \Vdash e : T \qquad \Gamma, u : S \vdash P}{\Gamma, u : [T] \triangleleft S \vdash u[e].P} \text{(SEND)}$$

$$\frac{\Gamma \vdash P \qquad \Gamma' \vdash Q}{\Gamma, \Gamma' \vdash (P \mid Q)} \text{(PAR)} \qquad \frac{\Gamma, u : S, v : S^{\perp} \vdash P}{\Gamma \vdash (\nu u v) P} \text{(RES)}$$

$$\frac{\{\Gamma, x : S_\ell \vdash P_\ell\}_{\ell \in L}}{\Gamma, x : \&\{\ell : S_\ell\}_{\ell \in L} \vdash x \triangleright \{\ell : P_\ell\}_{\ell \in L}} \text{(BRA)}$$

$$\frac{\Gamma,\ x:S_k \vdash P}{\Gamma,\ x:\oplus\{l:S_l\}_{l\in L} \vdash x \triangleleft k : P}\ (\text{SEL})_{k\in L}$$

Note that a shared channel need not (cannot!) be closed.
It might be used by an unbounded number of threads (possibly none) that the type system does not keep track of.

Regarding <u>type safety</u>, with suitable adaptations to the definition of runtime errors, it is still possible to prove <u>preservation</u> and <u>absence of immediate errors</u>.
But of course not that typable processes do not contain races

---

In this course, we have introduced the <u>key concepts</u> of session types

- <u>π-calculus with sessions</u> and its operational semantics with basic message passing and choice operations then extended <u>with recursive behaviour and sharing</u>

- <u>session types</u> for this calculus and how session typing rules out some undesirable behaviours

- the idea of <u>global types</u> to account for more precise <u>multiparty</u> interactions

  we have concidered a <u>simplified setting</u> with only one session but you should now be equiped to read on multiparty session types with several sessions, shared types and recursion (as we saw in binary) and beyond (e.g. compatibility vs projection)

As mentioned, this has evolved into a very active research area and influenced the design of new or existing (real-world) programming lang.