OREGON PROGRAMMING LANGUAGES SUMMER SCHOOL

Session Types and Mailbox Types — Simon Gay

Lecture 2 - *July 2, 2026*

1 Session Types in the Pi Calculus

The lecture begins by finishing the pi-calculus material from Lecture 1. This first part recalls how typing tracks protocol structure and rules out communication mismatch before the course moves to functional session types.

The typing judgment in Pi Calculus has the following form.

$$\Gamma \vdash P,$$

where P is not assigned a type of its own. Instead, the judgment says that P is well formed relative to the resources in Γ. These resources are the channel endpoints that P is allowed to use, together with their current session types. In the first system, contexts are exact: the domain of Γ is precisely the set of free variables of P, i.e. $FV(P) = dom(\Gamma)$.

This exactness is the mechanism behind several of the invariants of typed Pi Calculus:

- If $\Gamma \vdash P$ and $x\colon T$ appears in Γ, then the initial communication behaviour of P on x must match the initial constructor of T.
- If a process creates a private channel by $(\nu xy)P$, the two endpoint types used internally must be dual.

The rules for $\mathbf{0}$, $x?.P$, and $x!.P$ make the exact-resource discipline visible: The inactive process needs no resources. The process $x?.P$ needs x at type $\mathsf{end}_?$, plus exactly the resources needed by P; similarly $x!.P$ needs x at type $\mathsf{end}_!$, plus the resources needed by its continuation. Besides, linear ownership is enforced syntactically: in a parallel composition $P \mid Q$, a variable may occur in at most one side.

The type of an endpoint progresses through its session continuation under the typing rules of send, receive, wait, and close. Eventually it reaches one of the terminal session types, $\mathsf{end}_!$ or $\mathsf{end}_?$; in the

Compiled By:
Billy Bai, Henry Olson, Xusheng Zhi

functional language later, the corresponding close and wait operations return a value of type unit. For example, the receive process has the following form.

$$x?y.P$$

uses x first at an input type $?T.U$. After the receive has occurred, the continuation P uses the same endpoint x at the continuation type U, and it may also use the newly bound message variable y at type T. Thus the bound variable changes just as in a lambda-abstraction rule; the unusual part is that the type of x changes across the line.

The restriction rule is the central place where compatibility is checked.

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

To type $(\nu xy)P$, the body P is typed under assumptions $x\colon T$ and $y\colon U$, and the rule requires the duality $T \perp U$. Usually one expects x and y to be used by different threads inside P, but the rule does not itself require that. Consequently, this basic type system still admits some self-blocking processes: its safety theorem is about communication mismatch, not all possible forms of irreducible non-canonical forms.

The parallel rule is where disjoint ownership between threads is enforced.

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

The comma in the conclusion is defined only when the two contexts have disjoint domains. Read declaratively, this says that the resources used by $P \mid Q$ are the resources used by P and Q separately. Read algorithmically, it raises the usual problem for linear typing of deciding how the environment should be split. The standard implementation idea is to typecheck with "leftovers".

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

A checker is given a large input context, consumes the resources used by one subterm, and returns the unused part to be checked by the next subterm. At the top level one verifies that no linear resources remain.

2 Preservation in Pi Calculus

The following process is the important and also interesting example in the proof of preservation of session type over Pi Calculus.

$$(\nu ab)(a!m.a!.\mathbf{0} \mid b?x.b?.x!.\mathbf{0}),$$

where the free message m is itself a channel endpoint that will eventually be closed. The first communication step sends m from the a side to the b side.

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

The full derivation trees of both processes are shown in the slide of lecture one. The typing derivations before and after the reduction show the whole idea of preservation. Before the step, m occurs in the context of the sending thread $a!m.a!.\mathbf{0}$. After the step, it occurs in the context of the receiving thread $b?.m!.\mathbf{0}$, because ownership has moved with the message. At the same time, the bound endpoints a and b still have dual types, but those types are shorter: their first send/receive actions have been consumed. This is why preservation for processes is not the statement that every internal endpoint keeps the same type. The externally visible context is preserved, while the types hidden inside restrictions evolve in matching dual ways.

Choice behaves similarly to send and receive, but without a message value. Branching corresponds to external choice: the process offers a family of continuations indexed by labels. Selection corresponds to internal choice: the process selects one label and continues with the corresponding session continuation. In the basic rules, the labels in the process and type must match exactly. However, this may be stricter than what's actually needed. A branch process may safely implement extra branches that the type will never allow the other endpoint to choose, and a selection process should not necessarily need to know the entire menu of possible labels. These are typical motivations for adding subtyping in session type, which will be explained in the future lecture.

3 Runtime Errors and Type Safety

To state the safety theorem precisely, the runtime errors need a formal definition rather than relying on an informal notion of being stuck, which we've seen that is not completely protected by the basic session type. First, a prefix process is a process whose top-level form is a communication action: receive, send, branch, select, wait, or close. Its *subject* is the endpoint on which that first action communicates.

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 each P_i is a prefix process, except that the parallel part may be just $\mathbf{0}$ when there are no active prefixes. The proof is by rearranging the process using structural congruence: restrictions are pulled outward, alpha conversion avoids capture, and the remaining top-level parallel components are collected into one process.

A *redex* is a local configuration that can perform one successful communication step: wait with close, receive with send, or branch with a compatible select. A *runtime error* is then a canonical form containing two threads whose subjects are the two endpoints of the same restricted channel,

but whose actions do not form a redex. Such a pair is trying to communicate on endpoints that are linked together, but the operations are not complementary.

This definition deliberately excludes some deadlocks. A cyclic dependency may be stuck because each thread is waiting on a channel whose partner is not ready yet. A self-blocking process may use both endpoints in one sequential thread and therefore be unable to make progress. These are real problems, but they are not communication mismatches in the narrow sense being ruled out here.

The type safety proof has two main parts.

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

Preservation. If $\Gamma \vdash P$ and $P \to Q$, then $\Gamma \vdash Q$.

Because reduction is closed under structural congruence, one also proves that structural congruence preserves typability. Two simple lemmas are used repeatedly: 1) linearity lemma: if $\Gamma \vdash P$ then $\texttt{dom}(\Gamma) = \texttt{fv}(P)$, and 2) substitution lemma: variables can be consistently renamed in typing derivations.

Preservation is proved by induction on the reduction derivation. The contextual cases, such as reducing inside a restriction or in parallel with another process, are routine. The communication cases are the interesting ones. In each case, the typing derivation for the source process is inspected far enough to find the two communicating threads. The pieces are then reassembled into a typing derivation for the target process. The example above with m moving from the sender's context to the receiver's context is the prototype of this rearrangement.

Absence of immediate errors is proved by contradiction. Put a runtime error in canonical form. If it were typable, then before the final restriction rules were applied, each restricted pair x_k, y_k would have dual types. Looking at the two offending prefix processes, their first actions determine the first constructors of those two types. Since the pair is not a redex, those actions are not complementary; hence the types cannot be dual. This contradicts the restriction rule.

The first safety guarantee obtained this way is absence of communication mismatch. A second standard safety guarantee, *session fidelity*, says that the actual sequence of runtime messages follows the protocol described by the session type. A fully formal proof of fidelity would require more instrumentation: one would give labelled transitions for processes, recording transmitted values or labels, and labelled transitions for types, recording the corresponding protocol actions. Fidelity would then be a simulation result relating concrete runtime labels to their abstract type labels.

4 Recursive Behaviour and Shared Channels

The basic system so far is intentionally restricted. It only describes finite behaviour, and it enforces unique ownership for every endpoint. Two extensions are needed before moving to functional

programming examples.

First, repeated protocols require recursive types and repeated process behaviour. A recursive mathematical server can be described by an equation of the following form.

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

or sometime with an extra recursive type keyword $\mathsf{rec}\, S.\cdots$. In the session type textbook, it first develops a coinductive theory of infinite types and later adds finite recursive syntax. Duality and type equivalence must then account for unfolding recursive types.

Firstly, processes need the ability to loop, especially infinite loop, which is what a server usually does. there are several standard ways to express infinite behaviour in processes: replication $!P \equiv P \mid !P$, replicated input $x\star?y.P \equiv x?y.P \mid x\star?y.P$, recursive process definitions, and, in the functional language, recursive functions via a fixpoint operator. Replicated input is operationally attractive because the replicated process waits for an incoming message before a new copy is spawned.

Secondly, realistic programs need some form of sharing. Ordinary session types are linear because sharing a stateful protocol endpoint can scramble the intended sequence of actions. However, some communication types are *stateless*: after one communication step, the endpoint returns to an equivalent type. Examples include the following types.

$$\mathsf{rec}\, X.!T.X \qquad \text{and} \qquad \mathsf{rec}\, X.\&\{a\colon X, b\colon X\}.$$

Such types can safely be treated as unrestricted. This semantic notion of unrestrictedness is then connected to the usual structural rules of weakening and contraction: unrestricted resources may be discarded or duplicated, while linear resources may not.

5 Why Functional Session Types Need Linearity

After this pi-calculus recap, the lecture moves to a higher-order functional language with session-typed communication primitives. The main integration problem is that linear session endpoints interact with ordinary functional constructs.

Pairs are the first example. If c is a linear endpoint and we form the pair (c, v), then sharing the pair would also share c. Therefore the language uses a linear product type $T \times_1 U$. A full language would also want an unrestricted product type, but the lecture keeps the core system minimal.

Functions raise the same issue through closures. Suppose we have the following type.

```
f : !Int.end! → Int → unit
```

and we partially apply f to a linear channel c. The result is a closure waiting for the integer argument, but that closure contains c. If the closure were copied, then the endpoint would be

copied as well. Thus the language distinguishes linear functions $T \multimap U$ from unrestricted functions $T \to_\omega U$ (written in the slides as multiplicities 1 and ω). An unrestricted function can only be constructed when its closure contains no linear variables.

In the formal notation used below, these function types are written $T \to_1 U$ and $T \to_\omega U$; the slides write the same distinction as $\mathsf{T} \to_1 \mathsf{U}$ and $\mathsf{T} \to_\omega \mathsf{U}$.

There is a useful subtyping relationship.

$$T \to_\omega U <: T \to_1 U.$$

An unrestricted function may be used where a linear function is expected, because a function that can safely be used many times can certainly be used once. This avoids forcing programmers or type inference to guess too early whether a function will later be needed linearly.

6 The Bookshop Example

The running example is an online bookshop. On the shop side, the protocol offers two choices.

```
type Shop = &{add:?Book.Shop,
              checkout:?Card.?Address.end?}
```

The shopper has the following dual protocol.

```
type Shopper = ⊕{add:!Book.Shopper,
                 checkout:!Card.!Address.end!}
```

The recursive branch for add lets the shopper add any number of books before checking out.

The key programming idea is that communication operations return the channel endpoint with its evolved type. Sending a value on a channel returns that same channel.

$$\mathsf{send}\ v\ c \to c,$$

but the returned c has the continuation type. The type of send is curried.

$$\mathsf{send} : T \to_\omega\ !T.S \to_1 S.$$

The outer function is unrestricted because the primitive send has no linear data in its closure. The inner function is linear because the partially applied function send v may contain a linear value v.

Receiving returns both the received value and the evolved channel.

$$\mathsf{receive} : ?T.S \to_\omega T \times_1 S.$$

The product is linear because its second component is often a linear endpoint. Similarly, selecting a label returns the same channel with the branch continuation type. Matching on an external choice

is the dual operation: the matched branches are functions expecting the channel at the appropriate continuation type, and all branches must return a common result type.

In the code for mother and shop, the channel is explicitly threaded through the program by repeated rebinding. For example, after **select** add toShop, the name toShop is rebound to the same runtime endpoint but with the type that follows the add selection. This style is verbose, but it makes the type evolution explicit. The lecturer notes that a monadic interface can hide some of the plumbing.

The higher-order part of the example is the "gift voucher". The mother does not want to give her son free access to the shop channel, so she sends him a linear function of type Book $\to_1$ Book. The function closes over the shop channel and is capable of completing one purchase, choosing the son's book only if it is suitable and otherwise using the mother's default book. The function must be linear precisely because using it consumes the shop session hidden in its closure.

The full system uses **new** and **fork**. The operation **new** S creates a pair of linked endpoints of types S and $S^{\perp}$. The operation **fork** v starts a new thread by applying a linear function v : unit $\to_1$ unit to unit. The example creates one channel between shop and shopper, another between mother and son, forks the mother and son threads, and runs the shop in the original thread.

The shared-server variant illustrates unrestricted session types. A server channel can have the following type.

```
type ShopServer = ⋆!Shopper
```

abbreviates rec a.!$Shopper$.a. Each interaction on the shared server channel sends a fresh private endpoint to a client. The private endpoint is linear and supports one ordinary shop session; the server channel itself is stateless and can be used repeatedly.

Subtyping appears here only as a motivation when the shop adds a new remove option. Intuitively, an old client that only ever selects add or checkout should still be able to interact with a richer shop implementation. The formal subtype rules are left in the background at this point; the main idea is that subtyping relaxes the exact matching of branch and selection types.

7 Syntax and Semantics of the Functional Language

The formal language has two layers. The expression layer is a call-by-value functional language with variables, constants, abstractions, applications, pairs, pair splitting, channel creation, and match. The constants include unit, wait, close, receive, send, select, fork, and fix. Values include variables, constants, abstractions, pairs of values, and partially applied sends.

The process layer contains executing expressions.

$$P ::= \langle e \rangle \mid P \mid Q \mid (\nu xy)P.$$

There are no send or receive prefixes at the process layer, because those operations now live inside expressions. The restriction syntax is runtime syntax produced when **new** evaluates.

Expression reduction is standard small-step call-by-value reduction using evaluation contexts. Application performs beta-reduction on values, pair splitting substitutes the two components, and fix unfolds recursive functions. Process reduction then adds concurrency and communication.

The communication rules are the pi-calculus rules lifted into evaluation contexts. A thread whose next computation is wait x can synchronize with a thread whose next computation is close y when x and y are linked. The holes in both evaluation contexts are filled with unit. A receive on x synchronizes with a send of value v on y; the receiver gets the pair (v, x), while the sender gets back y. For choice, a match on x synchronizes with a selection on y, and the selected branch function is applied to x.

The rule for fork replaces the fork expression by unit in the current thread and creates a new thread running the forked function applied to unit. The rule for new S creates fresh linked endpoints and returns them as a pair inside a new restriction. These rules explain the relationship between the source-level functional operation **new** and the runtime pi-calculus-style restriction.

8 Typing and Safety for the Functional Language

The functional system has two judgments.

$$\Gamma \vdash e : T \qquad \text{and} \qquad \Gamma \vdash P.$$

Contexts now contain ordinary data types, function types, product types, and session types. Data types and unrestricted function types are unrestricted by declaration; linear products and linear functions are linear; session types may be linear or unrestricted depending on whether they are stateless.

The expression typing rules follow the usual structure, modified by linearity. The variable rule has only the variable being used in its context, since any additional linear assumptions would be silently discarded. Linear function introduction permits linear assumptions in the closure, while unrestricted function introduction requires the whole closure context to be unrestricted. Application combines disjoint contexts. Pair splitting is preferred over projection functions because it makes consumption of linear components explicit.

Weakening and contraction are available only for unrestricted types. The contraction rule is worth reading operationally. To type an expression that uses an unrestricted variable x twice, one first types a version using two fresh names, say y and z, and then contracts them back to x. The lecture illustrates this with a function that sends the same unrestricted endpoint twice on a channel. The expression contains two occurrences of the same variable, so the derivation must temporarily split them into two names before using contraction.

Process typing lifts expression typing to threads and reuses the familiar parallel and restriction rules. A thread is well typed when its expression has type unit. Parallel composition combines disjoint linear resources, and restriction requires dual endpoint types. Weakening and contraction also exist at the process layer for unrestricted resources.

The safety proof has the same shape as before: absence of immediate errors and preservation. Runtime errors are defined by putting a process into canonical form.

$$(\nu x_1 y_1) \cdots (\nu x_n y_n)(\langle e_1 \rangle \mid \cdots \mid \langle e_m \rangle)$$

and looking for two threads whose next communication subjects are linked endpoints but whose operations are not complementary. The subject of a thread is found by looking inside its evaluation context for a pending wait, close, receive, send, match, or select.

Preservation is more involved because it has to pass through the expression layer. The proof uses the standard call-by-value ingredients: substitution for expressions, typability of the subterm selected by an evaluation context, a typing judgment for evaluation contexts themselves, and a replacement lemma saying that a term of the right type can be put back into a typed context. These are the Wright–Felleisen-style proof components. Once expression preservation is established, process preservation follows by combining it with the communication cases and preservation under structural congruence.

The lecture ends by connecting the formal system to practical implementations. The key practical obstacle is support for linear or affine typing. As mainstream languages have gained richer type systems, implementing session types has become easier. FreeST [4] is a special-purpose functional language for session types, with polymorphic context-free session types and a rich higher-order type system. Library approaches include FuSE [3] for OCaml, Priority Sesh [2] for Haskell, and Ferrite [1] for Rust. The Haskell and Rust examples are especially natural now that Haskell has linear types and Rust has affine ownership built into the language.

References

[1] Ruo Fei Chen, Stephanie Balzer, and Bernardo Toninho. Ferrite: A Judgmental Embedding of Session Types in Rust (Artifact). *Dagstuhl Artifacts Series*, 8(2):14:1–14:2, 2022.

[2] 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.

[3] LUCA PADOVANI. A simple library implementation of binary sessions. *Journal of Functional Programming*, 27:e4, 2017.

[4] Peter Thiemann and Vasco T. Vasconcelos. Context-free session types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, page 462–475, New York, NY, USA, 2016. Association for Computing Machinery.