Our sponsors make OPLSS possible.

Diamond Sponsor

https://www.janestreet.com/

Platinum Sponsors

https://www.nsf.gov/

https://www.sigplan.org/

Silver Sponsors

Session Types & Mailbox Types
Lecture 1: Introduction to Session Types

Simon Gay
School of Computing Science
University of Glasgow, UK


Glasgow
Edinburgh
United
Kingdom
Isle of Man
Dublin
Ireland
Manchester
Birmingham
London

Overview of the course

- Session types and mailbox types are two forms of *behavioural types*.
- Key idea of session types: express a communication protocol (on a bidirectional point-to-point channel) as a type, and develop a static type system.
- Key idea of mailbox types: change the communication model to the actor paradigm — many senders/one receiver, no message ordering.

Overview of the course

- Lecture 1: basic concepts of session types in pi calculus.
- Lecture 2: session types in a functional language.
- Lecture 3: session types and linear logic.
- Lecture 4: mailbox types.

Session Types & Mailbox Types
Lecture 1: Introduction to Session Types

Simon Gay
School of Computing Science
University of Glasgow, UK

A session type specifies a communication protocol

Protocol

Send one message, which is an integer, then close the channel.

Type

$$!\mathsf{int}.\mathsf{end}_!$$

A session type specifies a communication protocol

Dual protocol

Receive one message, which is an integer, then wait for the channel to be closed.

Type

$$?\mathsf{int}.\mathsf{end}_?$$

Communication can go in both directions

Protocol

Send two integers, then receive an integer, then close the channel.

Type

$$!\mathsf{int}.!\mathsf{int}.?\mathsf{int}.\mathsf{end}_!$$

Communication can go in both directions

Dual protocol

Receive two integers, then send an integer, then wait for the channel to be closed.

Type

$$?\mathsf{int}.?\mathsf{int}.!\mathsf{int}.\mathsf{end}_?$$

Protocols can include choice

Protocol (a maths server)

Offer a choice between two operations: addition (plus) and negation (neg). Continue appropriately.

Type

$$\&\{\mathsf{plus}\colon ?\mathsf{int}.?\mathsf{int}.!\mathsf{int}.\mathsf{end}_?, \mathsf{neg}\colon ?\mathsf{int}.!\mathsf{int}.\mathsf{end}_?\}$$

Protocols can include choice

Dual protocol (a client of the maths server)

Make a choice between two operations: addition (plus) and negation (neg). Continue appropriately.

Type

$$\oplus\{\mathsf{plus}\colon !\mathsf{int}.!\mathsf{int}.?\mathsf{int}.\mathsf{end}_!, \mathsf{neg}\colon !\mathsf{int}.?\mathsf{int}.\mathsf{end}_!\}$$

Protocols can include recursion

Protocol

A recursive version of the maths server, with an exit option.

Type

$$S = \&\{\mathsf{plus}\colon ?\mathsf{int}.?\mathsf{int}.!\mathsf{int}.S, \mathsf{neg}\colon ?\mathsf{int}.!\mathsf{int}.S, \mathsf{quit}\colon \mathsf{end}_?\}$$

Protocols can include recursion

Protocol

A recursive version of the maths client; the client decides when to stop.

Type

$$T = \oplus\{\mathsf{plus}\colon !\mathsf{int}.!\mathsf{int}.?\mathsf{int}.T, \mathsf{neg}\colon !\mathsf{int}.?\mathsf{int}.T, \mathsf{quit}\colon \mathsf{end}_!\}$$

Messages can be channels

Protocol

Send a channel on which an integer will be received.

Type

$$!(?\mathrm{int}.\mathrm{end}_?).\mathrm{end}_!$$

This is called *delegation*.

For example, a maths client can select the operation and send the parameters, then send the channel to another process that will receive the result.

Messages can be channels

Dual protocol

Receive a channel on which an integer will be received.

Type

$$?(?\mathsf{int}.\mathsf{end}_?).\mathsf{end}_?$$

Assumptions about communication

- Point-to-point communication channels in a concurrent or distributed setting.
- A process can use any number of channels.
- A channel has two *endpoints*.
- Channels are bidirectional.
- Communication is reliable.
- Message ordering is preserved.
- Communication can be synchronous or asynchronous.

If we consider asynchronous communication, we are thinking of something similar to TCP/IP sockets.

Ownership of channel endpoints

A session type system aims to provide a compile-time guarantee that the protocol is followed at runtime.

To achieve this, each channel endpoint must be owned by a unique process.

This is enforced by techniques from linear type theory.

(In the case of *stateless* protocols it is safe to allow sharing of channel endpoints.)

The safety guarantees of session types

No communication mismatch

If the owner of one endpoint of a channel sends a message, then the owner of the other endpoint is expecting to receive a message, and the message types match; conversely if the owner of one endpoint is expecting to receive a message of a certain type, then the owner of the other endpoint will send a message of the expected type.

Session fidelity

The sequence and types of the messages sent on a channel at runtime match the session type of the channel.

Origin and development of session types

The original papers

Kohei Honda. "Types for Dyadic Interaction", CONCUR 1993.

Kaku Takeuchi, Kohei Honda, Makoto Kubo. "An Interaction-Based Language and Its Typing System", PARLE 1994.

Kohei Honda, Vasco Vasconcelos, Makoto Kubo. "Language Primitives and Type Discipline for Structured Communication-Based Programming", ESOP 1998.

The next paper ("first follower")

Simon Gay, Malcolm Hole. "Types and Subtypes for Client-Server Interactions", ESOP 1999.

Origin and development of session types

Session types and functional programming

Matthias Neubauer, Peter Thiemann. "An Implementation of Session Types", PADL 2004.

Vasco Vasconcelos, António Ravara, Simon Gay. "Session Types for Functional Multithreading", CONCUR 2004.

Simon Gay, Vasco Vasconcelos. "Linear type theory for asynchronous session types", JFP 2010.

Origin and development of session types

The extension to multiparty protocols

Kohei Honda, Nobuko Yoshida, Marco Carbone. "Multiparty asynchronous session types", POPL 2008.

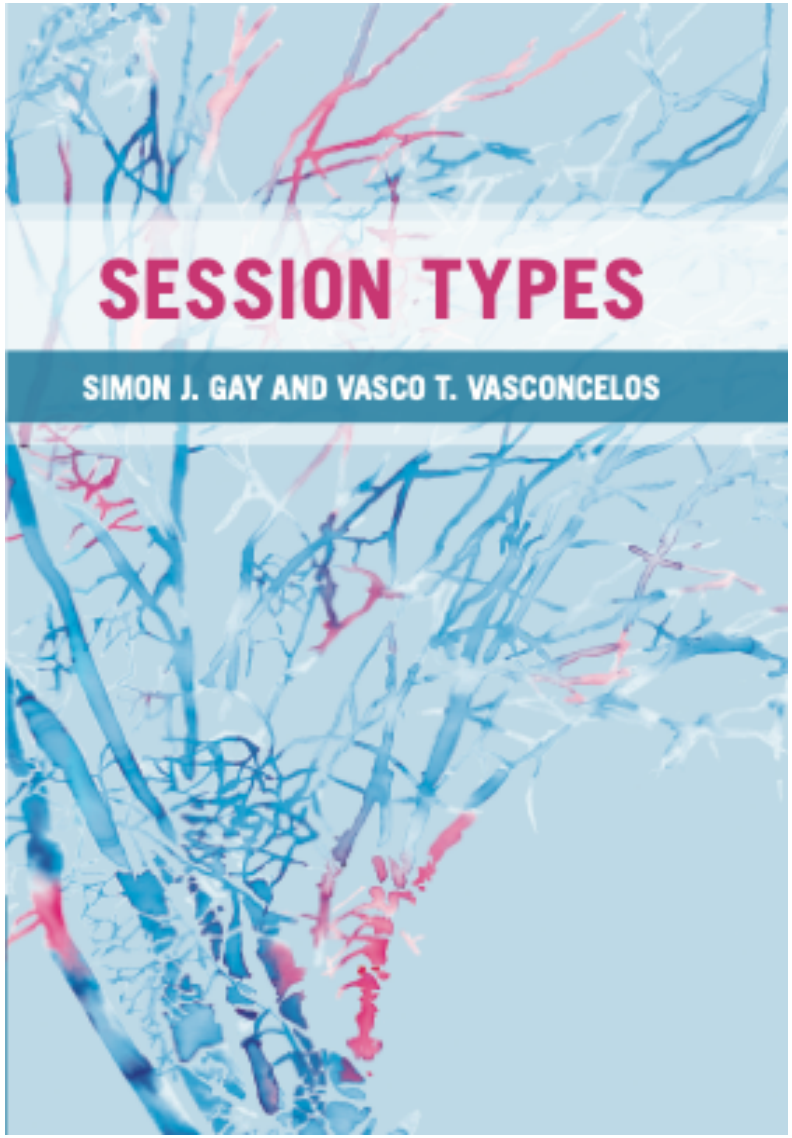
The Curry-Howard correspondence with linear logic

Luís Caires, Frank Pfenning. "Session types as intuitionistic linear propositions", CONCUR 2010.

Philip Wadler. "Propositions as sessions", ICFP 2012.

Luís Caires, Frank Pfenning, Bernardo Toninho. "Linear logic propositions as session types", MSCS 2016.

Reference for Lectures 1, 2, 3



Pi calculus syntax (finite behaviour only)

P ::=		
	0	inaction
	$x?.P$	wait
	$x!.P$	close
	$x?y.P$	receive
	$x!y.P$	send
	$P \mid P$	parallel composition
	$(\nu xy)P$	scope restriction

Non-standard: wait/close, double-binding ν.

Missing for now: recursion/repetition, choice.

We may also assume syntax for integers etc., in examples.

Pi calculus semantics 1: structural congruence

$$P \mid Q \equiv Q \mid P \qquad \text{(C-Comm)}$$

$$(P \mid Q) \mid M \equiv P \mid (Q \mid M) \qquad \text{(C-Assoc)}$$

$$P \mid 0 \equiv P \qquad \text{(C-Neut)}$$

$$(\nu xy)(P \mid Q) \equiv (\nu xy)P \mid Q \qquad \text{(C-Scope)}$$

$$(\nu xy)P \equiv (\nu yx)P \qquad \text{(C-SwapC)}$$

$$(\nu xy)(\nu zw)P \equiv (\nu zw)(\nu xy)P \qquad \text{(C-SwapB)}$$

Plus congruence rules such as

$$0 \equiv 0 \qquad \text{(C-Inact)}$$

$$\frac{P \equiv Q}{(\nu xy)P \equiv (\nu xy)Q} \qquad \text{(C-Res)}$$

This is standard, adapted for the double-binder syntax and wait/close.

Pi calculus semantics 2: reduction

$$(\nu xy)(x?.P \mid y!.Q) \to P \mid Q \qquad \text{(R-Close)}$$

$$(\nu xy)(x?z.P \mid y!w.Q) \to (\nu xy)(P[w/z] \mid Q) \qquad \text{(R-Com)}$$

$$\frac{P \to P'}{P \mid Q \to P' \mid Q} \qquad \text{(R-Par)}$$

$$\frac{P \to Q}{(\nu xy)P \to (\nu xy)Q} \qquad \text{(R-Res)}$$

$$\frac{P \equiv P' \qquad P' \to Q' \qquad Q' \equiv Q}{P \to Q} \qquad \text{(R-Struct)}$$

This is standard, adapted for double binders and wait/close.

Reduction example with delegation

The process

$$(\nu ab)w!a.y!b.(w!.0 \mid y!.0)$$

creates co-variables a and b and sends them on separate channels w and y, subsequently closing w and y.

The process

$$x?u.u!5.x?.u!.0$$

receives channel u from x and uses it to send 5, subsequently waiting for x to be closed and then closing u.

The process

$$z?v.z?.v?t.v?.Q$$

receives channel v from z and uses it to receive t, which can be used by the continuation Q after z and v have been closed.

Reduction example with delegation

This is the reduction sequence:

$$(\nu yz)(\nu wx)((\nu ab)w!a.y!b.(w!.0 \mid y!.0)) \mid x?u.u!5.x?.u!.0 \mid z?v.z?.v?t.v?.Q)$$
$$\downarrow$$
$$(\nu ab)(\nu yz)(\nu wx)(y!b.(w!.0 \mid y!.0)) \mid a!5.x?.a!.0 \mid z?v.z?.v?t.v?.Q)$$
$$\downarrow$$
$$(\nu ab)(\nu wx)(\nu yz)(w!.0 \mid y!.0 \mid a!5.x?.a!.0 \mid z?.b?t.b?.Q)$$
$$\downarrow$$
$$(\nu ab)(\nu wx)(w!.0 \mid a!5.x?.a!.0 \mid b?t.b?.Q)$$
$$\downarrow$$
$$(\nu ab)(\nu wx)(w!.0 \mid x?.a!.0 \mid b?.Q[5/t])$$
$$\downarrow$$
$$(\nu ab)(a!.0 \mid b?.Q[5/t])$$
$$\downarrow$$
$$Q[5/t]$$

Choice

Extend the syntax:

$$
\begin{array}{rll}
P & ::= & \ldots \\
 & & x \rhd \{l\colon P_l\}^{l \in L} \qquad \text{external choice} \\
 & & x \lhd l.P \qquad \text{internal choice}
\end{array}
$$

Extend the reduction relation:

$$
\frac{k \in L}{(\nu xy)(y \rhd \{l\colon P_l\}^{l \in L} \mid x \lhd k.Q) \rightarrow (\nu xy)(P_k \mid Q)} \quad \text{(R-Choice)}
$$

Example: one-shot maths server

$$
\begin{array}{ll}
(\nu xy)(& x \rhd \{\mathsf{plus}\colon x?a.x?b.x!(a+b).x?.0, \mathsf{neg}\colon x?a.x!(-a).x?.0\} \\
& \mid \\
& y \lhd \mathsf{plus}.y!2.y!3.y?u.y!.P(u) \quad) \\
& \downarrow \\
(\nu xy)(& x?a.x?b.x!(a+b).x?.0 \mid y!2.y!3.y?u.y!.P(u) \quad) \\
& \downarrow \\
(\nu xy)(& x?b.x!(2+b).x?.0 \mid y!3.y?u.y!.P(u) \quad) \\
& \downarrow \\
(\nu xy)(& x!(2+3).x?.0 \mid y?u.y!.P(u) \quad) \\
& \downarrow \\
(\nu xy)(& x?.0 \mid y!.P(5) \quad) \\
& \downarrow \\
(\nu xy)(& 0 \mid P(5) \quad) \\
& \equiv \\
& P(5) \quad \text{assuming that } x \text{ and } y \text{ are not free in } P
\end{array}
$$

Runtime errors

Examples of the runtime errors that the type system will prevent:

$$(\nu xy)(x?z.P \mid y \lhd l.Q) \quad \times$$
$$(\nu xy)(x?z.P \mid y \rhd \{l\colon P\}) \quad \times$$
$$(\nu xy)(x?z.P \mid y!.0) \quad \times$$
$$(\nu xy)(x?z.P \mid y?.P) \quad \times$$
$$(\nu xy)(x?z.P \mid y?w.Q) \quad \times$$
$$(\nu xy)(x \rhd \{l\colon P\} \mid y \lhd k.0) \quad \times$$

Runtime errors

Deadlocks are arguably runtime errors but the basic session type system does not prevent them.

Deadlock due to cyclic dependency:

$$(\nu x_1 y_1)(\nu x_2 y_2)(x_1?z_1.x_2?z_2.x_1?.x_2?.0 \mid y_2!v_1.y_1!v_2.y_1!.y_2!.0)$$

Self-blocking:

$$(\nu xy)x?z.y!v.x?.y!.0$$

Runtime errors

The type system will eliminate processes in which a channel endpoint is not uniquely owned, for example:

$$(\nu xy)(x?z.P \mid y!a.Q \mid y!b.R) \qquad \times$$

These are not defined as runtime errors themselves, but they can evolve to runtime errors, for example:

$$(\nu xy)(\, x?z.\, x!2.P \mid \, y!3.\, y?u.Q \mid y!5.y?v.R)$$

$$\downarrow$$

$$(\nu xy)(\, x!2.\, P[3/z] \mid y?u.Q \mid \, y!5.\, y?v.R) \qquad \times$$

Type system: syntax of types

$$
\begin{array}{lll}
T ::= & & \\
 & \mathsf{end}_? & \text{end wait} \\
 & \mathsf{end}_! & \text{end close} \\
 & ?T.T & \text{input} \\
 & !T.T & \text{output} \\
 & \&\{l\colon T_l\}^{l \in L} & \text{external choice} \\
 & \oplus\{l\colon T_l\}^{l \in L} & \text{internal choice}
\end{array}
$$

It is possible (straightforward) to add data types and a data expression language, but this puts a lot of noise around the key ideas related to channels and session types.

Type system: inductive definition of duality

$$\mathsf{end}_? \perp \mathsf{end}_! \qquad \text{(D-End?)}$$

$$\mathsf{end}_! \perp \mathsf{end}_? \qquad \text{(D-End!)}$$

$$\frac{U \perp V}{?T.U \perp !T.V} \qquad \text{(D-?)}$$

$$\frac{U \perp V}{!T.U \perp ?T.V} \qquad \text{(D-!)}$$

$$\frac{T_l \perp U_l \quad (\forall l \in L)}{\&\{l\colon T_l\}^{l \in L} \perp \oplus\{l\colon U_l\}^{l \in L}} \qquad \text{(D-\&)}$$

$$\frac{T_l \perp U_l \quad (\forall l \in L)}{\oplus\{l\colon T_l\}^{l \in L} \perp \&\{l\colon U_l\}^{l \in L}} \qquad \text{(D-}\oplus\text{)}$$

Alternative: the unique dual of a type

$$\mathsf{end}_?{}^{\perp} = \mathsf{end}_!$$

$$\mathsf{end}_!{}^{\perp} = \mathsf{end}_?$$

$$(?T.U)^{\perp} = !T.U$$

$$(!T.U)^{\perp} = ?T.U$$

$$(\&\{l\colon T_l\}^{l\in L})^{\perp} = \oplus\{l\colon T_l\}^{l\in L}$$

$$(\oplus\{l\colon T_l\}^{l\in L})^{\perp} = \&\{l\colon T_l\}^{l\in L}$$

This definition works for non-recursive session types but becomes complicated to extend to recursive session types.

Typing judgements

Typing judgements are of the form

$$\Gamma \vdash P$$

where Γ is a map from variables to types, written $x_1 : T_1, \ldots, x_n : T_n$, and P is a process.

A process does not have a type — it is either typable or not.

Γ describes the resources (typed channels) needed by P.

Invariants of the type system

1. If $\Gamma \vdash P$ and $x : T \in \Gamma$ then the sequence of communication operations in T matches the sequence of communication operations in P.
2. If $\Gamma \vdash (\nu xy)P$ then the types of x and y are dual.
3. If $\Gamma \vdash P$ then in any subprocess of P of the form $Q \mid R$, any given variable occurs free in at most one of Q and R.
4. If $\Gamma \vdash P$ then Γ contains exactly the free variables of P.

Invariants 3 and 4 will be relaxed later.

Typing rules

The terminated process needs no resources.

$$\vdash 0 \qquad \text{(T-Inact)}$$

Typing rules

$$\frac{\Gamma \vdash P}{\Gamma, x : \mathsf{end}_? \vdash x?.P} \quad \text{(T-Wait)}$$

$$\frac{\Gamma \vdash P}{\Gamma, x : \mathsf{end}_! \vdash x!.P} \quad \text{(T-Close)}$$

All environments must be well-formed, so writing $\Gamma, x : \mathsf{end}_?$ means that $x \notin \Gamma$.

Example:

$$\dfrac{\dfrac{}{\vdash 0}\ \text{T-Inact}}{w : \mathsf{end}_! \vdash w!.0}\ \text{T-Close}$$

Typing rules

$$\frac{\Gamma, y : T, x : U \vdash P}{\Gamma, x : ?T.U \vdash x?y.P} \quad \text{(T-Recv)}$$

The bound variable y appears above the line but not below the line.

The type of x is different above and below the line, corresponding to the additional message in the process.

Example:

$$\frac{w : \mathsf{end}_!, x : \mathsf{end}_! \vdash w!.x!.0}{x : ?\mathsf{end}_!.\mathsf{end}_! \vdash x?w.w!.x!.0}\ \text{T-Recv}$$

Typing rules

$$\frac{\Gamma, x : U \vdash P}{\Gamma, x : !T.U, y : T \vdash x!y.P} \quad \text{(T-Send)}$$

The bound variable y appears below the line but not above the line.

Unique ownership: $x!y.P$ sends y to another process, therefore P cannot use y.

The type of x is different above and below the line, corresponding to the additional message in the process (like T-Recv).

Typing rules

$$\frac{\Gamma, x\colon T, y\colon U \vdash P \qquad T \perp U}{\Gamma \vdash (\nu xy)P} \qquad \text{(T-Res)}$$

The bound variables x and y appear above the line but not below the line.

Usually we expect x and y to be used in separate parallel processes within P, but this is not enforced by the rule (cf. the self-blocking example earlier).

Typing rules

$$\frac{\Gamma \vdash P \qquad \Delta \vdash Q}{\Gamma, \Delta \vdash P \mid Q} \quad \text{(T-Par)}$$

Writing Γ, Δ means that Γ and Δ have disjoint sets of variables.

This is where we enforce that a variable cannot be used in two parallel processes.

This rule is more difficult to read upwards (as an algorithm) because we don't know how to split the environment into Γ and Δ.

The basic idea for algorithmic typechecking with any kind of linear typing (unique ownership) is to use the fact that $\mathtt{dom}(\Gamma) = \mathtt{fv}(P)$.

$$\frac{\Gamma_1 \,;\, P \mapsto \Gamma_2 \qquad \Gamma_2 \,;\, Q \mapsto \Gamma_3}{\Gamma_1 \,;\, (P \mid Q) \mapsto \Gamma_3}$$

Example

The process

$$(\nu ab)(a!m.a!.0 \mid b?x.b?.x!.0)$$

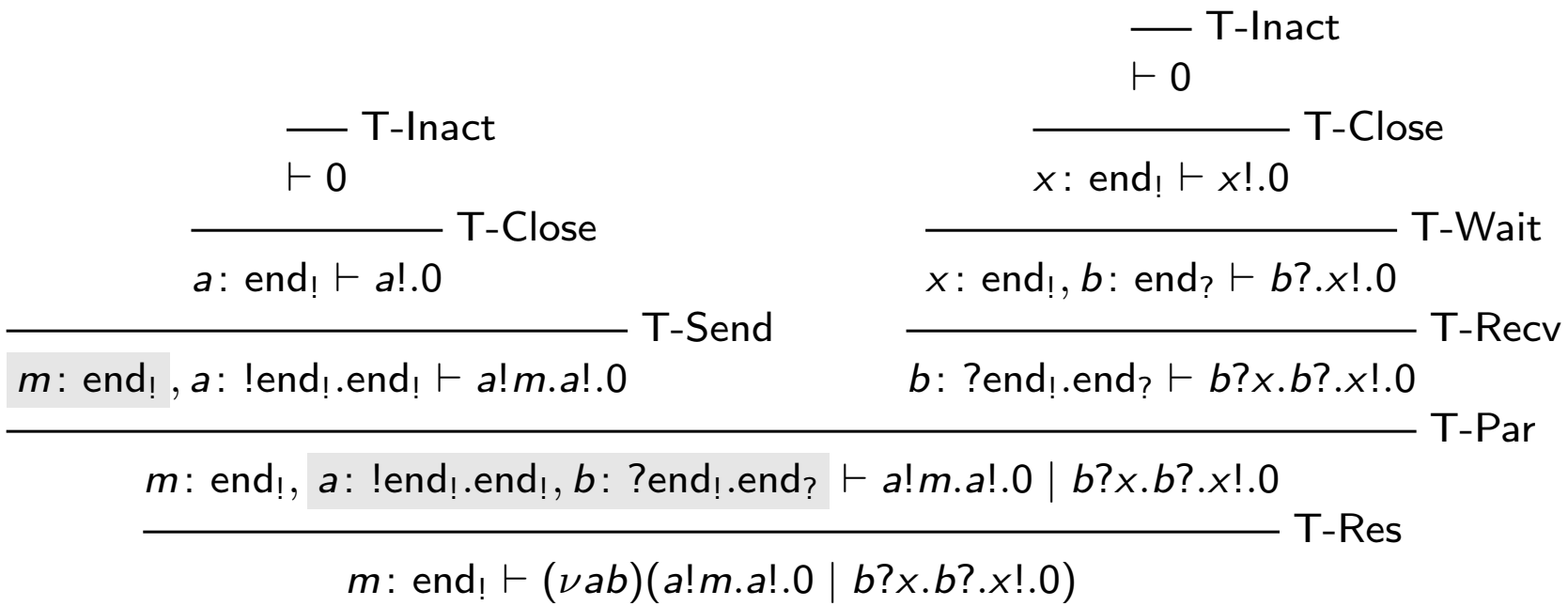
sends message m then closes a and b.

m is a channel which gets closed ($x!.$, as x becomes bound to m).

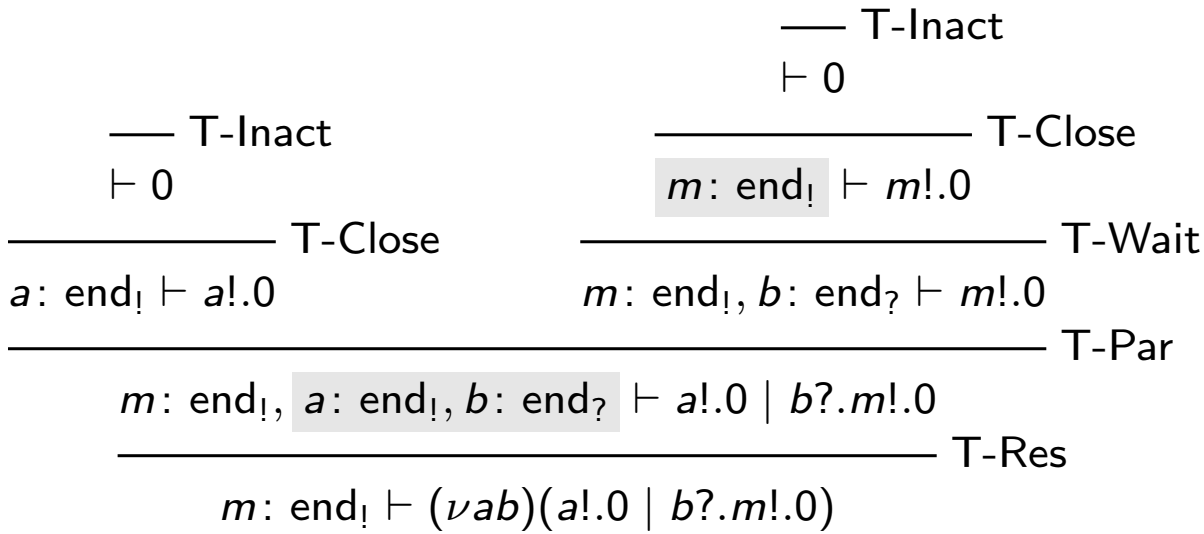
First reduction:

$$(\nu ab)(a!m.a!.0 \mid b?x.b?.x!.0) \rightarrow (\nu ab)(a!.0 \mid b?.m!.0)$$

Typing derivation for the initial process



Typing derivation for the next process



Typing rules (choice)

$$\frac{\Gamma, x\colon T_l \vdash P_l \qquad (\forall l \in L)}{\Gamma, x\colon \&\{l\colon T_l\}^{l\in L} \vdash x \rhd \{l\colon P_l\}^{l\in L}} \quad \text{(T-Bra)}$$

$$\frac{\Gamma, x\colon T_k \vdash P \qquad k \in L}{\Gamma, x\colon \oplus\{l\colon T_l\}^{l\in L} \vdash x \lhd k.P} \quad \text{(T-Sel)}$$

Very often in proofs, branch and select behave similarly to receive and send, but simpler.

The exact match between branches in the type and branches in the process (T-Bra) can be relaxed by subtyping.

In T-Sel, the requirement to know the whole type can be relaxed by subtyping.

Defining runtime errors (1): prefix processes

Prefix processes are processes of the form

$$x?y.P \qquad x!y.P \qquad x \triangleright \{l\colon P_l\}^{l \in L} \qquad x \triangleleft l.P \qquad x?.P \qquad x!.P$$

This formalises the notion of *thread*, namely a process that is initially sequential.

The *subject* of a prefix process is the channel endpoint that it is able to communicate on: x in the cases above.

Defining runtime errors (2): canonical forms

Canonical forms are processes of the form

$$(\nu x_1 y_1) \ldots (\nu x_n y_n)(P_1 \mid \ldots \mid P_m) \qquad (n, m \geq 0)$$

where each P_i is a prefix process. When $m = 0$, the part $P_1 \mid \ldots \mid P_m$ denotes the terminated process 0.

Lemma: Every process is structurally congruent to a process in canonical form.

Proof: Given a process, move restrictions outwards using C-Scope, taking advantage of α-identification to rename variables if necessary. The inner part of the process can then be organised as a parallel combination of processes that do not have parallel combination at the top level and are therefore either prefixes or 0. C-Neut can be used to remove 0 unless there are no prefixes, in which case a single 0 remains.

Defining runtime errors (3): redex

A process is a *redex* if it is structurally congruent to $(\nu xy)(P \mid Q)$ such that:

- P is $x?.P'$ and Q is $y!.Q'$, or
- P is $x?z.P'$ and Q is $y!w.Q'$, or
- P is $x \lhd k.P'$ and Q is $y \rhd \{l\colon Q_l\}^{l \in L}$ and $k \in L$.

A redex can achieve one step of successful communication.

Defining runtime errors (4)

A process is a *runtime error* if it is structurally congruent to a canonical form

$$(\nu x_1 y_1) \ldots (\nu x_n y_n)(P_1 \mid \ldots \mid P_m)$$

such that there are i, j, k with

$$\mathtt{subj}(P_i) = x_k, \qquad \mathtt{subj}(P_j) = y_k$$

and

$$(\nu x_k y_k)(P_i \mid P_j)$$

is not a redex.

A runtime error is a process in which endpoints x and y are bound together by ν and within the binding there are prefixes with subjects x and y but these prefixes cannot reduce by R-Close, R-Com or R-Choice because they do not have complementary natures.

Runtime errors

The previous examples of runtime errors are instances of the formal definition.

$$(\nu xy)(x?z.P \mid y \lhd l.Q) \quad \times$$
$$(\nu xy)(x?z.P \mid y \rhd \{l\colon P\}) \quad \times$$
$$(\nu xy)(x?z.P \mid y!.0) \quad \times$$
$$(\nu xy)(x?z.P \mid y?.P) \quad \times$$
$$(\nu xy)(x?z.P \mid y?w.Q) \quad \times$$
$$(\nu xy)(x \rhd \{l\colon P\} \mid y \lhd k.0) \quad \times$$

Runtime errors

The previous examples of deadlock do not satisfy the definition of runtime error.

Deadlock due to cyclic dependency:

$$(\nu x_1 y_1)(\nu x_2 y_2)(x_1?z_1.x_2?z_2.x_1?.x_2?.0 \mid y_2!v_1.y_1!v_2.y_1!.y_2!.0)$$

Self-blocking:

$$(\nu xy)x?z.y!v.x?.y!.0$$

Proving type safety

The strategy is to prove two results.

Absence of immediate errors If $\Gamma \vdash P$ then P is not a runtime error.

Preservation If $\Gamma \vdash P$ and $P \rightarrow Q$ then $\Gamma \vdash Q$.

Between them, these results guarantee that a typable process never produces a runtime error.

Because reduction involves structural congruence in rule R-Struct, we also need to prove

Preservation for structural congruence

If $P \equiv Q$ then $\Gamma \vdash P$ if and only if $\Gamma \vdash Q$.

Two lemmas

Lemma: If $\Gamma \vdash P$ then $\mathtt{dom}(\Gamma) = \mathtt{fv}(P)$.

Proof: By rule induction on $\Gamma \vdash P$.

Substitution lemma:
If $\Gamma, x : T \vdash P$ and $y \notin \Gamma$ then $\Gamma, y : T \vdash P[y/x]$.

Proof: Informally, simply replace x by y throughout the derivation of $\Gamma, x : T \vdash P$.
(Formally, define substitution properly and then use rule induction on the derivation of $\Gamma, x : T \vdash P$.)

Proving preservation for structural congruence

Lemma: If $P \equiv Q$ then $\Gamma \vdash P$ if and only if $\Gamma \vdash Q$.

Proof: By rule induction on the derivation of $P \equiv Q$.

For each case, consider the form that a derivation of $\Gamma \vdash P$ must have, and rearrange the components to form a derivation of $\Gamma \vdash Q$.

The congruence rules and the transitivity rule use the induction hypothesis.

Proving preservation

Theorem: If $\Gamma \vdash P$ and $P \rightarrow Q$ then $\Gamma \vdash Q$.

Proof: By rule induction on the derivation of $P \rightarrow Q$.

The inductive cases are straightforward.

The base cases are R-Com, R-Choice and R-Close. In each case, consider the form that a derivation of $\Gamma \vdash P$ must have, and rearrange the components to form a derivation of $\Gamma \vdash Q$.

The earlier example of

$$(\nu ab)(a!m.a!.0 \mid b?x.b?.x!.0) \rightarrow (\nu ab)(a!.0 \mid b?.m!.0)$$

illustrates the technique.

Proving absence of immediate errors

Theorem: If $\Gamma \vdash P$ then P is not a runtime error.

Proof: We prove the contrapositive: if P is a runtime error then we can't derive $\Gamma \vdash P$ for any Γ.

We can assume that P is in canonical form:

$$(\nu x_1 y_1) \ldots (\nu x_n y_n)(P_1 \mid \ldots \mid P_m).$$

If P is a runtime error then there are i, j, k, with $i \neq j$, such that $\mathtt{subj}(P_i) = x_k$, $\mathtt{subj}(P_j) = y_k$ and $(\nu x_k y_k)(P_i \mid P_j)$ is not a redex.

A derivation of $\Gamma \vdash P$ would have to contain

$$\Gamma, x_1 : T_1, y_1 : U_1, \ldots, x_n : T_n, y_n : U_n \vdash P_1 \mid \ldots \mid P_m$$

with $T_r \perp U_r$ for each r.

Proving absence of immediate errors

By further analysis of the derivation we have

$$\Delta_i, x_k : T_k \vdash P_i$$

and

$$\Delta_j, y_k : U_k \vdash P_j$$

for some Δ_i and Δ_j.

Because $\mathtt{subj}(P_i) = x_k$ and $\mathtt{subj}(P_j) = y_k$, the initial communication operations of P_i and P_j match the structure of T_k and U_k respectively.

Assuming that $(\nu x_k y_k)(P_i \mid P_j)$ is not a redex, these communication operations are not complementary and therefore T_k and U_k are not dual — so the derivation of $\Gamma \vdash P$ cannot be completed.

Extensions

This lecture has covered Chapter 2 of the book.

The session type system up to this point is very basic, and much less expressive than a typical paper in the literature.

We will now briefly summarise two extensions:

1. Infinite behaviour.

2. Sharable channels.

We will use these in the next lecture.

A third extension, subtyping, will be covered in the next lecture.

Infinite behaviour: types

The aim is to support repetitive protocols such as the maths server, either through equational recursive type definitions:

$$S = \&\{\mathsf{plus}\colon ?\mathsf{int}.?\mathsf{int}.!\mathsf{int}.S, \mathsf{neg}\colon ?\mathsf{int}.!\mathsf{int}.S, \mathsf{quit}\colon \mathsf{end}_?\}$$

or through recursive type expressions:

$$\mathsf{rec}\, S.\&\{\mathsf{plus}\colon ?\mathsf{int}.?\mathsf{int}.!\mathsf{int}.S, \mathsf{neg}\colon ?\mathsf{int}.!\mathsf{int}.S, \mathsf{quit}\colon \mathsf{end}_?\}$$

In the book we develop a coinductive theory of infinite types, then introduce finite syntax with rec.

Duality is defined coinductively, and we need type equivalence (also defined coinductively).

Infinite behaviour: processes

There are several established ways of supporting infinite process behaviour in pi calculus.

- Replication, defined by $!P \equiv P \mid !P$
- Replicated input, defined by $x\star?y.P \equiv x?y.P \mid x\star?y.P$
- Recursive process definitions, for example

$$\begin{aligned}\mathsf{mathsServer}(x\colon T) = x \rhd \{ & \\ & \mathsf{plus}\colon x?a.x?b.x!(a+b).\mathsf{mathsServer}(x), \\ & \mathsf{neg}\colon x?a.x!(-a).\mathsf{mathsServer}(x), \\ & \mathit{quit}\colon x?.0\}\end{aligned}$$

In the book we develop the theory with recursive process definitions, then for replicated input, then for recursive function definitions expressed by a fixpoint operator.

Sharable channels

The restriction to unique ownership of channel endpoints is significantly limiting.

Session type systems in the literature usually have a way for channels to be shared, if their type is suitable.

One way is to introduce a class of *unrestricted* types and design typing rules so that channels with unrestricted types can be shared between threads.

In the book we take a semantic approach, saying that a type is unrestricted if it is equivalent to the type that it becomes after one step of communication.

Unrestricted types

$$\frac{U \simeq ?T.U}{?T.U\ \mathrm{un}} \qquad \text{(U-?)}$$

$$\frac{U \simeq !T.U}{!T.U\ \mathrm{un}} \qquad \text{(U-!)}$$

$$\frac{T_l \simeq \&\{l\colon T_l\}^{l \in L} \qquad (\forall l \in L)}{\&\{l\colon T_l\}^{l \in L}\ \mathrm{un}} \qquad \text{(U-\&)}$$

$$\frac{T_l \simeq \oplus\{l\colon T_l\}^{l \in L} \qquad (\forall l \in L)}{\oplus\{l\colon T_l\}^{l \in L}\ \mathrm{un}} \qquad \text{(U-}\oplus\text{)}$$

A type is unrestricted if it is *stateless*, for example:

- $\mathrm{rec}\, X.!T.X$
- $\mathrm{rec}\, X.\&\{a\colon X, b\colon X\}$

Unrestricted types

We then add typing rules for *weakening* and *contraction*:

$$\frac{\Gamma \vdash P \qquad T\ \mathtt{un}}{\Gamma, x\colon T \vdash P} \qquad \text{(T-Weak)}$$

$$\frac{\Gamma, y\colon T, z\colon T \vdash P \qquad T\ \mathtt{un}}{\Gamma, x\colon T \vdash P[x/y][x/z]} \qquad \text{(T-Contr)}$$

This is a nice modular way of adding sharing to the type system.

Proofs become more complicated!

Our sponsors make OPLSS possible.

Diamond Sponsor

https://www.janestreet.com/

Platinum Sponsors

https://www.nsf.gov/

https://www.sigplan.org/

Silver Sponsors