An Introduction to Mailbox Types

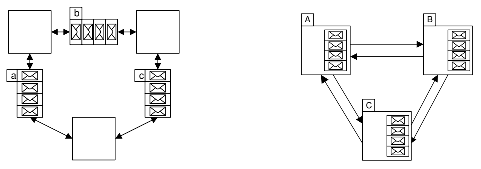
Simon Fowler, Simon J. Gay, and Luca Padovani

Abstract Communication-centric programming languages such as Go, Erlang, and Concurrent ML sidestep many of the issues that arise in shared-memory concurrency by supporting lightweight processes that communicate using explicit message passing. Behavioural type systems provide a lightweight way of statically detecting behavioural errors; among them, session types serve as a compelling behavioural type discipline for ruling out communication errors in channel-based languages like Go. Actor-based languages such as Erlang are widely used for writing reliable distributed systems but have a significantly different communication model to channel-based languages. Mailbox types are a recent behavioural type system that allows a user to capture the many-to-one communication patterns inherent in actor languages and to rule out several classes of communication errors such as protocol violations and deadlocks. This chapter begins by introducing mailbox types by example, before formally defining the semantics of mailbox types and using the semantics to define notions of subtyping and equivalence. We further describe the *Mailbox Calculus*, which integrates mailbox types in an extension of the asynchronous π-calculus, and *Pat*, which is a core mailbox-typed functional programming language.

Simon Fowler
School of Computing Science, University of Glasgow, UK e-mail: Simon.Fowler@glasgow.ac.uk

Simon J. Gay
School of Computing Science, University of Glasgow, UK e-mail: Simon.Gay@glasgow.ac.uk

Luca Padovani
Department of Computer Science and Engineering, University of Bologna, Italy e-mail: luca.padovani2@unibo.it



(a) Channels (b) Actors

Fig. 1: Channel- and actor-based communication [9]

1 Introduction

Concurrency and distribution are key components of modern computing. Writing programs using *shared memory concurrency* requires complex reasoning and synchronisation to avoid unsafe data races, and improper use of concurrency control mechanisms such as semaphores and mutexes can lead to whole-program deadlocks. Worse, locks are non-compositional and make modular reasoning difficult [18].

Communication-centric languages such as Go, Erlang and Elixir, instead advocate a style of programming in which lightweight processes communicate using explicit message passing instead of by sharing memory. Although programming using message-passing concurrency eases much of the mental burden of shared-memory concurrency, programming errors can lead to other issues such as communication mismatches and protocol deadlocks.

In general, programming language support for *correct* communication has lagged far behind support for correct sequential computation. Many contemporary programming languages have sophisticated static data type systems, typified within the mainstream by Haskell and Scala. Other languages, for example Python, ensure runtime safety by comprehensive dynamic typechecking. The situation is not the same for communication and concurrency. Whether dynamic or static, typechecking in standard communication oriented-languages and frameworks such as Go, Erlang, Elixir, Scala/Akka (and its extensions TAkka and Akka Typed) is limited to the data types of messages. Typically, neither the *properties* of communication protocols (for example deadlock-freedom or absence of races), nor the *conformance of programs* to these protocols, is checked as part of the development process.

1.1 Behavioural Type Systems: Channels vs. Actors

Behavioural type systems [2] go beyond checking of data types in a program in order to enforce *behavioural* properties of a system, for example ensuring that a file handle is always closed after it has been opened. An established class of behavioural type systems (theoretically, but not yet adopted in mainstream languages) is *session types* [12], which allow complex protocols to be specified as types and their implementations checked statically.

Session types work well for languages in which communication is based on point-to-point communication channels (Figure 1a), such as TCP/IP connections or the channels of Go. It is, however, more difficult to apply session types to languages such as Erlang, Elixir and Scala/Akka that are instead based on the actor model [1, 15, 21]. In actor languages, lightweight processes communicate via *mailboxes* (Figure 1b) that each have a single reader (the owning actor) and many writers (potentially any other actor in the system). This model does not follow the basic assumptions underlying session types, namely that a communication channel has one sender and one receiver and preserves the order of messages.

Actor languages and frameworks are widely used in practice due to their support for scalable and resilient distributed systems. A key strength is their support for design patterns such as *supervision hierarchies* that allow parts of a system to be restarted after a failure [3]. Session types can be applied to actor systems by using effect systems to restrict the communication actions that a process performs [8, 11, 14]. While session types are useful for actor programs that have *structured* interactions, programming with session types requires developers to rewrite their applications to use a particular programming style. The rigidity of session types when applied to actor languages therefore poses a challenge for the application of behavioural type systems to concurrent and distributed systems.

1.2 Mailbox Types

A recent alternative to session types, designed for actor languages, is *mailbox types*. A mailbox type is an invariant over a mailbox, specifying how many messages of each kind (classified by tags) are allowed to be in the mailbox—but crucially, and in contrast to session types, there is no information about the order in which messages enter the mailbox. It is possible to check at compile-time that mailbox types are respected by the send and receive operations in a program, and therefore statically eliminate a range of errors.

This chapter introduces the theory and practice of mailbox types. We follow the technical development from the literature, in which mailbox types were first introduced for a process calculus, the *Mailbox Calculus* [6], and later reformulated for a concurrent functional programming language called *Pat* [10]. Both models are a *generalisation* of actor languages and allow multiple first-class mailboxes as opposed to an implicit, monolithic mailbox. Each calculus has a typechecking tool that can be used to experiment with the examples given throughout the chapter.

The rest of the chapter is structured as follows. Section 2 begins by introducing two motivating examples—a future and a lock—and describing their mailbox types. Section 3 introduces mailbox types formally and describes a semantics for *mailbox patterns*. Section 4 introduces the Mailbox Calculus, a small mailbox-typed process calculus based on the asynchronous π-calculus. Section 5 moves from the setting of a process calculus to the setting of a mailbox-typed first-order concurrent functional language, Pat. Section 6 describes related work and complementary approaches, and Section 7 concludes.

Learning Objectives

After completing this chapter, you will be able to:

- Describe invariants on mailboxes using mailbox types, and understand how mailbox types can rule out common communication errors.
- Determine the semantics of a mailbox pattern, and write mailbox patterns from informal specifications.
- Understand the challenges of integrating mailbox types in a programming language, and reason about safe aliasing in mailbox-typed programs using quasilinearity.
- Write simple mailbox-typed programs in the Mailbox Calculus and the Pat programming language.

2 Mailbox Types by Example

In this section we discuss three motivating examples, written using the Elixir actor programming language [19]. Elixir is a communication-centric programming language supporting lightweight, addressable processes and message-passing concurrency. Even the simple programs that follow can contain subtle, hard-to-diagnose errors. We will then see how we can use mailbox types to codify the interactions of each example.

2.1 Future Example

A *future* is a placeholder variable for the result of an asynchronous computation. A future begins in an “empty” state. After an empty future is initialised with a value, all existing and future requests will return the given value. Importantly, futures are *write-once*: multiple write operations are not supported and will result in an error.

Although futures are often used in shared memory settings, they are also useful for describing the results of remote calls in actor languages (e.g., the `rpc:async_call/4` function in Erlang[1]) and are readily encodable using message-passing concurrency. We can describe the behaviour of a future in Elixir with the following functions:

[1] https://www.erlang.org/doc/apps/kernel/rpc.html#async_call/4

```
def emptyFuture() do
  receive do
    { :resolve, val } ->
      fullFuture(val)
  end
end
```

```
def fullFuture(val) do
  receive do
    { :get, replyTo } ->
      send(replyTo,
           { :reply, val })
      fullFuture(val)
    { :resolve, _ } ->
      raise "Invalid message!"
  end
end
```

A future begins by evaluating the `emptyFuture` function, and waits for a message of the form `{ :resolve, val }`, where `:resolve` is an Elixir *atom* that serves as a message tag and `val` is the value to store in the future. It is perfectly acceptable for other processes to send `:get` messages to an empty future, but they will not be processed until the future has been resolved. After receiving the `:resolve` message, the future will call the recursive `fullFuture` function. In this state, the process will handle messages of the form `{ :get, replyTo }`, where `replyTo` is the ID of the process making the request. To handle the request the future will send its value to `replyTo` and recursively process any remaining messages. Any further `:resolve` messages are invalid and will result in the future terminating with an exception; note that the `:resolve` clause in `fullFuture` is *defensive code* required to ensure that the system invariant is upheld.

We can write a client and a main function as follows:

```
def client(futurePid) do
 send(futurePid,
  { :get, self() })
 receive do
  { :reply, reply } ->
   IO.puts("#{reply + 1}")
 end
end
```

```
def main() do
 futurePid =
  spawn(fn -> emptyFuture() end)
 send(futurePid, { :resolve, 5 })
 spawn(fn -> client(futurePid) end)
 spawn(fn -> client(futurePid) end)
end
```

The `main` function creates a new future by spawning a process that evaluates the `emptyFuture` function, along with two clients. A client interacts by sending a `:get` message along with its process ID to the future, before waiting for a `:reply` message. The program would print out `6` twice to the console.

Even in this simple example, it is possible to make several communication errors, including but not limited to:

- Sending a `:resolve` message twice will result in a runtime error. Not sending a `:resolve` message at all will result in all clients of the future blocking indefinitely.
- Similarly, forgetting to send a `:reply` message after receiving a `:get` message will result in all clients blocking indefinitely.
- At present the client attempts to add 1 to the result of the request, meaning that resolving the future to an incompatible data type (e.g., a string or Boolean) would result in the client throwing a runtime error.
- Misspelling a message label, or sending an unsupported message, will not raise an error but will instead cause the message to silently be stored and not processed. This results in *junk*, a useless message that occupies memory.

Mailbox types describe the contents of a mailbox, or the messages that must be sent to a mailbox, at a given point in time. We can describe the mailbox types for the Future example as follows:

$$\begin{aligned} \mathsf{EmptyFuture} &\triangleq ?(\mathtt{Resolve}[\mathtt{int}] \cdot \mathtt{Get}[\mathsf{ClientReply}]^*) \\ \mathsf{FullFuture} &\triangleq ?(\mathtt{Get}[\mathsf{ClientReply}]^*) \\ \mathsf{ClientReply} &\triangleq !\mathtt{Reply}[\mathtt{int}] \end{aligned}$$

A mailbox type $?E$ or $!E$ consists of a *capability* ($?$ or $!$) and a *pattern* E. A mailbox type with *input* capability $?$ describes the messages that a mailbox can contain (and therefore must be consumed), whereas a mailbox type with *output* capability $!$ describes the messages that must be sent to the mailbox. A *pattern* is a *commutative regular expression* that describes the configuration of messages in the mailbox. Going back to our Future example:

- The EmptyFuture type is an input mailbox type that describes the messages receivable by an empty future. In an empty state, the future expects to receive *a single* `Resolve` message (with a payload of type `int`) and *an arbitrary number of* `Get` messages (with a payload of type ClientReply, allowing the future to respond with its result). The infix $\cdot$ connective represents the *multiplicative, commutative* composition of patterns whereas the postfix * connective represents *repetition* of patterns.
- The FullFuture type is an input type that describes the future mailbox after it has been resolved. Note that the type *does not* support receiving another `Resolve` message.
- The ClientReply type is an *output* type that describes an obligation to send a *single* `Reply` message with the future's value to the mailbox of a future's client.

We will see further examples making use of these types in Sections 4 and 5. However, informally, by making use of these types we can already rule out many of the previous errors: for example, attempting to send a `Resolve` message to a mailbox with type FullFuture or mis-spelling another of the message tags will result in a *static* error.

2.2 Lock Example

Let us now consider the example of an actor that implements a lock. Many processes can request the lock, and the lock process will grant the lock to a single client. The holder should eventually release the lock, and the lock should not be granted again until it is released. Although locks are commonly used in shared-memory programming, they are a higher-level abstraction and are often used in distributed programming to guarantee unique access to a resource. We can implement the lock in Elixir:

```
def freeLock() do
  receive do
    { :acquire, ownerPid } ->
      busyLock(ownerPid)
    { :release } ->
        raise "Bad release!"
  end
end
```

```
def busyLock(ownerPid) do
  send(ownerPid, { :acquired })
  receive do
    { :release } ->
      IO.puts("Lock released")
      freeLock()
  end
end
```

The lock process begins in the `freeLock` state, at which point it receives a message from its mailbox. A `:release` message at this stage is erroneous, since the lock is already free, so the process raises an exception. To handle an `:acquire` message, the lock transitions to the `busyLock` state and sends an `:acquired` message to the holder, before awaiting a `:release` message and transitioning back to the `freeLock` state.

The Lock example has many of the same potential issues as the Future example but also has some stronger correctness conditions: first, a lock should *only be released by the process that holds it*, and second (up-to issues such as non-termination on the part of the client), the lock *should always be released*. We can modify our Elixir code to always check that the release message is sent by the holder of the lock by sending the PID of the holder along with the `:release` message, and checking this against `ownerPid`:

```
def busyLock(ownerPid) do
  send(ownerPid, { :acquired })
  receive do
    { :release, pid } when pid == ownerPid ->
      IO.puts("Lock released")
      freeLock()
    { :release, pid } when pid != ownerPid ->
      IO.puts("Non-owner trying to release lock; ignoring")
      busyLock(ownerPid)
  end
end
```

Again this is defensive code that requires hard-coding the invariant and cluttering the code with identity checks. There is also no straightforward way, in Elixir, to ensure that the lock *must* be released.

Both of these new issues can be solved using mailbox types. We can define mailbox types for the Lock example as follows:

$$\begin{aligned}
\mathsf{FreeLock} &\triangleq {?}(\mathtt{Acquire}[\mathsf{LockReply}]^*)\\
\mathsf{BusyLock} &\triangleq {?}(\mathtt{Release} \cdot \mathtt{Acquire}[\mathsf{LockReply}]^*)\\
\mathsf{LockReply} &\triangleq {!}\mathtt{Acquired}[\mathsf{Unlock}]\\
\mathsf{Unlock} &\triangleq {!}\mathtt{Release}
\end{aligned}$$

Again, let us unpack these types.

- The FreeLock type describes the messages that can be received by a free lock: namely, *an arbitrary number of* `Acquire` messages. Each `Acquire` message has a payload of type LockReply, which allows and requires the server to respond to the request once it is received.

- The BusyLock type describes the messages that can be received by a busy lock: namely a *single* `Release` message and *an arbitrary number of* `Acquire` messages.
- The LockReply type is the type of a name that requires a process to send an `Acquired` message. It allows the server to respond to a lock request. Each `Acquired` message has the payload of type Unlock.
- The Unlock type is the type of a name that requires a process to send a `Release` message. It allows and requires the client to release the lock.

By writing a program that uses these types, we can guarantee that when the lock is free, its mailbox *only* contains `Acquire` messages, meaning that we can eliminate the check for invalid `Release` messages. When the lock is busy, its mailbox is expected to contain *one* `Release` message. For this reason, the holder of the lock has an *obligation* to send this `Release` message. Taken together this means that *only the holder of the lock can send a* `Release` *message*, and *the* `Release` *message must be sent*, solving both of the problems we encountered with our Elixir code. Additionally, we can delete the error-handling case from `freeLock` as the case is ruled out statically.

Note that a mailbox type does not necessarily describe the contents of the mailbox at a given moment in time. For example, as soon as a process has acquired the lock, the mailbox type of the lock's mailbox contains a `Release` message that will only be produced once the holder releases the shared resource. The fact that the holder of the lock *must* send a `Release` message compensates for this mismatch between the mailbox type and the actual state of the mailbox.

2.3 Guarantees

Although we have not yet described concrete languages making use of mailbox types, we can begin to see some of the guarantees we might expect from mailbox-typed languages:

Mailbox Conformance Processes will *only* receive messages that are allowed by a given mailbox type.

Deadlock-Freedom Mailbox types guarantee some degree of deadlock-freedom. The process calculus that we describe in Section 4 guarantees full deadlock-freedom, whereas the functional language that we describe in Section 5 guarantees freedom from self-deadlocks.

Junk-Freedom Modulo some caveats around recursion, mailbox type systems ensure *junk-freedom*: every message that is sent will eventually be received.[2]

[2] This is a stronger condition than most actor languages, which allow an actor to terminate before it has processed all of the messages in its mailbox. In mailbox-typed languages it is instead necessary to recursively process all remaining messages before terminating, which may not always be desirable.

3 On the Semantics of Mailbox Patterns and Subtyping

To better understand the meaning of mailbox types it is important to define a precise semantics of mailbox *patterns*. Types, atoms and patterns are terms generated by the following grammar:

$$\begin{array}{rl}
\textbf{Type} & \tau, \sigma ::= ?E \mid !E \mid \texttt{int} \mid \texttt{bool} \mid \cdots \\
\textbf{Atom} & M, N ::= \texttt{m}[\overline{\tau}] \\
\textbf{Pattern} & E, F ::= \mathbb{0} \mid \mathbb{1} \mid M \mid E + F \mid E \cdot F \mid E^*
\end{array}$$

A type is either the type of a mailbox with input ($?$) or output ($!$) capability or a type describing some other value such as an integer, a boolean, and so on. An atom $\texttt{m}[\overline{\tau}]$ consists of a *tag* $\texttt{m}$ and a (possibly empty) sequence $\overline{\tau}$ of types. It describes a *single occurrence* of a message in a mailbox. The tag is used to selectively retrieve the message from the mailbox and the types $\overline{\tau}$ describe the content (or payload) of the message. A *pattern* is a commutative regular expression over atoms describing sets of configurations expected to be found in a mailbox. A *configuration* is a *multiset* of atoms written $\langle M_1, \ldots, M_n \rangle$. Hereafter we let A and B range over configurations and we write $A \uplus B$ for the multiset union of A and B

The semantics of a pattern is inductively defined by the following equations:

$$\begin{aligned}
[\![\mathbb{0}]\!] &= \emptyset \\
[\![\mathbb{1}]\!] &= \{\langle\rangle\} \\
[\![M]\!] &= \{\langle M \rangle\} \\
[\![E + F]\!] &= [\![E]\!] \cup [\![F]\!] \\
[\![E \cdot F]\!] &= \{A \uplus B \mid A \in [\![E]\!], B \in [\![F]\!]\} \\
[\![E^*]\!] &= [\![\mathbb{1}]\!] \cup [\![E]\!] \cup [\![E \cdot E]\!] \cup \cdots
\end{aligned}$$

We can use these equations to compute the configurations of the patterns shown in Section 2. For example, we have

$$\begin{aligned}
[\![\texttt{Reply}]\!] &= \{\langle \texttt{Reply} \rangle\} \\
[\![\texttt{Get}^*]\!] &= \{\langle\rangle, \langle \texttt{Get} \rangle, \langle \texttt{Get}, \texttt{Get} \rangle, \ldots\} \\
[\![\texttt{Resolve} \cdot \texttt{Get}^*]\!] &= \{\langle \texttt{Resolve} \rangle, \langle \texttt{Resolve}, \texttt{Get} \rangle, \langle \texttt{Resolve}, \texttt{Get}, \texttt{Get} \rangle, \ldots\}
\end{aligned}$$

where we have omitted the payload types for simplicity. Thus, a mailbox whose pattern is `Reply` is meant to contain *exactly* one `Reply` message, while a mailbox whose pattern is $\texttt{Get}^*$ may contain an arbitrary number of `Get` messages. Finally, a mailbox whose pattern is $\texttt{Resolve} \cdot \texttt{Get}^*$ is meant to contain exactly one `Resolve` message and an arbitrary number of `Get` messages.

When a message is removed from a mailbox, the pattern that describes the mailbox configuration must be updated to reflect the new state of the mailbox. Such update is formalised via a partial operator, here denoted by $/$, closely connected to

Brzozowski's derivative for regular expressions [4] and defined below:

$$
\begin{aligned}
\mathbb{0}/M = \mathbb{1}/M &= \mathbb{0} \\
M/M &= \mathbb{1} \\
\mathtt{m}[\overline{\tau}]/\mathtt{m}'[\overline{\sigma}] &= \mathbb{0} && \mathtt{m} \neq \mathtt{m}' \\
(E + F)/M &= E/M + F/M \\
(E \cdot F)/M &= E/M \cdot F + E \cdot F/M \\
E^*/M &= E/M \cdot E^*
\end{aligned}
$$

Intuitively, if E describes the content of a mailbox, then E/M describes the content of the same mailbox after a message of type M has been retrieved.

Unlike Brzozowski's derivative, the operator / is partial. Specifically, $\mathtt{m}[\overline{\tau}]/\mathtt{m}[\overline{\sigma}]$ is undefined when $\overline{\tau} \neq \overline{\sigma}$. The rationale for this property of / is related to the fact that messages are selected based on their tag but not on the type of their payload. If we had a mailbox containing two messages with the same tag and differently typed payloads, in general we would not be able to discriminate between the two. As an example, if we retrieve a b-tagged message from a mailbox with pattern $E \stackrel{\text{def}}{=} \mathtt{m}[\mathtt{int}] \cdot \mathtt{m}[\mathtt{bool}] \cdot \mathtt{b}[\mathtt{bool}]$, then we are certain that its payload has type `bool`; but if we retrieve an a-tagged message from the same mailbox, we would not know whether its payload has type `int` or `bool`. Correspondingly, $E/\mathtt{b}[\mathtt{bool}]$ is defined but $E/\mathtt{a}[\mathtt{bool}]$ is not.

The definition of / given above can be relaxed if the type system supports a notion of *subtyping*, but this extension is quite technical and out of scope of the present chapter. The interested reader may refer to the literature [6, 10] for more details.

Exercise 1 (easy) Compute and compare the semantics of these mailbox patterns:

1. $\mathbb{0} \cdot \mathtt{A}$ and $\mathbb{1} \cdot \mathtt{A}$
2. $\mathbb{0} + \mathtt{A}$ and $\mathbb{1} + \mathtt{A}$
3. $(\mathtt{A} + \mathtt{B}) \cdot \mathtt{C}$ and $\mathtt{A} \cdot \mathtt{C} + \mathtt{B} \cdot \mathtt{C}$
4. $(\mathtt{A} + \mathtt{B})^*$ and $\mathtt{A}^* \cdot \mathtt{B}^*$

Exercise 2 (medium) Write patterns for the following mailbox configurations:

1. an even number of A or an odd number of B;
2. the same number of A, B and C;
3. twice as many A as the number of B.

Exercise 3 (medium)

1. Prove that M^*/M and M^* are equivalent;
2. Find a pattern E such that $(E/M) \cdot M$ is not equivalent to E.

4 The Mailbox Calculus

The Mailbox Calculus is a variant of the asynchronous π-calculus in which names represent mailboxes rather than channels and where messages consist of a *message tag* and a sequence of payloads.

4.1 The Mailbox Calculus by Example

We introduce the Mailbox Calculus by implementing the three examples from the previous section and arguing that the are well typed with respect to the mailbox types introduced earlier. A more comprehensive and more formal treatment of the calculus and of its typing rules is given by de'Liguoro and Padovani [6].

4.1.1 Future Example

We can implement the Future example in the Mailbox Calculus using two recursive definitions, one for each state of the variable, as follows:

$$\begin{aligned}
\mathsf{emptyFuture}(\mathit{self}) &\triangleq \mathit{self}?\mathtt{Resolve}(\mathit{val})\,\mathtt{.}\,\mathsf{fullFuture}[\mathit{self},\mathit{val}]\\[1em]
\mathsf{fullFuture}(\mathit{self},\mathit{val}) &\triangleq \mathtt{free}\ \mathit{self}\,\mathtt{.done}\\
&\quad + \mathit{self}?\mathtt{Get}(\mathit{sender})\,\mathtt{.}\,(\mathit{sender}\,\mathtt{!}\,\mathtt{Reply}[\mathit{val}]\mid\\
&\qquad\qquad\qquad\qquad\mathsf{fullFuture}[\mathit{self},\mathit{val}])\\
&\quad + \mathit{self}?\mathtt{Resolve}(\mathit{val})\,\mathtt{.}\,\mathtt{fail}\ \mathit{self}
\end{aligned}$$

These definitions closely follow the structure of the homonymous Elixir functions discussed in Section 2. Tags discriminate between types of messages, and the external choice operator + is used to selectively receive messages depending on their tag. Unlike the Elixir code, the mailbox from which processes receive messages is explicitly named (*self*). This makes it easy to model programming patterns involving *first-class mailboxes*. In line with other formulations of the asynchronous π calculus, output operations are continuation-free processes such as $\mathit{sender}\,\mathtt{!}\,\mathtt{Reply}[\mathit{val}]$, which is composed in parallel with the recursive process invocation $\mathsf{fullFuture}[\mathit{self},\mathit{val}]$.

Being an abstract model of computation, the Mailbox Calculus does not mandate a specific behaviour in the presence of multiple messages that match the guards of the branches. In this case, + behaves non-deterministically. Also, the $\mathtt{free}\ \mathit{self}$ input guard models the condition representing the fact that the mailbox can be garbage collected, and the $\mathtt{fail}\ \mathit{self}$ process form models a runtime error concerning the mailbox *self*. No error recovery mechanism is provided, but the presence of failing processes is important in order to formulate some key soundness properties ensured by the mailbox type system, most notably mailbox conformance.

Let us now try to convince ourselves that these process definitions are *well typed* according to the mailbox types defined in Section 2.1. The claim we want to verify is that emptyFuture uses *self* according to the mailbox type EmptyFuture and that fullFuture uses *self* according to FullFuture and *val* according to `int`.

Starting from emptyFuture, we see that the process begins by *reading* a `Resolve` message from the *self* mailbox. To check that this input operation makes sense we inspect EmptyFuture, which is a mailbox type with *input* capability ? and pattern $\texttt{Resolve}[\texttt{int}] \cdot \texttt{Get}[\mathsf{ClientReply}]^*$. We use the derivative operator to compute the pattern that describes the configuration of *self* after this message has been received. In this case we can establish that

$$(\texttt{Resolve}[\texttt{int}] \cdot \texttt{Get}[\mathsf{ClientReply}]^*)/\texttt{Resolve}[\texttt{int}] = \texttt{Get}[\mathsf{ClientReply}]^* \quad (1)$$

hence the subsequent invocation of fullFuture must be checked in a typing environment where *self* is associated with the type ClientReply and *val* with the type `int`, which is perfectly in line with the assumption made for the definition of fullFuture.

Let us move on to fullFuture, recalling that $\mathsf{FullFuture} = ?(\texttt{Get}[\mathsf{ClientReply}]^*)$. The process begins with a choice between three possibilities, each guarded by an input operation. The guard `free` *self* checks whether the *self* mailbox is empty. The fact that this is a sensible operation to perform on *self* is witnessed by the property $\langle\rangle \in [\![\texttt{Get}[\mathsf{ClientReply}]^*]\!]$, meaning that *self* is indeed allowed to be *empty* and without writing references to it. If this turns out to be the case, the process may safely terminate (`done`) without leaving unprocessed messages behind. It is fine forgetting about the variable *val*, which is a plain integer value.

The guard $\mathit{self}?\texttt{Get}(\mathit{sender})$ tries to consume a `Get` message from *self*. If this operation succeeds, the pattern that describes the residual configuration of *self* is

$$\texttt{Get}[\mathsf{ClientReply}]^*/\texttt{Get}[\mathsf{ClientReply}] = \texttt{Get}[\mathsf{ClientReply}]^* \quad (2)$$

and *sender* is associated with the type of the message payload, that is ClientReply.

At this stage, we are left with type checking the parallel composition

$$\mathit{sender}\,!\,\texttt{Reply}[\mathit{val}] \mid \mathsf{fullFuture}[\mathit{self}, \mathit{val}]$$

in a typing environment that associates *sender* to ClientReply, *val* to `int` and *self* to FullFuture. As in many substructural type systems, typing a parallel composition $P \mid Q$ implies *splitting* the typing environment in such a way that P and Q are well typed with respect to their own partition. In this case, it is easy to see that $\mathit{sender}\,!\,\texttt{Reply}[\mathit{val}]$ retains the ownership of *sender* while $\mathsf{fullFuture}[\mathit{self}, \mathit{val}]$ retains the ownership of *self*. The variable *val* can be safely shared among the two processes given that its type is *unrestricted*. Now, the invocation $\mathsf{fullFuture}[\mathit{self}, \mathit{val}]$ is obviously well typed and so is the output operation $\mathit{sender}\,!\,\texttt{Reply}[\mathit{val}]$. Indeed, ClientReply is a mailbox type with output capability and its pattern $\texttt{Reply}[\texttt{int}]$ matches the shape of the message being sent on it.

To conclude this type checking exercise we have to discuss the branch guarded by $\mathit{self}?\texttt{Resolve}(\mathit{val})$. This branch does not implement any useful operation of the future variable, because we expect future variables to be resolved only once, but the presence of the branch allows us to model explicitly the idea that "well-typed programs don't go wrong". From the viewpoint of type checking we proceed exactly as before, by using the derivative operator so as to compute the residual configuration of *self* after a `Resolve` message has been received. In this case we obtain

$$\mathtt{Get}[\mathsf{ClientReply}]^{*}/\mathtt{Resolve}[\mathtt{int}] = \mathbb{0} \tag{3}$$

meaning that, from now on, the mailbox *self* has no valid configuration of messages. This is a sign that the input operation we have tried to perform is never going to succeed. Finding ourselves in a branch of the program where *self* has type $?\mathbb{0}$, we are entitled to terminate the program here with `fail` *self*.

Exercise 4 (easy) Using the pattern derivative operator and the semantics of patterns given in Section 3, prove the equivalences (1), (2) and (3).

4.1.2 Lock Example

The modelling of the `freeLock` and `busyLock` functions of Section 2.2 in the Mailbox Calculus is based on the very same ingredients we have already seen in action in the Future example, as illustrated by the definitions below:

$$\begin{aligned}
\mathsf{freeLock}(\mathit{self}) &\triangleq \mathtt{free}\ \mathit{self}\,.\,\mathtt{done} \\
&\quad + \mathit{self}?\mathtt{Acquire}(\mathit{owner})\,.\,\mathsf{busyLock}[\mathit{self}, \mathit{owner}] \\
&\quad + \mathit{self}?\mathtt{Release}\,.\,\mathtt{fail}\ \mathit{self} \\
\mathsf{busyLock}(\mathit{self}, \mathit{owner}) &\triangleq \mathit{owner}!\mathtt{Acquired}[\mathit{self}] \mid \mathit{self}?\mathtt{Release}\,.\,\mathsf{freeLock}[\mathit{self}]
\end{aligned}$$

It is fairly easy to build a correspondence between the structure of the freeLock and busyLock processes and that of the FreeLock and BusyLock mailbox types, although this correspondence is not always exact. For example, the busyLock process is only capable of dealing with `Release` messages received from the *self* mailbox, whereas the BusyLock type indicates that the *self* mailbox may also contain an arbitrary number of `Acquire` messages. This is to be expected: a busy lock (which has been acquired) ignores any further acquisition request until it is released. Hence, its mailbox may contain `Acquire` messages even though busyLock may only handle `Release`. A similar thing happens with the emptyFuture process, which can only receive a `Resolve` message, and the EmptyFuture type, which allows `Get` messages to be stored in the mailbox even before the future variable is resolved. In this respect, mailbox types differ from session types in that they describe the (expected) content of a mailbox rather than the actual behaviour of the process using that mailbox.

We use this example to showcase the role of mailbox types when composing a process receiving messages from a mailbox (such as freeLock) with processes sending messages to a mailbox. To this aim, we model a simple user process as

$$\mathsf{user}(\mathit{self}) \triangleq \mathit{self}?\mathtt{Acquired}(l)\,.\,(l!\mathtt{Release} \mid \mathtt{free}\ \mathit{self}\,.\,\mathtt{done})$$

which is parametric in the user's mailbox *self*. The user process awaits for an $\mathtt{Acquired}$ message from its own mailbox, signalling that it has acquired the lock. It then $\mathtt{Release}$s the lock and deallocates its own mailbox. It is possible to establish that user uses *self* according to the mailbox type $?\mathtt{Acquired}[\mathsf{Unlock}]$ (see Exercise 5).

Consider now the process

$$(alice)(carol)(lock)\left(\begin{array}{l} \mathsf{freeLock}[lock] \mid \mathsf{user}[alice] \mid \mathsf{user}[carol] \\ \mid lock!\mathtt{Acquire}[alice] \mid lock!\mathtt{Acquire}[carol] \end{array}\right) \qquad (4)$$

which represents the composition of the lock (in its "free" state), two acquisition requests and the two users identified by their mailboxes named *alice* and *carol*. As in the π-calculus, the *restrictions* $(alice)$, $(carol)$ and $(lock)$ delimit the scope of the respective names. Establishing that this composition is well typed means establishing, for each restricted mailbox, some sort of balancing between the messages that are produced and those that are consumed. We can reason on such balancing by looking at mailbox types. Concerning *lock*, it is used by $\mathsf{freeLock}$ according to the type $\mathsf{FreeLock}$ and by each acquisition request according to the type $!\mathtt{Acquire}[\mathsf{LockReply}]$. If we compare the patterns describing the overall messages that are produced and those that are consumed we see that the relation

$$[\![\mathtt{Acquire}[\mathsf{LockReply}] \cdot \mathtt{Acquire}[\mathsf{LockReply}]]\!] \subseteq [\![\mathtt{Acquire}[\mathsf{LockReply}]^*]\!]$$

holds, confirming that every $\mathtt{Acquire}$ message that is produced can also be consumed.

Concerning *alice*, it is used directly by $\mathsf{user}[alice]$ according to the type $?\mathtt{Acquired}[\mathsf{Unlock}]$, but we should not forget that it also appears as payload in one of the two acquisition requests. The type of that payload is $\mathsf{LockReply} = !\mathtt{Acquired}[\mathsf{Unlock}]$. By comparing the patterns in these types we see again that there is a perfect balancing between the messages that are produced and those that are consumed. A similar argument holds for *carol*, which is used directly by $\mathsf{user}[carol]$ and appears also in the other acquisition request. Having established the balancing between outputs and inputs, we can thus conclude that the composition (4) is well typed and therefore correct.

Exercise 5 (medium) Along the lines of what has been done for the future variable in the previous section, elaborate an informal argumentation assessing that $\mathsf{freeLock}$ and $\mathsf{busyLock}$ are well typed using the mailbox types $\mathsf{FreeLock}$ and $\mathsf{BusyLock}$ defined in Section 2.2.

5 Pat: A Functional Language with Mailbox Types

So far we have seen some motivating examples and their mailbox types, and we have seen how to express the examples in the Mailbox Calculus. However, the Mailbox Calculus is a minimal core *process calculus* and does not include many of the abstractions and constructs needed for programming. For example, the Mailbox Calculus does not support expressions that return values; it does not allow variable renaming; and it does not support more interesting control flow such as nested evaluation contexts.

Furthermore, whereas a process calculus succinctly describes the concurrent behaviour of a system (e.g., representing processes composed in parallel), it is difficult to see how this configuration arises from a *static* program that one would write in a text editor or IDE.

This section gives an informal introduction to Pat, a first-order functional programming language with mailbox types. Moving from a process calculus to a programming language involves overcoming some non-trivial challenges due to aliasing (Section 5.2). A comprehensive treatment of the language and its metatheory can be found in Fowler et al. [10]. We begin by introducing Pat using the examples from Section 2.

5.1 Pat by Example

5.1.1 Future Example

We begin by writing *mailbox type interfaces* for the two kinds of mailbox. An interface specifies a set of types of messages that a mailbox can receive, but (unlike a mailbox type) does not specify the valid patterns of messages. For the Future example we need two types of mailbox: a `Future` mailbox which can receive `Resolve` and `Get` requests, and a `User` mailbox that can receive `Reply` messages. The payload type of the `Get` message is `User!`, denoting a *send* reference to a `User` mailbox. Note that we do *not* need to fill in the pattern, which will be inferred by the typechecker.

```
interface Future { Resolve(Int), Get(User!) }
interface User   { Reply(Int) }
```

The `emptyFuture` and `fullFuture` functions closely follow the Elixir definitions in Section 2:

```
def emptyFuture(self: Future?): Unit {
    guard self : Resolve.Get* {
        receive Resolve(x) from self ->
            fullFuture(self, x)
    }
}

def fullFuture(self: Future?, value: Int): Unit {
    guard self : Get* {
        free -> ()
        receive Get(user) from self ->
            user ! Reply(value);
            fullFuture(self, value)
    }
}
```

As with the Mailbox Calculus, mailbox references (e.g. `self` and `user`) are *first class*. However, whereas the Mailbox Calculus is a minimal process calculus, Pat is a functional programming language and includes features such as sequential composition of arbitrary expressions (as opposed to the Mailbox Calculus approach of allowing parallel composition of send actions and sequential composition of receive actions).

Concretely, the `emptyFuture` definition takes a mailbox reference `self` of type `Future?` (denoting that the name is the *input* reference and has the `Future` interface). It begins by receiving from the mailbox using the **`guard`** expression, which is annotated with the pattern of messages that the mailbox is expected to contain. The **`guard`** block *consumes* the `self` variable, meaning that it cannot be used again. The **`receive`** clause details a selective receive for a `Resolve` message from the mailbox, and binds the payload of the message to the variable `x`. The *continuation* of the mailbox (with the type `?Get*`, calculated via the pattern residual operator / defined in Section 3), is bound to the `self` variable, so that it can be used for future receives. Given that the future has now been resolved and should not be able to support further `Resolve` messages and instead should process `Get` messages, it calls `fullFuture` with the re-bound mailbox name and the received value.

The `fullFuture` definition again receives from `self`, but notice that this time the expected pattern only permits many `Get` messages. The **`guard`** block contains two clauses: the **`free`** clause is invoked when the mailbox is empty and is expecting no more messages (i.e., when there are no more output references to the mailbox) and allows the future to terminate. We also sometimes use **`free`**`(x)` as syntactic sugar for **`guard`** `x : 1 {` **`free`** `->() }`. The **`receive`** clause receives a `Get` message from the mailbox, binding its payload to the `user` variable and the continuation of the mailbox to `self`. The future then sends a `Reply` message to `user`, containing the future's `value`, before recursively processing the remaining requests.

We can set the Future example up as follows:

```
def user(future: Future!): Int {
    let self = new[User] in
    future ! Get(self);
    guard self : Reply {
        receive Reply(x) from self ->
            free(self);
            x
    }
}

def main(): Unit {
    let futureMb = new[Future] in
    spawn { emptyFuture(futureMb) };
    futureMb ! Resolve(5);
    print(intToString(user(futureMb)));
    print(intToString(user(futureMb)))
}
```

The `main` definition is the entry-point of the program. It creates a new mailbox `futureMb` for the future using the **`new`** construct, specifying that the new mailbox should satisfy the `Future` interface, and spawns a process that runs the `emptyFuture` function with futureMb as its argument. Next, the process sends a Resolve message with the payload 5, and sequentially makes two requests by calling the user function.

The user function takes a send reference to a Future mailbox. It creates a new User mailbox called self from which it can receive the result, and then sends a Get message to the future, with a reference to self included as a payload. The user function then awaits a Reply message from self, before freeing it and returning the received value. Note that freeing a mailbox is possible since mailbox types are precise enough to allow us to *statically determine* that only a single message will ever be sent to self.

The Pat typechecker then infers the precise patterns for each mailbox-typed parameter:

```
def future(self: Future?(Resolve . Get . Get)): Unit { ... }
def fullFuture(self: Future?(Get*), value: Int): Unit { ... }
def user(future: Future!(Get)): Int { ... }
```

5.1.2 Lock Example

We start writing the Lock example in Pat by specifying the interfaces for the Lock and User mailboxes:

```
interface Lock { Acquire(User!), Release() }
interface User { Acquired(Lock!) }
```

Here a Lock mailbox can receive an Acquire message, which specifies a request to acquire the lock, and a Release message. The payload of the Acquire message is an output reference to a User mailbox so that the client can be notified whenever the lock is acquired. The lock can be written as two mutually-recursive functions, freeLock and busyLock, as follows:

```
def freeLock(self: Lock?): Unit {
    guard self : Acquire*  {
        free -> ()
        receive Acquire(owner) from self ->
            busyLock(self, owner)
    }
}

def busyLock(self: Lock?, owner: User!): Unit {
    owner ! Acquired(self);
    guard (self) : Acquire*.Release {
        receive Release() from self ->
            freeLock(self)
    }
}
```

The freeLock function takes a mailbox name satisfying the Lock interface as its parameter, and guards on it. At this point, since the lock is free, it should only contain Acquire messages. As before there are two clauses: a **free** clause to free the mailbox and terminate the lock once there are no more requests, and a **receive** clause to handle an Acquire message. Note that the function does *not* need to include a clause to handle a `Release` message as this is ruled out by the type. After receiving an `Acquire` message, the function calls the `busyLock` function that sends an `Acquired` message to the new holder of the lock before awaiting a `Release` message.

Note that the guard in the `busyLock` function does not include a **`free`** guard, because it must eventually be able to handle a `Release` message from the holder of the lock. Note also that there is no need to handle any `Acquire` messages: while it is permissible for the mailbox to contain many `Acquire` messages, these will not be processed until the lock is released and the lock returns to the `freeLock` state.

The system can be set up as follows:

```
def user(num: Int, lock: Lock!): Unit {
    let self = new[User] in
    lock ! Acquire(self);
    guard(self) : Acquired {
        receive Acquired(lock) from self ->
            print(intToString(num));
            lock ! Release();
            free(self)
    }
}

def main(): Unit {
    let lock = new[Lock] in
    spawn { freeLock(lock) };
    spawn { user(1, lock) };
    spawn { user(2, lock) }
}
```

In this case, the `user` function is given a numeric identifier and a send reference to the lock. It first creates a new mailbox `self` with the `User` interface, and sends an `Acquire` message to the lock with an output reference to `self` as a payload. The process then waits for an `Acquired` message to denote that it has acquired the lock. The `Acquired` message contains a send mailbox reference that allows and requires the user to send a `Release` message to release the lock.

It is worth contrasting this example with the Elixir example in Section 2.2. The first iteration of the example suffered from the issue that *any* client—even a client that did not currently hold the lock—could release the lock by sending a `:release` message. The second iteration of the example fixed this issue by tracking the owner of the lock and checking that the identity contained in the `:release` message matched the holder of the lock, but this required explicit checking logic. Furthermore, there was no way of ensuring that the holder of the lock should eventually release it.

In contrast, the more precise nature of mailbox types allow us to avoid both of these problems in the Pat program (and also in the Mailbox Calculus). Specifically:

- All existing references to the lock process have a type that only allows them to send `Acquire` messages.
- After sending the `Acquired` message (along with a new output reference) to the lock, the lock waits for a `Release` message.
- Since the lock waits for a `Release` message, at least one process must *send* the `Release` message. Since all other references can only send `Acquire` messages, the `Release` message *must* be sent by the reference returned to the process as part of

the `Acquired` message. In turn, this solves *both* problems originally apparent in the original Elixir code, without the need for any explicit checks.
- Once the process sends a `Release` message, it *cannot* send another `Release` message later on, since the communication would be unbalanced (with two `Release` messages sent and only one received). This is ruled out statically.

5.2 Technical Challenges

The biggest difference between a programming language and a process calculus is that a programming language design needs *static* constructs that give rise to dynamic behaviour. For example, consider parallel composition in the Mailbox Calculus, where $P \mid Q$ denotes two processes P and Q evaluating in parallel. As we have seen in the examples so far, Pat has a variety of static constructs that mirror the process states in the Mailbox Calculus, and has a core calculus built on the ideas behind various concurrent λ-calculi (e.g. [22, 23]). For example, `spawn { M }` is a static construct that, when evaluated, gives rise to term `M` evaluating in parallel with the calling thread. However, the move from a process calculus to a programming language brings challenges with name hygiene that we will explore by example.

5.2.1 Unsafe Aliasing

This static fragment of the language supports the usual programming idioms such as sequential composition, recursion, and let-binding. Unfortunately, this additional expressive power brings with it some opportunities for errors. Consider the following snippet of code (recalling that the variable after `from` is in *binding* position and re-binds a mailbox name):

```
def bad1(): Unit {
    let a = new[Test] in
    guard a : Msg {
        receive Msg(x) from b ->
            free(b)
    };
    a ! Msg("Hello")
}
```

This code creates a new mailbox `a` with interface `Test`, before waiting to receive a `Msg` message and freeing the mailbox. Only *after* that point does the program send the `Msg` message to `a`.

From a mailbox typing perspective, the code is correct since the send of `Msg` balances out with it being received. However, the sequential nature of the program means that the *ordering* of these operations is not accounted for by the mailbox types, and so there are two serious problems in practice:

Self-deadlock Receiving from `a` will block indefinitely because the mailbox is initially empty, and the send operation will never be evaluated.

Use-after-free The `free(a)` construct appears to free and deallocates the mailbox after `Msg` is received, a message is sent lexically *after* the mailbox has been freed and should no longer be in scope.

The use-after-free problem becomes more apparent when we consider a mailbox pattern that includes repetition:

```
def bad2(x : Test?) : Unit {
    guard x : Msg* {
        receive Msg() from y -> bad2(y)
        free -> x ! Msg()
    }
}
```

In this case the code would typecheck *even when* `a` *is used in the* `free` *guard itself!* Worse, the obvious approach of requiring that a mailbox name used as the subject of a `guard` expression cannot also be used within guard clauses fails as soon as we introduce an alias for the mailbox:

```
def bad3(x : Test?) : Unit {
    let a = x in
    guard a : Msg* {
        receive Msg() from y -> bad3(y)
        free -> x ! Msg()
    }
}
```

5.2.2 Aliasing through Communication

Since Pat is a concurrent language, it is also possible to introduce aliasing as a result of communication. Consider the following program:

```
interface Self { Msg(Other!) }
interface Other { Reply() }
def main(): Unit {
 let self = new[Self] in
 let other = new[Other] in
  spawn {
   interprocess(self, other)
 };
 self ! Msg(other);
 guard other : Reply {
  receive Reply() from other ->
   free(other)
 }
}
```

```
def interprocess(self: Self?,
    other1: Other!): Unit {
 guard self : Msg {
  receive Msg(other2)
        from self ->
   other1 ! Reply();
   free(self)
 }
}
```

Here, the `main` function creates two mailboxes, `self` and `other`, and spawns a process running the `interprocess` function supplied with references to both mailboxes. The function then sends a `Msg` message along with another reference to `other`. The `interprocess` function then receives from the `self` mailbox, but the body of the `receive` clause then contains *two* names for the same underlying mailbox.

5.3 Enforcing Name Hygiene

In short, to avoid the issues raised by aliasing, Pat programs need to satisfy the following properties:

- **Mailboxes should have a single static name:** Within a given scope in a single thread, no two distinct variables should refer to the same underlying mailbox (for example, x and z in our use-after-free example).
- **Mailbox name consumption:** A mailbox variable is deemed out-of-scope as soon as it is let-bound or used as the subject of a **guard** expression.

These two requirements allow us to rule out the aliasing issues described in the previous section. A common approach used in programming language designs incorporating session types [13] is to use *linear typing* where each name can be used *precisely once*. Unfortunately, though, linear typing is too strict to support programming with mailbox types: unlike session types which support point-to-point interaction, mailbox types allow *many* senders and a *single* receiver. For example, recall the `main` function of the Future example (abbreviating `intToString` as `i2s` and the `Future` interface to `F`):

```
def main(): Unit {
  let futureMb = new[F] in
  spawn {
      emptyFuture(futureMb)
  };
  futureMb ! Resolve(5);
  print(i2s(user(futureMb)));
  print(i2s(user(futureMb)))
}
```

Here the `futureMb` mailbox is used *four times*: once as an input reference that is passed to the `emptyFuture` function; once as an output reference used to send a `Resolve` message, and twice as an output reference passed to the `user` function so that it can be used to send a `Get` message.

5.3.1 Quasi-Linear Typing

Pat's solution to the problems introduced by aliasing is inspired by *quasi-linear typing*, first introduced by Kobayashi [20]; the formulation used by Pat is closer to that of Ennals et al. [7]. Quasi-linear typing allows for multiple uses of a variable as long as the *last lexical occurrence of the variable in the current scope is used linearly*, with preceding occurrences used in a *second-class* way. A second-class value may only be used as part of an expression (e.g. as an argument to a function call, or as the target of a send operation) rather than being returned (e.g. returned as part of a tuple or the subject of a let-binder).

In Pat, we make use of these ideas to allow mailbox names to be used many times for output, but *only once* to receive messages. Second-class occurrences of a name can be used as the target of a send or as part of the payload of a message. In contrast, *only first-class references* may be returned from the subject of a let-expression, and used as the subject of a **guard** expression. Since the first-class occurrence must be the final lexical occurrence of a name, a **guard** expression necessarily consumes the mailbox name and thus avoids the previous issues with use-after-free.

Suppose we have a Pat mailbox type `Interface!E` or `Interface?E`, where `E` refers to a pattern. We can augment these types with annotations `<R>` for a returnable value, or `<U>` for a second-class value.

We can show how our use-after-free error can be solved using quasi-linear types as follows. Let us begin by seeing how the technique works with a program that is not erroneous, by annotating each bound occurrence of a mailbox name with its type:

```
def good(): Unit {
    let a = new[Test] in
    a ! Msg("Hello");              # a : Test!Msg<U>
    guard a : Msg {                # a : Test?Msg<R>
        receive Msg(x) from b ->
            free(b)                # b : Test?1<R>
    };
}
```

The first bound occurrence of name `a` is used to send a `Msg` message, and so can be given type `Test!Msg`. Since it is only used to send a message (as opposed to appearing in the result of an expression or being used to receive a message), it can also be given quasi-linearity annotation `<U>` to recognise that it is a second-class value. The next occurrence of `a` occurs as the subject of a **guard** expression, and so *must* be returnable (hence given the type `Test?Msg<R>`. Since the returnable occurrence of a name must be the last in the scope, this ensures that `a` is "consumed" and cannot be reused later in the definition. The **receive** clause re-binds the mailbox (this time with name `b`), with a type indicating that the `Msg` message has been consumed. Therefore `b` can be given type `Test?1<R>` and freed. Note that the Mailbox Calculus allows name re-use *without* re-binding, since the typing rules for typed process calculi allow us to update the type of the mailbox in the continuation of the guard. Re-using a name with an updated type is more difficult to do in a functional language due to the fact that the subject of a **guard** can be an arbitrary expression rather than a variable literal. This difference is also evident in session-typed process calculi [12] (which allow re-use without re-binding) and session-typed functional languages [13, 22] (which require re-binding of a name after a communication action).

To see how quasi-linearity can rule out the errors we saw previously, let us revisit our `bad1` function:

```
def bad1(): Unit {
    let a = new[Test] in
    guard a : Msg {                # a : Test?Msg<R>
        receive Msg(x) from b ->
            free(b)                # b : Test?1<R>
    };
    a ! Msg("Hello")               # a : Test!Msg<U> (ERROR)
}
```

Again we create a mailbox `a` and receive from it. Since we are using `a` as the subject of a **guard** expression, it *must* be a returnable usage. Next, we re-bind the continuation of the mailbox to b and free it as before. However, in the code following the **guard** expression we try to send a message to a, which has type Test!Msg<U>. However this is a *type error* since a second-class occurrence appears after the previous returnable occurrence. Note that if we were to give the final occurrence of a the type Test!Msg<R>, this would *also* be an error since there can only be a *single* returnable occurrence of a name per thread.

For the same reason, we can rule out our second incorrect program:

```
def bad2(x : Test?) : Unit {
    guard x : Msg* {              # x : Test?Msg*<R>
        receive Msg() from y ->
          bad2(y)                 # y : Test?Msg*<R>
        free -> x ! Msg()         # x : Test!Msg<U> (ERROR)
    }
}
```

Here we try to re-use a after it has been consumed by the **guard** expression. More specifically we attempt to use a as a second-class use *after* its returnable use as the subject of **guard**.

The approach also scales to the example where we introduce an alias a for x:

```
def bad3(x : Test?) : Unit {
    let a = x in                          # x : Test?Msg*<R>
    guard a : *Msg {                      # a : Test?Msg*<R>
        receive Msg() from y -> bad3(y)
        free -> x ! Msg()                 # x : Test!Msg<U> (ERROR)
    }
}
```

Because x occurs in the result of a let-expression, it *must* be returnable and so is consumed. Later, when trying to send in the **free** clause, x appears after the returnable usage and so is ruled out statically.

5.3.2 Masking

Note that the above constraints are only relevant *within a single thread*. If a receive operation happens in a parallel thread, then it does not matter whether the receive operation is the last lexical occurrence. Let us now return to the main function of the Future example, adding type annotations on the mailbox references:

```
def main(): Unit {
  let futureMb = new[F] in
  spawn {
      emptyFuture(futureMb)     # futureMb : F?(Resolve.Get*)<R>
  };
  futureMb ! Resolve(5);        # futureMb : F!Resolve<U>
  print(i2s(user(futureMb)));   # futureMb : F!Get<U>
  print(i2s(user(futureMb)))    # futureMb : F!Get<U>
}
```

Here, the emptyFuture function requires an input capability for the mailbox, which necessarily must be returnable. This poses a problem, since the call to emptyFuture occurs prior to two second-class usages. However, since this occurrence is in the body of the `spawn` expression, it will be used in a *separate thread*, so we can safely *mask* it as a second-class usage when typing the remainder of the program after the `spawn`.

Exercise 6 (easy–medium) For each of the following Pat functions, annotate each bound mailbox-typed variable with a mailbox type and quasilinearity annotation, and state whether the function would be typable. Assume an interface `interface I { Msg1(), Msg2() }`.

(a)

```
def ql1(): Unit {
  let mb = new[I] in
  mb ! Msg1();
  guard mb : Msg1 {
    receive Msg1() from mb ->
      free(mb)
  }
}
```

(b)

```
def ql2(): Unit {
  let mb = new[I] in
  mb ! Msg1();
  guard mb : Msg1 {
    receive Msg1() from self ->
      free(mb);
      free(self)
  }
}
```

(c)

```
def ql3(): Unit {
  let mb = new[I] in
  spawn {
    guard mb : Msg1 {
      receive Msg1() from mb ->
        free(mb)
    }
  };
  mb ! Msg1()
}
```

(d)

```
def ql4(): Unit {
  let mb = new[I];
  mb ! Msg1();
  let mb2 = mb in
  mb ! Msg2();
  guard mb2 : Msg1 . Msg2 {
    receive Msg1() from mb2 ->
      guard mb2 : Msg2 {
        receive Msg2() from mb2 ->
          free(mb2)
      }
  }
}
```

5.3.3 Preventing aliasing through communication

Due to the absence of a `let` construct in the Mailbox Calculus, aliasing can only occur due to communication, and since names in the Mailbox Calculus are static literals, the Mailbox Calculus can construct a *dependency graph* to rule out such aliasing. Informally, a dependency arises if a mailbox name as a payload in a message to another mailbox, or if a mailbox name occurs free in the continuation of a receive guard. The dependency graph in the Mailbox Calculus was in fact originally conceived as a device to ensure deadlock freedom, and alias control is a welcome side-effect.

Since it is not possible to use the same approach in Pat (see Section 5.4), Pat rules out potential communication-based aliasing using a syntactic approximation: in a receive guard `receive Msg(x1, ..., xn) ->M`, none of the message payloads `x1, ..., xn` can have the same interface type as any free variable in the body `M`, and any received name is treated as *second-class* rather than returnable in order to avoid violating the requirement that only the final lexical occurrence of a name can be returnable.

We have found that these restrictions are flexible enough to encode all of the examples from the Savina benchmark suite [17], but incorporating more intricate inter-process alias control for Pat is left for future work.

5.4 Comparison to the Mailbox Calculus

Whereas the Mailbox Calculus aims to investigate mailbox typing in the context of a core model of concurrency, Pat's purpose is to act as a programming language. In general this means that Pat needs to consider a wider range of programming features, particularly those that introduce more interesting flow control (e.g., nested evaluation contexts requiring a call stack, and potential aliasing due to let-bindings). This adds to the complexity of the language but means that it can be used to express realistic examples with a strong distinction between static terms and the configurations to which they evaluate at runtime.

The big technical difference between the two models however is the techniques used for alias control. The Mailbox Calculus rules out unsafe aliasing and deadlocks using a dependency graph. Unfortunately the dependency graph construct is difficult to integrate with a functional language because mailbox names are dynamically generated from the `new` construct as opposed to being known statically, and because the more complex control flow cannot be easily accounted for in the dependency graph design. Quasi-linear typing, along with the syntactic restrictions on received values, suffice to support proving type preservation and avoidance of self-deadlocks, but Pat cannot guarantee full deadlock-freedom.

5.5 Pat Exercises

The following exercises give experience with writing some simple Pat programs. To typecheck your solutions, you can download the MBCheck tool from GitHub (`https://github.com/simonjf/mbcheck`); the GitHub page includes installation and usage instructions.

Exercise 7 (easy) Write an "echo" program in Pat. Your program should include:

- A server process that infinitely awaits a `Request(data, replyTo)` message, where `data` has type `Int`. Upon receiving the `Request` message, the server should reply to the client by sending a `Reply(data)` message that contains the same data as in the request.
- A client process that sends a `Request` message to the server, receives a `Reply` message, and terminates.
- A `main` process that spawns the server process and a number of clients.

Your server should be able to handle an unbounded number of clients.

Exercise 8 (easy) Write a program consisting of two processes:

- A "pinger" process that nondeterministically chooses either to send a `Ping` message to a server before recursively calling itself, or to terminate. You can use the `randBool()` function to generate a random Boolean.
- A server process that consumes each `Ping` message, and frees itself when there are no more messages to consume.

Exercise 9 (hard) A two-factor authentication procotol can be described as follows:

- A client sends a `LoginRequest` message containing a username and password to a login server.
- The server checks the credentials. If the credentials are correct, then the server either sends a `AuthGranted` message, or a `Challenge` message containing a challenge key (represented as a string). If the credentials are incorrect, then the server sends a `AuthDenied` message.
- Upon receiving a `Challenge` message, the client sends the server a `ChallengeResponse` message containing its response.
- The server checks the response before responding with either `AuthGranted` or `AuthDenied`.

Write Pat program that implements the above protocol. You can assume the following functions (or write function stubs):

- `def checkDetails(username: String, password: String): Bool`, which returns `true` if the credentials are valid, and `false` if not.
- `def genChallengeKey(): String`, which generates a challenge key.
- `def challengeResponse(challenge: String): String`, which generates a challenge response given a challenge key.

- `def responseValid(response: String): Bool`, which returns `true` if `response` is a valid two-factor authentication challenge response, and `false` otherwise.

Your server should be able to handle multiple clients at once, and should be able to handle requests from other clients between sending a challenge request and receiving a response.

6 Related Work

In Section 1 we referred briefly to selected work on session types for actor languages [8, 11, 14]. More comprehensive analysis of the relevant literature can be found in our research papers on mailbox types [6, 10]. Here we focus on comparisons with two chapters in the present volume, which also deal with asynchronous message-passing communication in concurrent or distributed systems.

Chapter 3 deals with a subtyping relation for session types, and an algorithm for checking it. The chapter focuses on *asynchronous* subtyping, which (in a setting of asynchronous channel-based communication using message queues) allows send operations to be moved earlier than receive operations. The asynchronous subtyping relation thus relaxes some of the strict message ordering specified by session types. The mailbox type system also uses subtyping, but because mailbox types are unordered from the beginning, it is not necessary for subtyping to consider reordering messages. Instead, subtyping between mailbox types is used to compare the communication operations inferred from code with the permitted contents of a mailbox as expressed by its type, so that a program cannot violate the intended invariants.

Chapter 5 concerns a model of distributed computation in which a *swarm* of components communicate by asynchronous message-passing. Although the model initially seems similar to the actor model, it is substantially different because messages are not sent to particular recipients — instead, they are treated as events that any component may respond to. The focus of the work is a type system that can guarantee eventual consistency between components, even though they can make local decisions without taking into account all of the available global information. The type system has much in common with the approach of multiparty session types [16], including a foundation in ordered sequences of messages; it does not have an obvious connection with mailbox typing.

7 Conclusion

Mailbox types are a recent behavioural type system that allow developers to ensure the communication correctness of many-sender, single-receiver interactions such as those found in common actor languages, where processes receive from a mailbox. In contrast to other behavioural type disciplines for communicating systems, mailbox types concentrate on *patterns* of unordered interactions, specifying invariants on the contents of a mailbox expressed as a commutative regular expression.

This chapter has introduced mailbox types both by example and by formally defining their semantics, and has shown how mailbox types can be used in a process calculus, the *Mailbox Calculus*, and a programming language, *Pat*.

7.1 Ongoing and Future Research Directions

There are many interesting future research challenges in the field of mailbox typing. First, both the Mailbox Calculus and Pat support first-class, explicit mailboxes. We are currently investigating how to make use of mailbox types to ensure safety of actor languages such as Erlang and Elixir that instead have implicit, monolithic mailboxes.

Second, although both MC^2 and MBCheck support a limited amount of inference, users must specify explicit patterns on each guard, and so improved mailbox type inference would greatly ease the development of larger mailbox-typed applications. Third, mailbox types do not support any kind of failure handling, which is critical for real-world applications. Fourth, mailbox typing is handled entirely statically; it would be interesting to investigate dynamic or hybrid approaches to mailbox typing, potentially to handle external messages via boundary monitors or to enforce more interesting properties on message payloads.

On the theoretical side, it is difficult not to notice several intriguing similarities between mailbox typing rules and linear logic proof rules. Whether and how it is feasible to provide a logical foundation to mailbox types, as it has been done for session types [5, 24], are open research questions.

References

[1] Gul A. Agha. *Actors: a Model of Concurrent Computation in Distributed Systems*. PhD thesis, University of Michigan, USA, 1985.

[2] Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. Behavioral types in programming languages. *Foundations and Trends® in Programming Languages*, 3(2-3):95–230, 2016.

[3] Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 2003.

[4] Janusz A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4): 481–494, 1964. doi: 10.1145/321239.321249.

[5] Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logic propositions as session types. *Math. Struct. Comput. Sci.*, 26(3):367–423, 2016. doi: 10.1017/S0960129514000218.

[6] Ugo de'Liguoro and Luca Padovani. Mailbox types for unordered interactions. In *Proceedings of ECOOP*, volume 109 of *LIPIcs*, pages 15:1–15:28. Schloss Dagstuhl — Leibniz-Zentrum für Informatik, 2018.

[7] Robert Ennals, Richard Sharp, and Alan Mycroft. Linear types for packet processing. In *Proceedings of ESOP*, volume 2986 of *Lecture Notes in Computer Science*, pages 204–218. Springer, 2004.

[8] Simon Fowler and Raymond Hu. Speak now: Safe actor programming with multiparty session types. Draft, October 2025., 2023. URL https://simonjf.com/drafts/maty-draft-oct25.pdf.

[9] Simon Fowler, Sam Lindley, and Philip Wadler. Mixing metaphors: Actors as channels and channels as actors. In *ECOOP*, volume 74 of *LIPIcs*, pages 11:1–11:28. Schloss Dagstuhl — Leibniz-Zentrum für Informatik, 2017.

[10] Simon Fowler, Duncan Paul Attard, Franciszek Sowul, Simon J. Gay, and Phil Trinder. Special delivery: Programming with mailbox types. *Proc. ACM Program. Lang.*, 7(ICFP):78–107, 2023.

[11] Adrian Francalanza and Gerard Tabone. ElixirST: A session-based type system for Elixir modules. *J. Log. Algebraic Methods Program.*, 135:100891, 2023.

[12] Simon J. Gay and Vasco T. Vasconcelos. *Session Types*. Cambridge University Press, 2025.

[13] Simon J. Gay and Vasco Thudichum Vasconcelos. Linear type theory for asynchronous session types. *J. Funct. Program.*, 20(1):19–50, 2010.

[14] Paul Harvey, Simon Fowler, Ornela Dardha, and Simon J. Gay. Multiparty session types for safe runtime adaptation in an actor language. In *ECOOP*, volume 194 of *LIPIcs*, pages 10:1–10:30. Schloss Dagstuhl — Leibniz-Zentrum für Informatik, 2021.

[15] Carl Hewitt, Peter Boehler Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *IJCAI*, pages 235–245. William Kaufmann, 1973.

[16] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9:1–9:67, 2016. doi: 10.1145/2827695. URL https://doi.org/10.1145/2827695.

[17] Shams Mahmood Imam and Vivek Sarkar. Savina — an actor benchmark suite: Enabling empirical evaluation of actor libraries. In *Proceedings of AGERE!@SPLASH*, pages 67–80. ACM, 2014.

[18] Simon Peyton Jones. Beautiful concurrency. *Beautiful Code: Leading Programmers Explain How They Think*, pages 385–406, 2007.

[19] Sasa Juric. *Elixir in Action*. Manning Publications, 2024.

[20] Naoki Kobayashi. Quasi-linear types. In *Proceedings of POPL*, pages 29–42. ACM, 1999.

[21] Joeri De Koster, Tom Van Cutsem, and Wolfgang De Meuter. 43 years of actors: a taxonomy of actor models and their key properties. In *Proceed-*

ings of AGERE!@SPLASH, pages 31–40. ACM, 2016. doi: 10.1145/3001886.3001890.

[22] Sam Lindley and J. Garrett Morris. A semantics for propositions as sessions. In *Proceedings of ESOP*, volume 9032 of *Lecture Notes in Computer Science*, pages 560–584. Springer, 2015.

[23] Joachim Niehren, Jan Schwinghammer, and Gert Smolka. A concurrent lambda calculus with futures. *Theor. Comput. Sci.*, 364(3):338–356, 2006.

[24] Philip Wadler. Propositions as sessions. *J. Funct. Program.*, 24(2-3):384–418, 2014. doi: 10.1017/S095679681400001X.

Appendix: Solutions to Selected Exercises

Solutions to Exercises in Section 3 (Semantics of Patterns)

Exercise 1

We only show the solution of item (4). In this case we have

$$[\![(\mathtt{a}+\mathtt{b})^*]\!] = [\![\mathbb{1}]\!] \cup [\![\mathtt{a}+\mathtt{b}]\!] \cup [\![(\mathtt{a}+\mathtt{b})\cdot(\mathtt{a}\cup\mathtt{b})]\!] \cup \cdots$$

Since $[\![\mathtt{a}+\mathtt{b}]\!] = [\![\mathtt{a}]\!] \cup [\![\mathtt{b}]\!] = \{\langle\mathtt{a}\rangle\} \cup \{\langle\mathtt{b}\rangle\} = \{\langle\mathtt{a}\rangle, \langle\mathtt{b}\rangle\}$, we conclude

$$[\![(\mathtt{a}+\mathtt{b})^*]\!] = \{\langle\mathtt{a}^m, \mathtt{b}^n\rangle \mid m, n \in \mathbb{N}\}$$

where $\mathtt{a}^m$ and $\mathtt{b}^n$ respectively denote m and n occurrences of the atoms $\mathtt{a}$ and $\mathtt{b}$.

We also have $[\![\mathtt{a}^*]\!] = \{\langle\mathtt{a}^m\rangle \mid m \in \mathbb{N}\}$ and $[\![\mathtt{b}^*]\!] = \{\langle\mathtt{b}^n\rangle \mid n \in \mathbb{N}\}$, therefore $(\mathtt{a}+\mathtt{b})^*$ and $\mathtt{a}^*\cdot\mathtt{b}^*$ are equivalent patterns. Note the difference between the semantics of patterns and that of regular expressions, where $\cdot$ is not commutative in general.

Exercise 2

1. $(\mathtt{a}\cdot\mathtt{a})^* + \mathtt{b}\cdot(\mathtt{b}\cdot\mathtt{b})^*$
2. $(\mathtt{a}\cdot\mathtt{b}\cdot\mathtt{c})^*$
3. $(\mathtt{a}\cdot\mathtt{a}\cdot\mathtt{b})^*$

Exercise 3

Concerning the first item we have

$$[\![M^*/M]\!] = [\![(M/M)\cdot M^*]\!] = [\![\mathbb{1}\cdot M^*]\!] = [\![\langle\rangle \uplus \langle M^n\rangle \mid n \in \mathbb{N}]\!] = [\![\langle M^n\rangle \mid n \in \mathbb{N}]\!]$$

which is the same semantics of M^*. Concerning the second item, it is enough to take $E \overset{\text{def}}{=} M + \mathtt{a}$. Then $(E/M) = \mathbb{1}$ and $\mathbb{1}\cdot M = M \neq E$.

Solutions to Exercises in Section 5 (Pat)

Quasilinearity Exercises

Exercise 6(a)

```
def ql1(): Unit {
  let mb = new[I] in
  mb ! Msg1();                        # mb : I!Msg1<U>
  guard mb : Msg1 {                   # mb : I?Msg1<R>
    receive Msg1() from mb ->
      free(mb)                        # mb : I?1<R>
  }
}
```

This would typecheck.

Exercise 6(b)

```
def ql2(): Unit {
  let mb = new[I] in
  mb ! Msg1();                          # mb : !Msg1<U>
  guard mb : Msg1 {                     # mb : ?Msg1<R>
    receive Msg1() from self ->
      free(mb);                         # mb   : ?1<R>
      free(self)                        # self : ?1<R>
  }
}
```

This would not typecheck, since there are two returnable occurrences of `mb`: one as the subject of the **`guard`**, and another as the subject of the **`free`** expression in the **`receive`** clause. Remember that the subject of a **`guard`** must always be returnable, and **`free`**`(mb)` is syntactic sugar for **`guard`** `mb : 1 {` **`free`** `->() }`.

Exercise 6(c)

```
def ql3(): Unit {
  let mb = new[I] in
  spawn {
    guard mb : Msg1 {                 # mb : ?Msg1<R>
      receive Msg1() from mb ->
        free(mb)                      # mb : ?1<R>
    }
  };
  mb ! Msg1()                         # mb : !Msg1<U>
}
```

This would typecheck, since the returnable use of `mb` in the body of the **`spawn`** would be masked as second-class.

Exercise 6(d)

```
def ql4(): Unit {
  let mb = new[I];
  mb ! Msg1();                            # mb : !Msg1<U>
  let mb2 = mb in                         # mb : ?Msg1<R>
  mb ! Msg2();                          # mb : !Msg2<U>
  guard mb2 : Msg1 . Msg2 {             # mb2 : ?(Msg1.Msg2)<R>
    receive Msg1() from mb2 ->
      guard mb2 : Msg2 {                # mb2 : ?Msg2<R>
        receive Msg2() from mb2 ->
          free(mb2)                     # mb2 : ?1<R>
      }
  }
}
```

This would not typecheck, since a returnable use of `mb` (in the subject of the **`let`**-expression) occurs before a second-class use (when sending `Msg2` on the folowing line). Remember that the subject of a **`let`**-expression must always be returnable.

Pat Program Exercises

- Exercise 7 (Echo): See `https://github.com/SimonJF/pat-exercise-solutions/blob/main/ex1-echo.pat`
- Exercise 8 (Pinger): See `https://github.com/SimonJF/pat-exercise-solutions/blob/main/ex2-pinger.pat`
- Exercise 9 (Two-Factor): See `https://github.com/SimonJF/pat-exercise-solutions/blob/main/ex3-two-factor-a.pat` and `https://github.com/SimonJF/pat-exercise-solutions/blob/main/ex3-two-factor-b.pat`.