Session Types and Mailbox Types — Simon Gay

Lecture 4 - *July 4, 2026*

1 From Session Types to Mailbox Types

The first three lectures concentrated on session types, which are a typing discipline for channel-based communication. Lecture 4 changes model: the target is actor-style communication, where each process has a mailbox and other processes send messages to that mailbox by using the receiver's process identity. This is the communication model behind languages such as Erlang and Elixir, and it also motivates Go's slogan that one should not communicate by sharing memory, but share memory by communicating.

Communication-centric languages avoid many of the problems of shared-memory concurrency, but they still leave room for communication errors. The aim of mailbox types is to rule out such errors statically in the actor setting. The problem is different from ordinary session typing because actor communication is unordered and many-to-one. Any process that knows a process identifier can send a message to the associated mailbox, and messages from different senders may arrive in any order.

This explains the tradeoff stated in the lecture. Channels are relatively easy to type but hard to distribute. Actors are easy to distribute but hard to type. Channel mobility is especially awkward in a distributed implementation because sending a channel endpoint transfers not only a name, but also a bidirectional communication structure with buffers and ownership information. By contrast, actor communication only needs the sender to know the receiver's identity and place a message in the receiver's mailbox.

Previous attempts to combine session types with actors often used multiparty session types. Multiparty session types begin with a global architecture saying which participants exist and how they communicate, and then project local behaviours from that global description. That architecture is useful, but it is not completely faithful to the actor model, where the communication topology is less rigid and anyone with the right identity may send. The mailbox-type work discussed in this lecture starts from the actor model itself rather than forcing actors into a session discipline.

The technical origin is the ECOP 2018 paper on mailbox types for unordered interactions by de'Liguoro and Padovani. Gay, Fowler, and collaborators then asked how to move the idea from a

Compiled By:
Billy Bai, Henry Olson, Xusheng Zhi

process calculus presentation into a more ordinary programming-language setting, especially with Erlang in mind. The lecture describes the result through PAT, a functional language with first-class mailboxes and mailbox types. PAT is not yet Erlang: in Erlang a process has exactly one mailbox permanently attached to it, whereas PAT lets mailboxes be values that can be created and passed around. That gap is important, because first-class mailboxes are convenient for a functional core language but not the literal programming model of Erlang.

2 The Future Example and Its Errors

The running example is a future, meaning a placeholder for a value that can be written once and then read many times. In the lecture's example, the value is an integer. The future starts empty, accepts one *put* message containing the integer, and then becomes full. Once full, it accepts any number of *get* messages. Each *get* message carries the mailbox or process identity to which the answer should be sent, and the future replies with a *reply* message containing the stored integer.

The important point is not the data structure itself, but the kinds of mistakes that actor programs can make. If a client sends two *put* messages, the second write violates the intended write-once protocol. If no *put* message is ever sent, then *get* requests may accumulate, but they cannot be answered because the future never obtains a value. If the full future receives a *get* request but the implementation forgets to send the reply, then the client that waits for *reply* will block forever.

There are also errors that are less protocol-like but just as practical. Sending a message with an unexpected tag, such as *surprise*, leaves a message in the mailbox that no receive branch will ever handle. This might be a typo in the tag rather than a deliberately new protocol action. If this happens repeatedly inside a loop, the mailbox may grow without bound.

Payload types also matter. If a client sends *put* with a string payload but later uses the returned value as an integer, the program has a data mismatch even though the tag-level protocol looked plausible. Finally, the lecture highlighted a self-deadlock: if the client waits for *reply* before it has sent the *get* message that would cause the reply, the computation blocks on an event that it has prevented itself from initiating. The mailbox type system is designed to catch all of these examples statically.

3 Mailbox Patterns

A mailbox type is a capability paired with a pattern. The capability says whether a reference is used for sending or receiving. The pattern describes either the messages that must be sent to a mailbox, or the messages that may be present when receiving from it. This is a resource-control discipline, but it is not ordinary linear typing. For a given mailbox there may be many sending references, but there is only one receiving reference.

The pattern language describes unordered mailbox contents. An atomic pattern is a message tag such as *put* or *get*, with payload types handled separately. The product-like operator, written here as $\odot$, combines patterns:

$$E \odot F$$

means that the mailbox contains messages matching E and messages matching F. The operator is commutative, because the mailbox contents are unordered. This is one of the places where mailbox types differ sharply from session types: the written order of a pattern is not a temporal order.

There is also a choice pattern $E + F$, meaning that the mailbox contents match either E or F. The pattern E^* means zero or more copies of E. The pattern 1 is the empty mailbox. The pattern 0 is the unreliable mailbox, representing a state that the type system should prevent. The safety story can be read as saying that a well-typed program never evolves to a configuration whose mailbox information collapses to 0.

For the future example, the empty future has a receiving capability for a pattern of the following shape.

$$?(put \odot \bigstar get)$$

This says that at most one *put* message is expected, while any number of *get* messages may be present. It does not say that the *put* message arrives before the *get* messages. Indeed, a *get* can arrive early and wait in the mailbox until the future receives the *put* and becomes full. This is why mailbox types require a kind of lookahead: the type of the empty future must anticipate messages that will only be handled after the state transition to the full future.

The full future has a receiving capability for $\bigstar get$. Each *get* message carries a return address. Typing that payload creates a sending obligation for the future: if it receives a *get* containing a mailbox reference, it must send a *reply* to that mailbox. On the client side, after sending *put* and *get*, the client has a receiving capability for *reply*. The type system checks that the overall obligations balance, so that sends promised by one part of the program match receives expected by another.

4 PAT as a Programming Language

The original mailbox-types paper was formulated in a pi-calculus style. A process calculus describes a snapshot of a concurrent system: names have scopes, components run in parallel, and a whole configuration is visible at once. A programming language has a different shape. The programmer writes a program containing constructs such as let, spawn, and function calls, and the runtime later creates fresh mailboxes, starts threads, and moves to configurations that look more like the process-calculus snapshots.

PAT was introduced to bridge this gap. It is a functional programming language with actor communication and mailbox types. Its name points both to patterns and to the British children's character

Postman Pat, a joke that is also conceptually apt because the language is about typed delivery to mailboxes.

In PAT, the future implementation is written as functions parameterized by a mailbox reference, often called `self`. The empty future function guards on its mailbox and receives a *put* message. Once it receives the integer payload, it calls the full future function with that value. The full future guards on *get* messages, sends *reply* to the requester, and recurs as the same full future because the value remains available.

The guard construct is the PAT form that examines a mailbox. If an appropriate message is present, the corresponding receive branch runs. If no matching message is present, the guard blocks. A guard may have several receive clauses for different tags. There is also a free guard, which succeeds when the mailbox is empty and no further messages can arrive. Implementing that operation requires runtime information such as reference counts for outstanding send capabilities, because the runtime must know that there are no remaining senders that could later place a message in the mailbox.

The client code creates a future mailbox, spawns the empty future with that mailbox, creates its own mailbox for the reply, sends *put* to the future, sends *get* with the client's mailbox as the return address, then waits for *reply*. The order of the sends can be more flexible than a session-typed reader might expect. A client may send *get* before *put*, provided the program still ensures that the *put* eventually happens before the client waits in a way that causes self-deadlock. The unordered mailbox semantics permits the message to wait; the type system must still rule out impossible or unbalanced behaviours.

The live demo in the lecture illustrated the error reporting for an extra *put*. With two *put* messages and one *get*, the type checker inferred a send pattern like $put \odot put \odot get$ for the client behaviour. The declared future mailbox type allowed only $put \odot \bigstar get$. The type checker therefore produced a failed inclusion constraint, saying essentially that the inferred pattern is not included in the declared pattern. The current implementation catches the bug, but its diagnostic quality is limited because the generated constraint error does not yet point cleanly to a source location.

5 Name Hygiene and Quasi-Linearity

Moving from a process calculus to a programming language creates several name-hygiene problems. The type system must prevent use-after-free errors: if a mailbox has been freed, a later send to the same mailbox should not type check. This becomes difficult because the actor discipline allows many send references but only one receiving reference. Strict linear typing would make the ownership story simpler, but it would also be too restrictive for the intended many-writer, single-reader model.

Aliasing is the main complication. If a variable a is bound to the same mailbox as x, then freeing through a also invalidates later uses of x. The alias may be introduced directly by a let binding, indirectly through an evaluation context, or dynamically through communication when a mailbox reference is sent as a payload. At runtime, two names that look distinct in the source may denote

the same mailbox. The type system therefore needs a disciplined way to allow aliases while still controlling the final use of the resource.

The solution used in PAT is quasi-linear typing, following ideas from Kobayashi and later work on linear types for packet processing. Each reference to a resource is classified as either returnable, also called first class in the lecture, or second class. A reference may be used many times as a second-class reference within a thread. It may be used once as a returnable reference, and that use must be the last occurrence of the name in that thread.

This is how PAT supports many ordinary uses without giving up control of the receiving capability. Sending a mailbox as a payload can use a second-class reference. Guarding on a mailbox or freeing it requires the returnable reference. In the future client, the occurrence of `self` sent as the return address in the *get* message is second class, while the later guard or free on `self` is returnable. The program is accepted because the returnable use is the final use.

Sequential composition of typing contexts enforces this ordering. The type system uses a partial operator, written in the slides as a small right triangle, to combine usage annotations from earlier and later parts of a program. A second-class use followed by another second-class use remains second class. A second-class use followed by a returnable use becomes returnable. A returnable use followed by a later second-class use is undefined, so the typing rule fails. This is the formal mechanism behind the slogan that the returnable occurrence must be last.

At the same time, the type system must combine mailbox capabilities. Two send capabilities for the same mailbox combine by multiplying their patterns.

$$!E \boxplus !F = !(E \odot F)$$

A send capability can cancel against a matching receive capability, leaving the remaining receive pattern. For example, if the global receiver expects $E \boxplus F$ and one part of the program supplies the send obligation E, then the residual receiver must still account for F. In a complete program, the sends and receives should balance all the way down to a receiving capability for 1, the empty mailbox.

6 Typing Rules and Soundness

The typing rules are declarative, but the lecture emphasized their computational content. The rule for new creates a fresh mailbox with receiving capability for the empty pattern 1. The rule for spawn converts the environment of the spawned thread to second-class usages, because quasi-linearity is checked separately within each thread. The rule for send requires a second-class reference to the destination mailbox and checks that the payloads match the declared payload types for the message tag. It then records the appropriate send obligation.

The rule for let explains what "followed by" means in the quasi-linear usage operator. In a term of the form let $x = M$ in N, the computation evaluates M first and then N. The typing contexts for

the two subterms are therefore combined sequentially. If a returnable use of a mailbox appears in M and a second-class use appears in N, the sequential combination is not defined, so the program is rejected.

The lecture gave a small abstract derivation for a program that creates a mailbox, sends a message to itself, receives that message, and then frees the mailbox. In that derivation, the usage annotations and the mailbox patterns are combined at the same time. The send and receive patterns cancel so that the mailbox returns to the empty pattern. The second-class occurrence used for sending is followed by the returnable occurrence used for guarding and freeing, which is allowed.

The metatheory has the expected shape, but the proof is technically involved. Type preservation says that if a runtime configuration is well typed and takes a reduction step, the resulting configuration is also well typed. The preservation theorem is stated under the assumption that the surrounding typing context is not unreliable. There is also a safety result for the explicit runtime failure term: a correctly typed system does not reduce to fail. Intuitively, well-typed programs do not reach the unreliable mailbox pattern 0.

7 Algorithmic Type Checking

The declarative rules are not directly an implementation. They contain context-combination operations whose inputs are not obvious when read bottom-up as a type checker. They also use subtyping and pattern inclusion, so the checker has to decide when a pattern is included in another and when subtyping should be applied. The implementation therefore uses constraint generation and later constraint solving.

The main technique is co-contextual typing. Instead of giving the checker a context and asking it to type a term, the checker is given a term and returns the context required to type it, together with the type and a set of constraints. This direction is helpful when the context is exactly what needs to be inferred from the term's uses of mailboxes and variables.

PAT also uses bidirectional typing. In synthesis mode, the checker is given a term and produces its type. In checking mode, the checker is given both a term and an expected type, and it checks that the term has that type. The implementation combines the two ideas into backward bidirectional typing. Given a term, the checker returns required environments and pattern-inclusion constraints, using checking mode where possible and synthesis mode where necessary.

Pattern variables appear during this process. For instance, when combining a send and a receive capability whose exact residual pattern is not yet known, the checker may introduce a fresh pattern variable and generate a constraint involving it. The constraints from different subterms are accumulated and solved later. The soundness and completeness theorem for the algorithm says that successful type checking implies typability in the declarative system, and every declaratively typable program is accepted by the algorithm.

The implementation pipeline uses external solving machinery. A key step relies on results by Hopkins and Dexter for transforming or solving pattern-inclusion constraints. After that transformation, residual constraints can be sent to Z3. This is an example of a type checker whose practical implementation depends on an external solver rather than a hand-written domain-specific constraint solver. The lecture noted that, as far as the speaker knew, type checking remains decidable in this system.

8 Case Studies, Erlang, and Open Problems

The paper includes case studies beyond the future. One industrially motivated example concerns robots in a factory that coordinate access to a collection point so that only one robot at a time enters the critical area. The lecture presented this as a scenario that mailbox types describe more naturally than session types, because the communication is actor-like and not organized around a fixed session channel topology.

The implementation was also tested on examples from the Savina collection of concurrent and distributed benchmarks, as well as examples from the original mailbox-types paper. There was no directly comparable checker, because PAT was the first functional language with mailbox types, but the reported running times suggested that constraint generation and external solving did not impose a large overhead on the example programs. Some benchmarks could be checked in a stricter mode that gives a stronger correctness property, while others required the less strict variant.

The central slogan of the lecture is that mailbox types type the mailbox, not the process. The type describes the allowed contents of the mailbox and the obligations associated with references to it. The process code is then checked against those mailbox descriptions so that sends, receives, payload types, and final mailbox states line up. This is different from session types, where the type is usually attached to an endpoint and describes a temporal protocol along that endpoint.

Several pieces of future work remain. More type inference would reduce the number of annotations that PAT programmers must write. At the time of the ICFP paper, PAT had a type checker but not a full executor; the lecture noted that an interpreter now exists, so PAT programs can be run. The larger remaining challenge is the bridge to Erlang. PAT's first-class mailboxes are useful, and in some examples an extra mailbox can act almost like a data structure. Erlang does not offer that model directly, although one can sometimes simulate it by spawning a process whose mailbox plays that role.

The Q&A clarified how one might handle actors with several independent tasks. In PAT, using separate mailboxes can give smaller and more precise types, because each mailbox has its own pattern. In Erlang, a process has one physical mailbox, so the ongoing idea is to use interfaces: an interface is a set of message tags representing one logical view of the mailbox. Different interfaces could let the type system approximate multiple logical mailboxes implemented by one Erlang mailbox.

The final question asked whether constraint solving produces minimal or canonical patterns. The

answer was cautious, but the key point was that the implementation is looking for a complete solution in which everything balances to the empty mailbox. An unsolved pattern variable would correspond to an unbalanced obligation, so it would be treated as an error rather than left as a residual principal type. That answer fits the whole lecture: mailbox types are not merely descriptive summaries of possible traffic, but a static accounting discipline for making actor communication reliable.