Session Types and Mailbox Types — Simon Gay

Lecture 1 - *July 1, 2026*

1 Introduction

Session types and mailbox types are both examples of *behavioural types*. This phrase does not have a precise, formal definition; it is a heuristic way of describing type systems whose purpose goes beyond checking the shape of ordinary data. A behavioural type describes some aspect of how a program behaves. In session type and the rest of the lectures, the behaviour of interest is communication.

The central idea of session types is to express a communication protocol as a type. The protocols considered in this lecture are protocols on **bidirectional, point-to-point** channels. A channel has two endpoints, and the type at one endpoint describes the sequence of communication actions that the owner of that endpoint must perform. Static type checking can then verify the communication-oriented part of a program before it runs.

Mailbox types, which appear later in the course, address a different communication model: the actor model. In that setting each actor has a mailbox, many processes may send messages to it, and the receiver cannot generally rely on messages being delivered in a particular order.

2 Session Types as Protocols: Motivated Examples

A first example is the protocol "send one integer and then close the channel". Its session type is

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

The constructor $!\mathsf{int}.$ says that the next action is output of an integer; the dot indicates sequencing; and $\mathsf{end}_!$ says that this endpoint will close the channel. There is also a dual protocol: receive one integer and then wait for the other endpoint to close the channel,

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

The two forms of termination are both session types. The difference is not that one endpoint "has ended" and the other has not; rather, ending itself is a synchronising action. One endpoint performs the close, and the other endpoint waits for that close.

Compiled By:
Billy Bai, Henry Olson, Xusheng Zhi

Protocols may involve several messages and may change direction. For example,

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

describes an endpoint that sends two integers, receives one integer, and then closes.

The session type records the direction and type of each message, but not semantic relationships between data values. If a client sends two integers and expects a product or sum in return, the ordinary session type does not itself express that arithmetic specification.

Protocols can also include choice. A simple mathematical server might offer addition or negation:

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

The external choice constructor $\&\{\cdots\}$ means that the environment chooses a branch by sending a label. The server then follows the protocol attached to that label. The dual client type uses internal choice:

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

Here the client chooses the branch and sends the selected label to the server. In these examples the client closes the channel and the server waits, but that convention is not forced by the mathematics; the important point is that the two endpoints use dual termination actions.

Realistic services often need to run repeatedly. One can describe a recursive version of the mathematical server by the equation

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

The corresponding client type is obtained by dualizing the choice, directions, and termination:

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

The formal system developed in the main part of this lecture concentrates on finite behaviour, but recursive types are the natural extension needed for services of this form.

Messages need not be ordinary data. They may themselves be channels, in which case their message type is another session type. For instance,

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

means: send a channel endpoint whose future behaviour is to receive an integer and wait for closure, and then close the original channel. This passing of channels is called *delegation*. The owner of an endpoint can run part of a protocol and then transfer responsibility for the rest of that protocol to another process. Notice that dualizing the outer protocol does not dualize the type of the message being sent:

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

The receiver gets a channel endpoint with exactly the type that the sender sent.

3 Communication Model and Ownership

The communication model for this lecture is concurrent or distributed *point-to-point* communication. A process may use many channels, but each channel has **exactly** two endpoints.

Channels are *bidirectional*: **either** endpoint may send or receive, depending on the current point in the protocol. Communication is assumed to be reliable, and messages sent on a particular channel are received in the same order.

The message ordering in the same channel is preserved, which is what makes the sequential structure of session types meaningful. However, the orders of message in different channel remains independent.

The operational presentation in the lecture uses synchronous communication: a send and the corresponding receive happen together, and the sender blocks until the receiver is ready. Asynchronous versions of session types are also possible, and are closer to an implementation model based on sockets and buffers, but they require a more elaborate operational semantics. The main typing ideas are already visible in the synchronous setting.

A central discipline is *unique ownership* of endpoints. If two processes could freely share the same endpoint, their actions could interfere. For example, a type might say that a single integer is to be sent, but two threads sharing the endpoint might both try to send. The basic session type system prevents this by using techniques from linear type theory: a channel endpoint is a resource that must belong to one thread at a time. Delegation respects this discipline because sending an endpoint transfers ownership; the continuation of the sender may no longer use the endpoint it has sent.

The principal safety guarantee is absence of communication mismatch. If one endpoint sends a message, the other endpoint is ready to receive a message of the right type; conversely, if one endpoint is waiting to receive, the other endpoint is ready to send the expected kind of message. Another property guaranteed by the safety of session types is *session fidelity*: the sequence and types of messages observed at runtime match the session type assigned statically.

4 A Brief History of Session Types

The original line of work on session types begins with Kohei Honda's CONCUR 1993 paper, "Types for Dyadic Interaction", followed by papers with Kaku Takeuchi and Makoto Kubo at PARLE 1994 and with Vasco Vasconcelos and Makoto Kubo at ESOP 1998. These papers introduced the idea of typing structured communication protocols, first in process-calculus settings and then moving toward programming-language primitives.

Simon Gay (the lecturer) described his ESOP 1999 paper with Malcolm Hole, "Types and Subtypes for Client-Server Interactions", as the first session-types paper from outside the original group. That

work contributed subtyping, a careful distinction between the two endpoints of a channel, and the use of linear typing ideas to enforce unique ownership of channel endpoints. It was also apparently the first paper to use the phrase "session types" itself: earlier papers talked about sessions and types, but did not yet use the now-standard name.

Later developments include work on integrating session types with functional programming, multiparty session types for protocols with more than two participants, and the Curry–Howard correspondence between session types and linear logic. The reference for the first three lectures is the recent book by Simon Gay and Vasco Vasconcelos.

5 A Pi Calculus for Sessions

The lecture presents session typing for a small pi calculus. It is useful to start with an untyped process language, because then one can define runtime errors directly and prove that well-typed processes never reach them. For finite behaviour, the syntax of processes is

$$\begin{array}{rcll} P & ::= & \mathbf{0} & \text{inaction} \\ & | & x?.P & \text{wait for close} \\ & | & x!.P & \text{close} \\ & | & x?y.P & \text{receive} \\ & | & x!y.P & \text{send} \\ & | & P \mid P & \text{parallel composition} \\ & | & (\nu xy)P & \text{scope restriction.} \end{array}$$

The process $\mathbf{0}$ means the process has already terminated. The process $x?.P$ waits for the other endpoint of x to close and then continues as P; $x!.P$ closes x and then continues as P. The process $x?y.P$ receives a message on x, **binds** it to y, and continues as P. The process $x!y.P$ sends y on x and continues as P. Parallel composition $P \mid Q$ runs P and Q concurrently. Finally, $(\nu xy)P$ creates a private channel whose two endpoints are x and y, scoped over P. The double binder is deliberately non-standard: it makes the two endpoints explicit. There are also other ways to bind the two endpoints, such as $(\nu x^+x^-)P$. But it turns out the double binder is easier to work with for session type.

Processes are considered up to structural congruence. The main structural congruence rules used in the lecture are:

$$P \mid Q \equiv Q \mid P \qquad \text{(C-Comm)}$$
$$(P \mid Q) \mid M \equiv P \mid (Q \mid M) \qquad \text{(C-Assoc)}$$
$$P \mid \mathbf{0} \equiv P \qquad \text{(C-Neut)}$$
$$(\nu xy)(P \mid Q) \equiv (\nu xy)P \mid Q \qquad \text{(C-Scope)}$$
$$(\nu xy)P \equiv (\nu yx)P \qquad \text{(C-SwapC)}$$
$$(\nu xy)(\nu zw)P \equiv (\nu zw)(\nu xy)P \qquad \text{(C-SwapB)}$$

There are also congruence rules saying that structural congruence is preserved by process constructors, for example:

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

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

Parallel composition is commutative and associative, $\mathbf{0}$ is its unit, and restrictions may be reordered or moved outward when bound names are kept fresh. This lets reduction ignore irrelevant syntactic order and bring potential communication partners together.

The two main reduction rules are the close/wait interaction and message passing:

$$(\nu xy)(x!.P \mid y?.Q) \;\to\; (\nu xy)(P \mid Q)$$

and, schematically,

$$(\nu xy)(x!v.P \mid y?z.Q) \;\to\; (\nu xy)(P \mid Q[v/z]).$$

There are also contextual rules allowing reduction inside parallel composition and restrictions, and a structural rule allowing reduction after rearranging by structural congruence. Choice adds two more process forms:

$$x \rhd \{l_i \colon P_i\}_{i \in I} \qquad \text{and} \qquad x \lhd l.P.$$

The first offers an external choice of labelled branches; the second selects one of those labels. The reduction rule chooses the matching branch when the offered and selected labels agree.

As an example, a one-shot mathematical server can be connected to a client as follows:

$$\begin{aligned}(\nu xy)(\quad & x \rhd \{\mathsf{plus} \colon x?a.x?b.x!(a+b).x?.\mathbf{0}, \mathsf{neg} \colon x?a.x!(-a).x?.\mathbf{0}\} \\ & \mid y \lhd \mathsf{plus}.y!2.y!3.y?u.y!.P(u)).\end{aligned}$$

The client selects plus, sends 2 and 3, receives 5, closes the channel, and continues as $P(5)$, assuming the arithmetic expression is evaluated in the expected way.

6 Runtime Errors and Deadlock

The type system is intended to rule out communication mismatches. Typical bad configurations include a receive facing a select, a receive facing a close, two receives facing one another, or a branch offering one label while the other endpoint selects a different label:

$$\begin{aligned}\times \quad & (\nu xy)(x?z.P \mid y \lhd l.Q), \\ \times \quad & (\nu xy)(x?z.P \mid y!.\mathbf{0}), \\ \times \quad & (\nu xy)(x?z.P \mid y?w.Q), \\ \times \quad & (\nu xy)(x \rhd \{l \colon P\} \mid y \lhd k.\mathbf{0}) \quad (k \neq l).\end{aligned}$$

To define this formally, the lecture introduces *prefix processes*: processes whose first action is receive, send, branch, select, wait, or close. The *subject* of such a prefix is the endpoint on which the first action occurs.

Every process is structurally congruent to a *canonical form*

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

where the P_i are prefixes, or $\mathbf{0}$ when there are no prefixes. A *redex* is a pair of prefixes on opposite endpoints that can communicate: wait with close, receive with send, or selection with a branch containing the selected label. A runtime error is a canonical form containing prefixes on opposite endpoints of the same restricted channel which are not a redex.

There are other kind of processes, namely deadlock and self-blcoking, that do not fit the definition of runtime error, which is outside of safety guarantee of basic session type. Deadlock refers to the cyclic dependency such as

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

Self-blocking happens when the receive and send is sequenced in the same process

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

The safety theorem proved in basic session type is therefore not "well-typed programs always make progress" in the strongest sense. It is the more specific claim that well-typed programs do not reach the communication mismatches just described. There are also works by Wen Kokke et al. [1] to add deadlock into the behaves the session type want to avoid.

7 Types, Duality, and Typing Rules

For the finite calculus, the session type syntax is

T	::=	$\mathsf{end}_?$	wait for close
	|	$\mathsf{end}_!$	close
	|	$?T.U$	input a value of type T, then behave as U
	|	$!T.U$	output a value of type T, then behave as U
	|	$\&\{l_i\colon T_i\}_{i\in I}$	external choice
	|	$\oplus\{l_i\colon T_i\}_{i\in I}$	internal choice.

Similar to the duality of threads, duality also appears in session types. Duality relates the types that may safely appear at the two endpoints of a channel. Closing is dual to waiting, input is dual to output with the same message type, and external choice is dual to internal choice with matching labels and dual continuations. Equivalently, for non-recursive types:

$$\begin{array}{rclcrcl} \mathsf{end}_?{}^{\perp} & = & \mathsf{end}_!, & & \mathsf{end}_!{}^{\perp} & = & \mathsf{end}_?, \\ (?T.U)^{\perp} & = & !T.U, & & (!T.U)^{\perp} & = & ?T.U, \\ (\&\{l_i\colon T_i\}_{i\in I})^{\perp} & = & \oplus\{l_i\colon T_i{}^{\perp}\}_{i\in I}, & & (\oplus\{l_i\colon T_i\}_{i\in I})^{\perp} & = & \&\{l_i\colon T_i{}^{\perp}\}_{i\in I}. \end{array}$$

For recursive types, duality is usually better presented coinductively.

Typing judgements have the form

$$\Gamma \vdash P,$$

where Γ maps variables to session types. A process is not itself assigned a type; rather, it is checked against the resources it uses. The environment Γ contains exactly the free endpoints of P in the basic system.

References

[1] Wen Kokke and Ornela Dardha. Deadlock-free session types in linear Haskell. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell*, pages 1–13, 2021.