Mailbox Types for Unordered Interactions

Ugo de'Liguoro
Università di Torino, Dipartimento di Informatica, Torino, Italy
deligu@di.unito.it
https://orcid.org/0000-0003-4609-2783

Luca Padovani
Università di Torino, Dipartimento di Informatica, Torino, Italy
luca.padovani@unito.it
https://orcid.org/0000-0001-9097-1297

Abstract

We propose a type system for reasoning on protocol conformance and deadlock freedom in networks of processes that communicate through unordered mailboxes. We model these networks in the mailbox calculus, a mild extension of the asynchronous π-calculus with first-class mailboxes and selective input. The calculus subsumes the actor model and allows us to analyze networks with dynamic topologies and varying number of processes possibly mixing different concurrency abstractions. Well-typed processes are deadlock free and never fail because of unexpected messages. For a non-trivial class of them, junk freedom is also guaranteed. We illustrate the expressiveness of the calculus and of the type system by encoding instances of non-uniform, concurrent objects, binary sessions extended with joins and forks, and some known actor benchmarks.

2012 ACM Subject Classification Theory of computation → Type structures, Theory of computation → Process calculi, Software and its engineering → Concurrent programming structures, Software and its engineering → Message passing

Keywords and phrases actors, concurrent objects, first-class mailboxes, unordered communication protocols, behavioral types, protocol conformance, deadlock freedom, junk freedom

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.15

Related Version Proofs and additional examples are found in the related technical report [17], `https://arxiv.org/abs/1801.04167`.

Acknowledgements The authors are grateful to the anonymous reviewers for their valuable feedback. The second author misses his beloved cat Arturo, who passed away on January 8, 2018. Much of this paper was written next to Arturo during the last weeks of his long illness.

1 Introduction

Message passing is a key mechanism used to coordinate concurrent processes. The order in which a process consumes messages may coincide with the order in which they arrive at destination (*ordered processing*) or may depend on some intrinsic property of the messages themselves, such as their priority, their tag, or the shape of their content (*out-of-order or selective processing*). Ordered message processing is common in networks of processes connected by point-to-point *channels*. Out-of-order message processing is common in networks of processes using *mailboxes*, into which processes concurrently store messages and from which one process selectively receives messages. This communication model is typically found in the various implementations of actors [26, 1] such as Erlang [3], Scala and Akka actors [24], CAF [8] and Kilim [52]. Non-uniform, concurrent objects [50, 48, 15] are also

```scala
1  class Account(var balance: Double) extends ScalaActor[AnyRef] {
2    private val self = this
3    override def process(msg: AnyRef) {
4      msg match {
5        case dm: DebitMessage =>
6          balance += dm.amount
7          val sender = dm.sender.asInstanceOf[Account]
8          sender.send(ReplyMessage.ONLY)
9        case cm: CreditMessage =>
10         balance -= cm.amount
11         val sender    = cm.sender.asInstanceOf[ScalaActor[AnyRef]]
12         val recipient = cm.recipient.asInstanceOf[Account]
13         recipient.send(new DebitMessage(self, cm.amount))
14         receive {
15           case rm: ReplyMessage =>
16             sender.send(ReplyMessage.ONLY)
17         }
18       case _: StopMessage => exit()
19       case message =>
20         val ex = new IllegalArgumentException("Unsupported␣message")
21         ex.printStackTrace(System.err)
22     }
23   }
24 }
```

Listing 1 An example of Scala actor taken from the `Savina` benchmark suite [32].

examples of out-of-order message processors. For example, a busy lock postpones the processing of any `acquire` message until it is released by its current owner. Out-of-order message processing adds further complexity to the challenging task of concurrent and parallel application development: storing a message into the wrong mailbox or at the wrong time, forgetting a message in a mailbox, or relying on the presence of a particular message that is not guaranteed to be found in a mailbox are programming mistakes that are easy to do and hard to detect without adequate support from the language and its development tools.

The Scala actor in Listing 1, taken from the `Savina` benchmark suite [32], allows us to illustrate some of the subtle pitfalls that programmers must carefully avoid when dealing with out-of-order message processing. The `process` method matches messages found in the actor's mailbox according to their type. If a message of type `DebitMessage` is found, then `balance` is incremented by the deposited amount and the actor requesting the operation is notified with a `ReplyMessage` (lines 5–8). If a message of type `CreditMessage` is found, `balance` is decremented by the amount that is transferred to `recipient` (lines 9–13). Since the operation is meant to be atomic, the actor temporarily changes its behavior and waits for a `ReplyMessage` from `recipient` signalling that the transfer is complete, before notifying `sender` in turn (lines 14–17). A message of type `StopMessage` terminates the actor (line 18).

Note how the correct execution of this code depends on some key assumptions:

- `ReplyMessage` should be stored in the actor's mailbox only when the actor is involved in a transaction, or else the message would trigger the "catch all" clause that throws a "unsupported message" exception (lines 19–21).
- No debit or credit message should be in the actor's mailbox by the time it receives `StopMessage`, or else some critical operations affecting the balance would not be performed.

- Two distinct accounts should not try to simultaneously initiate a transaction with each other. If this were allowed, each account could consume the credit message found in its own mailbox and then deadlock waiting for a reply from the other account (lines 14–17).

Static analysis techniques that certify the validity of assumptions like these can be valuable for developers. For example, session types [29] have proved to be an effective formalism for the enforcement of communication protocols and have been applied to a variety of programming paradigms and languages [2], including those based on mailbox communications [41, 7, 19, 43]. However, session types are specifically designed to address point-to-point, ordered interactions over channels [27]. Retrofitting them to a substantially different communication model calls for some inevitable compromises on the network topologies that can be addressed and forces programmers to give up some of the flexibility offered by unordered message processing.

Another aspect that complicates the analysis of actor systems is that the pure actor model as it has been originally conceived [26, 1] does not accurately reflect the actual practice of actor programming. In the pure actor model, each actor owns a single mailbox and the only synchronization mechanism is message reception from such mailbox. However, it is a known fact that the implementation of complex coordination protocols in the pure actor model is challenging [54, 53, 33, 9]. These difficulties have led programmers to mix the actor model with different concurrency abstractions [31, 53], to extend actors with controlled forms of synchronization [54] and to consider actors with multiple/first-class mailboxes [23, 33, 9]. In fact, popular implementations of the actor model feature disguised instances of multiple/first-class mailbox usage, even if they are not explicitly presented as such: in Akka, the messages that an actor is unable to process immediately can be temporarily stashed into a different mailbox [23]; in Erlang, hot code swapping implies transferring at runtime the input capability on a mailbox from a piece of code to a different one [3].

In summary, there is still a considerable gap between the scope of available approaches used to analyze mailbox-based communicating systems and the array of features used in programming these systems. To help narrowing this gap, we make the following contributions:

- We introduce *mailbox types*, a new kind of behavioral types with a simple and intuitive semantics embodying the unordered nature of mailboxes. Mailbox types allow us to describe mailboxes subject to selective message processing as well as mailboxes concurrently accessed by several processes. Incidentally, mailbox types also provide precise information on the size of mailboxes that may lead to valuable code optimizations.
- We develop a mailbox type system for the *mailbox calculus*, a mild extension of the asynchronous π-calculus [51] featuring tagged messages, selective inputs and first-class mailboxes. The mailbox calculus allows us to address a broad range of systems with dynamic topology and varying number of processes possibly using a mixture of concurrency models (including multi-mailbox actors) and abstractions (such as locks and futures).
- We prove three main properties of well-typed processes: the absence of failures due to unexpected messages (*mailbox conformance*); the absence of pending activities and messages in irreducible processes (*deadlock freedom*); for a non-trivial class of processes, the guarantee that every message can be eventually consumed (*junk freedom*).
- We illustrate the expressiveness of mailbox types by presenting well-typed encodings of known concurrent objects (locks and futures) and actor benchmarks (atomic transactions and master-workers parallelism) and of binary sessions extended with forks and joins. In discussing these examples, we emphasize the impact of out-of-order message processing and of first-class mailboxes.

Structure of the paper. We start from the definition of the *mailbox calculus* and of the properties we expect from well-typed processes (Section 2). We introduce mailbox types (Section 3.1) and dependency graphs (Section 3.2) for tracking mailbox dependencies in processes that use more than one. Then, we present the typing rules (Section 3.3) and the soundness results of the type system (Section 3.4). In the latter part of the paper, we discuss a few more complex examples (Section 4), related work (Section 5) and ideas for further developments (Section 6).

2 The Mailbox Calculus

We assume given an infinite set of *variables* x, y, an infinite set of *mailbox names* a, b, a set of *tags* $\mathtt{m}$ and a finite set of *process variables* X. We let u, v range over variables and mailbox names without distinction. Throughout the paper we write $\overline{e}$ for possibly empty sequences $e_1, \dots, e_n$ of various entities. For example, $\overline{u}$ stands for a sequence $u_1, \dots, u_n$ of names and $\{\overline{u}\}$ for the corresponding set.

The syntax of the *mailbox calculus* is shown below:

$$\begin{array}{llcl}
\textbf{Process} & P, Q & ::= & \mathtt{done} \mid u!\mathtt{m}[\overline{v}] \mid G \mid P \mid Q \mid (\nu a)P \mid \mathsf{X}[\overline{u}] \\
\textbf{Guard} & G, H & ::= & \mathtt{fail}\ u \mid \mathtt{free}\ u.P \mid u?\mathtt{m}(\overline{x}).P \mid G + H
\end{array}$$

The term $\mathtt{done}$ represents the terminated process that performs no action. The term $u!\mathtt{m}[\overline{v}]$ represents a message stored in mailbox u. The message has tag $\mathtt{m}$ and arguments $\overline{v}$. A guarded process G is a composition of actions offered on a mailbox. Actions will be described in a moment. We assume that all actions in the same guard refer to the same mailbox u. The term $P \mid Q$ represents the parallel composition of P and Q and $(\nu a)P$ represents a restricted mailbox a with scope P. The term $\mathsf{X}[\overline{u}]$ represents the invocation of the process named X with parameters $\overline{u}$. For each process variable X we assume that there is a corresponding *global process definition* of the form $\mathsf{X}(\overline{x}) \triangleq P$. The action $\mathtt{fail}\ u$ represents the process that fails with an error for having received an unexpected message from mailbox u. The action $\mathtt{free}\ u.P$ represents the process that deletes the mailbox u if u is empty and then continues as P. The action $u?\mathtt{m}(\overline{x}).P$ represents the process that receives an $\mathtt{m}$-tagged message from mailbox u and then continues as P with $\overline{x}$ replaced by the message's arguments. A compound guard $G + H$ offers all the actions offered by G and H. The notions of free and bound names of a process P are standard and respectively denoted by $\mathsf{fn}(P)$ and $\mathsf{bn}(P)$.

The operational semantics of the mailbox calculus is mostly conventional. We use the structural congruence relation $\equiv$ defined below to rearrange equivalent processes:

$$\begin{array}{ccc}
\mathtt{fail}\ a + G \equiv G & G + H \equiv H + G & G + (H + H') \equiv (G + H) + H' \\
\mathtt{done} \mid P \equiv P & P \mid Q \equiv Q \mid P & P \mid (Q \mid R) \equiv (P \mid Q) \mid R \\
 & (\nu a)(\nu b)P \equiv (\nu b)(\nu a)P & (\nu a)P \mid Q \equiv (\nu a)(P \mid Q) \quad \text{if } a \notin \mathsf{fn}(Q)
\end{array}$$

Structural congruence captures the usual commutativity and associativity laws of guard and process composition, with $\mathtt{fail}$ and $\mathtt{done}$ acting as the respective units. Additionally, the order of mailbox restrictions is irrelevant and the scope of a mailbox may shrink or extend dynamically. The reduction relation $\rightarrow$ is inductively defined by the rules

$$\begin{array}{lrl}
\text{[R-READ]} & a!\mathtt{m}[\overline{c}] \mid a?\mathtt{m}(\overline{x}).P + G \rightarrow P\{\overline{c}/\overline{x}\} & \\
\text{[R-FREE]} & (\nu a)(\mathtt{free}\ a.P + G) \rightarrow P & \\
\text{[R-DEF]} & \mathsf{X}[\overline{c}] \rightarrow P\{\overline{c}/\overline{x}\} & \text{if } \mathsf{X}(\overline{x}) \triangleq P \\
\text{[R-PAR]} & P \mid R \rightarrow Q \mid R & \text{if } P \rightarrow Q \\
\text{[R-NEW]} & (\nu a)P \rightarrow (\nu a)Q & \text{if } P \rightarrow Q \\
\text{[R-STRUCT]} & P \rightarrow Q & \text{if } P \equiv P' \rightarrow Q' \equiv Q
\end{array}$$

where $P\{\overline{c}/\overline{x}\}$ denotes the usual capture-avoiding replacement of the variables $\overline{x}$ with the mailbox names $\overline{c}$. Rule [R-READ] models the selective reception of an $\mathtt{m}$-tagged message from mailbox a, which erases all the other actions of the guard. Rule [R-FREE] is triggered when the process is ready to delete the empty mailbox a and no more messages can be stored in a because there are no other processes in the scope of a. Rule [R-DEF] models a process invocation by replacing the process variable X with the corresponding definition. Finally, rules [R-PAR], [R-NEW] and [R-STRUCT] close reductions under parallel compositions, name restrictions and structural congruence. We write $\rightarrow^*$ for the reflexive and transitive closure of $\rightarrow$, we write $P \not\rightarrow^* Q$ if not $P \rightarrow^* Q$ and $P \not\rightarrow$ if $P \not\rightarrow Q$ for all Q.

Hereafter, we will occasionally use numbers and conditionals in processes. These and other features can be either encoded or added to the calculus without difficulties.

▶ **Example 1** (lock). In this example we model a lock as a process that waits for messages from a *self* mailbox in which acquisition and release requests are stored. The lock is either free or busy. When in state free, the lock nondeterministically consumes an `acquire` message from *self*. This message indicates the willingness to acquire the lock by another process and carries a reference to a mailbox into which the lock stores a `reply` notification. When in state busy, the lock waits for a `release` message indicating that it is being released:

$$\begin{aligned} \mathsf{FreeLock}(\mathit{self}) &\triangleq \mathtt{free}\ \mathit{self}\,.\mathtt{done} \\ &\quad + \mathit{self}?\mathtt{acquire}(\mathit{owner})\,.\,\mathsf{BusyLock}[\mathit{self}, \mathit{owner}] \\ &\quad + \mathit{self}?\mathtt{release}\,.\,\mathtt{fail}\ \mathit{self} \\ \mathsf{BusyLock}(\mathit{self}, \mathit{owner}) &\triangleq \mathit{owner}!\mathtt{reply}[\mathit{self}] \mid \mathit{self}?\mathtt{release}\,.\,\mathsf{FreeLock}[\mathit{self}] \end{aligned}$$

Note the presence of the `free` *self* guard in the definition of FreeLock and the lack thereof in BusyLock. In the former case, the lock manifests the possibility that no process is willing to acquire the lock, in which case it deletes the mailbox and terminates. In the latter case, the lock manifests its expectation to be eventually released. Also note that FreeLock fails if it receives a `release` message. In this way, the lock manifests the fact that it can be released only if it is currently owned by a process. A system where two users *alice* and *carol* compete for acquiring *lock* can be modeled as the process

$$(\nu\mathit{lock})(\nu\mathit{alice})(\nu\mathit{carol})(\mathsf{FreeLock}[\mathit{lock}] \mid \mathsf{User}[\mathit{alice}, \mathit{lock}] \mid \mathsf{User}[\mathit{carol}, \mathit{lock}]) \tag{1}$$

where

$$\mathsf{User}(\mathit{self}, \mathit{lock}) \triangleq \mathit{lock}!\mathtt{acquire}[\mathit{self}] \mid \mathit{self}?\mathtt{reply}(l)\,.\,(l!\mathtt{release} \mid \mathtt{free}\ \mathit{self}\,.\mathtt{done})$$

Note that User uses the reference l – as opposed to *lock* – to release the acquired lock. As we will see in Section 3.3, this is due to the fact that it is this particular reference to the lock's mailbox – and not *lock* itself – that carries the capability to release the lock.

▶ **Example 2** (future variable). A future variable is a one-place buffer that stores the result of an asynchronous computation. The content of the future variable is *resolved* once and for all by the producer once the computation completes. After that, its content can be retrieved any number of times by the consumers. If a consumer attempts to retrieve the content of the future variable beforehand, the consumer suspends until the variable is resolved. We can model a future variable thus:

$$\begin{aligned} \mathsf{Future}(\mathit{self}) &\triangleq \mathit{self}?\mathtt{put}(x)\,.\,\mathsf{Present}[\mathit{self}, x] \\ \mathsf{Present}(\mathit{self}, x) &\triangleq \mathtt{free}\ \mathit{self}\,.\mathtt{done} \\ &\quad + \mathit{self}?\mathtt{get}(\mathit{sender})\,.\,(\mathit{sender}!\mathtt{reply}[x] \mid \mathsf{Present}[\mathit{self}, x]) \\ &\quad + \mathit{self}?\mathtt{put}\,.\,\mathtt{fail}\ \mathit{self} \end{aligned}$$

The process Future represents an unresolved future variable, which waits for a put message from the producer. Once the variable has been resolved, it behaves as specified by Present, namely it satisfies an arbitrary number of get messages from consumers but it no longer accepts put messages.

▶ **Example 3** (bank account). Below we see the process definition corresponding to the actor shown in Listing 1. The structure of the term follows closely that of the Scala code:

$$\begin{array}{rcl}
\mathsf{Account}(\mathit{self}, \mathit{balance}) & \triangleq & \mathit{self}?\mathtt{debit}(\mathit{amount}, \mathit{sender})\,.\\
& & (\mathit{sender}!\mathtt{reply} \mid \mathsf{Account}[\mathit{self}, \mathit{balance} + \mathit{amount}])\\
& + & \mathit{self}?\mathtt{credit}(\mathit{amount}, \mathit{recipient}, \mathit{sender})\,.\\
& & (\mathit{recipient}!\mathtt{debit}[\mathit{amount}, \mathit{self}] \mid\\
& & \;\mathit{self}?\mathtt{reply}.(\mathit{sender}!\mathtt{reply} \mid \mathsf{Account}[\mathit{self}, \mathit{balance} + \mathit{amount}]))\\
& + & \mathit{self}?\mathtt{stop}.\mathtt{free}\ \mathit{self}.\mathtt{done}\\
& + & \mathit{self}?\mathtt{reply}.\mathtt{fail}\ \mathit{self}
\end{array}$$

The last term of the guarded process, which results in a failure, corresponds to the catch-all clause in Listing 1 and models the fact that a reply message is not expected to be found in the account's mailbox unless the account is involved in a transaction. The reply message is received and handled appropriately in the credit-guarded term.

We can model a deadlock in the case two distinct bank accounts attempt to initiate a transaction with one another. Indeed, we have

$$\begin{array}{l}
\mathsf{Account}[\mathit{alice}, 8] \mid \mathit{alice}!\mathtt{credit}[2, \mathit{carol}, \mathit{bank}] \mid\\
\mathsf{Account}[\mathit{carol}, 9] \mid \mathit{carol}!\mathtt{credit}[5, \mathit{alice}, \mathit{bank}]
\end{array}
\rightarrow^*
\begin{array}{l}
\mathit{carol}!\mathtt{debit}[2, \mathit{alice}] \mid \mathit{alice}?\mathtt{reply}\dots \mid\\
\mathit{alice}!\mathtt{debit}[5, \mathit{carol}] \mid \mathit{carol}?\mathtt{reply}\dots
\end{array}$$

where both *alice* and *carol* ignore the incoming debit messages, whence the deadlock.

We now provide operational characterizations of the properties enforced by our typing discipline. We begin with mailbox conformance, namely the property that a process never fails because of unexpected messages. To this aim, we define a process context $\mathscr{C}$ as a process in which there is a single occurrence of an *unguarded hole* [ ]:

$$\mathscr{C} \quad ::= \quad [\,] \quad | \quad \mathscr{C} \mid P \quad | \quad P \mid \mathscr{C} \quad | \quad (\nu a)\mathscr{C}$$

The hole is "unguarded" in the sense that it does not occur prefixed by an action. As usual, we write $\mathscr{C}[P]$ for the process obtained by replacing the hole in $\mathscr{C}$ with P. Names may be captured by this replacement. A mailbox conformant process never reduces to a state in which the only action of a guard is fail:

▶ **Definition 4.** We say that P is *mailbox conformant* if $P \not\rightarrow^* \mathscr{C}[\mathtt{fail}\ a]$ for all $\mathscr{C}$ and a.

Looking at the placement of the fail u actions in earlier examples we can give the following interpretations of mailbox conformance: a lock is never released unless it has been acquired beforehand (Example 1); a future variable is never resolved twice (Example 2); an account will not be notified of a completed transaction (with a reply message) unless it is involved in an ongoing transaction (Example 3).

We express deadlock freedom as the property that all irreducible residuals of a process are (structurally equivalent to) the terminated process:

▶ **Definition 5.** We say that P is *deadlock free* if $P \rightarrow^* Q \not\rightarrow$ implies $Q \equiv \mathtt{done}$.

According to Definition 5, if a deadlock-free process halts we have that: (1) there is no sub-process waiting for a message that is never produced; (2) every mailbox is empty. Clearly, this is not the case for the transaction between *alice* and *carol* in Example 3.

▶ **Example 6** (deadlock)**.** Below is another example of deadlocking process using Future from Example 2, obtained by resolving a future variable with the value it does not contain yet:

$$(\nu f)(\nu c)(\mathsf{Future}[f] \mid f!\mathtt{get}[c] \mid c?\mathtt{reply}(x).\mathtt{free}\ c.f!\mathtt{put}[x]) \tag{2}$$

Notice that attempting to retrieve the content of a future variable not knowing whether it has been resolved is legal. Indeed, Future does not fail if a get message is present in the future variable's mailbox before it is resolved. Thus, the deadlocked process above is mailbox conformant but also an instance of undesirable process that will be ruled out by our static analysis technique (*cf.* Example 21). We will need dependency graphs in addition to types to flag this process as ill typed.

A property stronger than deadlock freedom is fair termination. A fairly terminating process is a process whose residuals always have the possibility to terminate. Formally:

▶ **Definition 7.** We say that P is *fairly terminating* if $P \to^* Q$ implies $Q \to^* \mathtt{done}$.

An interesting consequence of fair termination is that it implies *junk freedom* (also known as lock freedom [34, 44]) namely the property that every message can be eventually consumed. Our type system does not guarantee fair termination nor junk freedom in general, but it does so for a non-trivial sub-class of well-typed processes that we characterize later on.

3 A Mailbox Type System

In this section we detail the type system for the mailbox calculus. We start from the syntax and semantics of mailbox types (Section 3.1) and of dependency graphs (Section 3.2), the mechanism we use to track mailbox dependencies. Then we present the typing rules (Section 3.3) and the properties of well-typed processes (Section 3.4).

3.1 Mailbox Types

The syntax of mailbox types and patterns is shown below:

$$\begin{array}{rrcl} \textbf{Mailbox Type} & \tau,\sigma & ::= & ?E \mid !E \\ \textbf{Pattern} & E,F & ::= & \mathbb{0} \mid \mathbb{1} \mid \mathtt{m}[\overline{\tau}] \mid E+F \mid E\cdot F \mid E^* \end{array} \tag{3}$$

Patterns are *commutative regular expressions* [11] describing the configurations of messages stored in a mailbox. An atom $\mathtt{m}[\overline{\tau}]$ describes a mailbox containing a single message with tag $\mathtt{m}$ and arguments of type $\overline{\tau}$. We let M range over atoms and abbreviate $\mathtt{m}[\overline{\tau}]$ with $\mathtt{m}$ when $\overline{\tau}$ is the empty sequence. Compound patterns are built using sum ($E + F$), product ($E \cdot F$) and exponential (E^*). The constants $\mathbb{1}$ and $\mathbb{0}$ respectively describe the empty and the unreliable mailbox. There is no configuration of messages stored in an unreliable mailbox, not even the empty one. We will use the $\mathbb{0}$ pattern for describing mailboxes from which an unexpected message has been received. Let us look at a few simple examples. The pattern $\mathtt{A} + \mathtt{B}$ describes a mailbox that contains either an $\mathtt{A}$ message or a $\mathtt{B}$ message, but not both, whereas the pattern $\mathtt{A} + \mathbb{1}$ describes a mailbox that either contains a $\mathtt{A}$ message or is empty. The pattern $\mathtt{A} \cdot \mathtt{B}$ describes a mailbox that contains both an $\mathtt{A}$ message and also a $\mathtt{B}$ message. Note that $\mathtt{A}$ and $\mathtt{B}$ may be equal, in which case the mailbox contains *two* $\mathtt{A}$ messages. Finally, the pattern $\mathtt{A}^*$ describes a mailbox that contains an arbitrary number (possibly zero) of $\mathtt{A}$ messages. We adopt the usual conventions on the priority of connectives, whereby * binds stronger than $\cdot$ which, in turn, binds stronger than $+$.

A *mailbox type* consists of a *capability* (either ? or !) paired with a pattern. The capability specifies whether the pattern describes messages to be received from (?) or stored in (!) the mailbox. Here are some examples: A process using a mailbox of type $\mathtt{!A}$ *must* store an $\mathtt{A}$ message into the mailbox, whereas a process using a mailbox of type $\mathtt{?A}$ expects to receive an $\mathtt{A}$ message from the mailbox. A process using a mailbox of type $\mathtt{!}(\mathtt{A}+\mathbb{1})$ *may* store an $\mathtt{A}$ message into the mailbox, but is not obliged to do so. A process using a mailbox of type $\mathtt{!}(\mathtt{A}+\mathtt{B})$ decides whether to store an $\mathtt{A}$ message or a $\mathtt{B}$ message in the mailbox, whereas a process using a mailbox of type $\mathtt{?}(\mathtt{A}+\mathtt{B})$ must be ready to receive both kinds of messages. A process using a mailbox of type $\mathtt{?}(\mathtt{A}\cdot\mathtt{B})$ expects to receive both an $\mathtt{A}$ message and a $\mathtt{B}$ message and may decide in which order to do so. A process using a mailbox of type $\mathtt{!}(\mathtt{A}\cdot\mathtt{B})$ must store both $\mathtt{A}$ and $\mathtt{B}$ into the mailbox. A process using a mailbox of type $\mathtt{!A}^*$ decides how many $\mathtt{A}$ messages to store in the mailbox, whereas a process using a mailbox of type $\mathtt{?A}^*$ must be prepared to receive an arbitrary number of $\mathtt{A}$ messages.

To cope with possibly infinite types we interpret the productions in (3) coinductively and consider as types the regular trees [13] built using those productions. We require every infinite branch of a type tree to go through infinitely many atoms. This strengthened contractiveness condition allows us to define functions inductively on the structure of patterns, provided that these functions do not recur into argument types (*cf.* Definitions 8 and 14).

The semantics of a pattern is a *set of multisets of atoms*. Because patterns include types, the given semantics is parametric in the subtyping relation, which will be defined next:

▶ **Definition 8** (subpattern). The *configurations* of E are inductively defined by the following equations, where A and B range over multisets $\langle\overline{M}\rangle$ of atoms and $\uplus$ denotes multiset union:

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

Given a preorder relation $\mathscr{R}$ on types, we write $E \sqsubseteq_{\mathscr{R}} F$ if $\langle\mathtt{m}_i[\overline{\tau}_i]\rangle_{i\in I}\in[\![E]\!]$ implies $\langle\mathtt{m}_i[\overline{\sigma}_i]\rangle_{i\in I}\in[\![F]\!]$ and $\overline{\tau}_i \mathrel{\mathscr{R}} \overline{\sigma}_i$ for every $i\in I$. We write $\simeq_{\mathscr{R}}$ for $\sqsubseteq_{\mathscr{R}}\cap\sqsupseteq_{\mathscr{R}}$.

For example, $[\![\mathtt{A}+\mathtt{B}]\!]=\{\langle\mathtt{A}\rangle,\langle\mathtt{B}\rangle\}$ and $[\![\mathtt{A}\cdot\mathtt{B}]\!]=\{\langle\mathtt{A},\mathtt{B}\rangle\}$. It is easy to see that $\sqsubseteq_{\mathscr{R}}$ is a pre-congruence with respect to all the connectives and that $\simeq_{\mathscr{R}}$ includes all the known laws of commutative Kleene algebra [11]: both $+$ and $\cdot$ are commutative and associative, $+$ is idempotent and has unit $\mathbb{0}$, $\cdot$ distributes over $+$, it has unit $\mathbb{1}$ and is absorbed by $\mathbb{0}$. Also observe that $\sqsubseteq_{\mathscr{R}}$ is related covariantly to $\mathscr{R}$, that is $\overline{\tau}\mathrel{\mathscr{R}}\overline{\sigma}$ implies $\mathtt{m}[\overline{\tau}]\sqsubseteq_{\mathscr{R}}\mathtt{m}[\overline{\sigma}]$.

We now define subtyping. As types may be infinite, we resort to coinduction:

▶ **Definition 9** (subtyping). We say that $\mathscr{R}$ is a *subtyping relation* if $\tau\mathrel{\mathscr{R}}\sigma$ implies either

1. $\tau = \mathtt{?}E$ and $\sigma=\mathtt{?}F$ and $E\sqsubseteq_{\mathscr{R}} F$, or
2. $\tau = \mathtt{!}E$ and $\sigma=\mathtt{!}F$ and $F\sqsubseteq_{\mathscr{R}} E$.

We write $\leqslant$ for the largest subtyping relation and say that τ is a *subtype* of σ (and σ a *supertype* of τ) if $\tau\leqslant\sigma$. We write $\lesseqgtr$ for $\leqslant\cap\geqslant$, $\sqsubseteq$ for $\sqsubseteq_{\leqslant}$ and $\simeq$ for $\simeq_{\leqslant}$.

Items 1 and 2 respectively correspond to the usual covariant and contravariant rules for channel types with input and output capabilities [47]. For example, $\mathtt{!}(\mathtt{A}+\mathtt{B})\leqslant\mathtt{!A}$ because a mailbox of type $\mathtt{!}(\mathtt{A}+\mathtt{B})$ is more permissive than a mailbox of type $\mathtt{!A}$. Dually, $\mathtt{?A}\leqslant\mathtt{?}(\mathtt{A}+\mathtt{B})$ because a mailbox of type $\mathtt{?A}$ provides stronger guarantees than a mailbox of type $\mathtt{?}(\mathtt{A}+\mathtt{B})$. Note that $\mathtt{!}(\mathtt{A}\cdot\mathtt{B})\lesseqgtr\mathtt{!}(\mathtt{B}\cdot\mathtt{A})$ and $\mathtt{?}(\mathtt{A}\cdot\mathtt{B})\lesseqgtr\mathtt{?}(\mathtt{B}\cdot\mathtt{A})$, to witness the fact that the order in which messages are stored in a mailbox is irrelevant.

Mailbox types whose patterns are in particular relations with the constants $\mathbb{0}$ and $\mathbb{1}$ will play special roles, so we introduce some corresponding terminology.

▶ **Definition 10** (type and name classification). We say that (a name whose type is) τ is:

- *relevant* if $\tau \not\leqslant \texttt{!}\mathbb{1}$ and *irrelevant* otherwise;
- *reliable* if $\tau \not\leqslant \texttt{?}\mathbb{0}$ and *unreliable* otherwise;
- *usable* if $\texttt{!}\mathbb{0} \not\leqslant \tau$ and *unusable* otherwise.

A relevant name *must* be used, whereas an irrelevant name may be discarded because not storing any message in the mailbox it refers to is allowed by its type. All mailbox types with input capability are relevant. A reliable mailbox is one from which no unexpected message has been received. All names with output capability are reliable. A usable name *can* be used, in the sense that there exists a construct of the mailbox calculus that expects a name with that type. All mailbox types with input capability are usable, but $\texttt{?}(\texttt{A} \cdot \mathbb{0})$ is unreliable. Both $\texttt{!A}$ and $\texttt{!}(\mathbb{1} + \texttt{A})$ are usable. The former type is also relevant because a process using a mailbox with this type must (eventually) store an $\texttt{A}$ message in it. On the contrary, the latter type is irrelevant, since not using the mailbox is a legal way of using it.

Henceforth we assume that all types are usable and that all argument types are also reliable. That is, we ban all types like $\texttt{!}\mathbb{0}$ or $\texttt{!}(\mathbb{0} \cdot \texttt{m})$ and all types like $\texttt{?m}[\texttt{?}\mathbb{0}]$ or $\texttt{!m}[\texttt{?}\mathbb{0}]$.

▶ **Example 11** (lock type). The mailbox used by the lock (Example 1) will have several different types, depending on the viewpoint we take (either the lock itself or one of its users) and on the state of the lock (whether it is free or busy). As we can see from the definition of FreeLock, a free lock waits for an `acquire` message which is supposed to carry a reference to another mailbox into which the capability to release the lock is stored. Since the lock is meant to have several concurrent users, it is not possible in general to predict the number of `acquire` messages in its mailbox. Therefore, the mailbox of a free lock has type

$$\texttt{?acquire}[\texttt{!reply}[\texttt{!release}]]^*$$

from the viewpoint of the lock itself. When the lock is busy, it expects to find one `release` message in its mailbox, but in general the mailbox will also contain `acquire` messages corresponding to pending acquisition requests. So, the mailbox of a busy lock has type

$$\texttt{?}(\texttt{release} \cdot \texttt{acquire}[\texttt{!reply}[\texttt{!release}]]^*)$$

indicating that the mailbox contains (or will eventually contain) a single `release` message along with arbitrarily many `acquire` messages.

Prospective owners of the lock may have references to the lock's mailbox with type $\texttt{!acquire}[\texttt{!reply}[\texttt{!release}]]$ or $\texttt{!acquire}[\texttt{!reply}[\texttt{!release}]]^*$ depending on whether they acquire the lock exactly once (just like *alice* and *carol* in Example 1) or several times. Other intermediate types are possible in the case of users that acquire the lock a bounded number of times. The current owner of the lock will have a reference to the lock's mailbox of type `!release`. This type is relevant, implying that the owner must eventually release the lock.

3.2 Dependency Graphs

We use *dependency graphs* for tracking dependencies between mailboxes. Intuitively, there is a dependency between u and v if either v is the argument of a message in mailbox u or v occurs in the continuation of a process waiting for a message from u. Dependency graphs have names as vertices and undirected edges. However, the usual representation of graphs does not account for the fact that mailbox names may be restricted and that the multiplicity of dependencies matters. Therefore, we define dependency graphs using the syntax below:

$$\textbf{Dependency Graph} \qquad \varphi, \psi \quad ::= \quad \emptyset \quad | \quad \{u, v\} \quad | \quad \varphi \sqcup \psi \quad | \quad (\nu a)\varphi$$

$$\{u,v\} \xrightarrow{u-v} \emptyset \; \text{[G-AXIOM]} \qquad \frac{\varphi \xrightarrow{u-v} \varphi'}{\varphi \sqcup \psi \xrightarrow{u-v} \varphi' \sqcup \psi} \; \text{[G-LEFT]} \qquad \frac{\psi \xrightarrow{u-v} \psi'}{\varphi \sqcup \psi \xrightarrow{u-v} \varphi \sqcup \psi'} \; \text{[G-RIGHT]}$$

$$\frac{\varphi \xrightarrow{u-v} \psi \qquad a \neq u, v}{(\nu a)\varphi \xrightarrow{u-v} (\nu a)\psi} \; \text{[G-NEW]} \qquad \frac{\varphi \xrightarrow{u-w} \psi \qquad \psi \xrightarrow{w-v} \varphi'}{\varphi \xrightarrow{u-v} \varphi'} \; \text{[G-TRANS]}$$

Figure 1 Labelled transitions of dependency graphs.

The term $\emptyset$ represents the empty graph which has no vertices and no edges. The unordered pair $\{u, v\}$ represents the graph made of a single edge connecting the vertices u and v. The term $\varphi \sqcup \psi$ represents the union of φ and ψ whereas $(\nu a)\varphi$ represents the same graph as φ except that the vertex a is restricted. The usual notions of free and bound names apply to dependency graphs. We write $\mathsf{fn}(\varphi)$ for the free names of φ.

To define the semantics of a dependency graph we use the labelled transition system of Table 1. A label $u - v$ represents a path connecting u with v. So, a relation $\varphi \xrightarrow{u-v} \varphi'$ means that u and v are connected in φ and φ' describes the residual edges of φ that have not been used for building the path between u and v. The paths of φ are built from the edges of φ (*cf.* [G-AXIOM]) connected by shared vertices (*cf.* [G-TRANS]). Restricted names cannot be observed in labels, but they may contribute in building paths in the graph (*cf.* [G-NEW]).

▶ **Definition 12** (graph acyclicity and entailment)**.** Let $\mathsf{dep}(\varphi) \overset{\text{def}}{=} \{(u, v) \mid \exists\varphi' : \varphi \xrightarrow{u-v} \varphi'\}$ be the *dependency relation* generated by φ. We say that φ is *acyclic* if $\mathsf{dep}(\varphi)$ is irreflexive. We say that φ *entails* ψ, written $\varphi \Rightarrow \psi$, if $\mathsf{dep}(\psi) \subseteq \mathsf{dep}(\varphi)$.

Note that $\sqcup$ is commutative, associative and has $\emptyset$ as unit with respect to $\mathsf{dep}(\cdot)$. These properties of dependency graphs are key to prove that typing is preserved by structural congruence on processes. Note also that $\sqcup$ is *not* idempotent. Indeed, $\{u, v\} \sqcup \{u, v\}$ is cyclic whereas $\{u, v\}$ is not. The following example motivates the reason why the multiplicity of dependencies is important.

▶ **Example 13** (multiplicity of dependencies)**.** Even though we have not presented the typing rules yet, we can use the above intuition on how dependencies are established to argue that the process below yields a cyclic dependency graph:

$$P \overset{\text{def}}{=} (\nu a)(\nu b)(a!\mathtt{A}[b] \mid a!\mathtt{B}[b] \mid a?\mathtt{A}(x).a?\mathtt{B}(y).\mathtt{free}\ a.x!\mathtt{m}[y]) \rightarrow^* (\nu b)b!\mathtt{m}[b] \not\rightarrow$$

Observe that P stores two messages in the mailbox a, each containing a reference to the mailbox b. Thus, the same dependency $\{a, b\}$ arises twice, resulting in a cyclic dependency involving a and b. By reducing the process, we note that the two variables x and y, which were syntactically different in P, are unified into b in the reduct, which is deadlocked.

3.3 Typing Rules

We use *type environments* for tracking the type of free names occurring in processes. A type environment is a partial function from names to types written as $\overline{u} : \overline{\tau}$ or $u_1 : \tau_1, \dots, u_n : \tau_n$. We let Γ and Δ range over type environments, we write $\mathsf{dom}(\Gamma)$ for the domain of Γ and Γ, Δ for the union of Γ and Δ when $\mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Delta) = \emptyset$. We say that Γ is reliable if so are all the types in its range.

Typing rules for processes $\boxed{\Gamma \vdash P :: \varphi}$

$$\frac{}{\emptyset \vdash \texttt{done} :: \emptyset}\ \text{[T-DONE]} \qquad \frac{\mathsf{X} : (\overline{x} : \overline{\tau}; \varphi)}{\overline{u} : \overline{\tau} \vdash \mathsf{X}[\overline{u}] :: \varphi\{\overline{u}/\overline{x}\}}\ \text{[T-DEF]} \qquad \frac{\Gamma, a : ?\mathbb{1} \vdash P :: \varphi}{\Gamma \vdash (\nu a)P :: (\nu a)\varphi}\ \text{[T-NEW]}$$

$$\frac{}{u : !\mathtt{m}[\overline{\tau}], \overline{v} : \overline{\tau} \vdash u!\mathtt{m}[\overline{v}] :: \{u, \{\overline{v}\}\}}\ \text{[T-MSG]} \qquad \frac{u : ?E, \Gamma \vdash G \qquad \models E}{u : ?E, \Gamma \vdash G :: \{u, \mathsf{dom}(\Gamma)\}}\ \text{[T-GUARD]}$$

$$\frac{\Gamma_i \vdash P_i :: \varphi_i\ ^{(i=1,2)}}{\Gamma_1 \parallel \Gamma_2 \vdash P_1 \mid P_2 :: \varphi_1 \sqcup \varphi_2}\ \text{[T-PAR]} \qquad \frac{\Delta \vdash P :: \psi \qquad \Gamma \leqslant \Delta \qquad \varphi \Rightarrow \psi}{\Gamma \vdash P :: \varphi}\ \text{[T-SUB]}$$

Typing rules for guards $\boxed{\Gamma \vdash G}$

$$\frac{}{u : ?\mathbb{0}, \Gamma \vdash \texttt{fail}\ u}\ \text{[T-FAIL]} \qquad \frac{\Gamma \vdash P :: \varphi}{u : ?\mathbb{1}, \Gamma \vdash \texttt{free}\ u\,.\,P}\ \text{[T-FREE]}$$

$$\frac{u : ?E, \Gamma, \overline{x} : \overline{\tau} \vdash P :: \varphi}{u : ?(\mathtt{m}[\overline{\tau}] \cdot E), \Gamma \vdash u?\mathtt{m}(\overline{x})\,.\,P}\ \text{[T-IN]} \qquad \frac{u : ?E_i, \Gamma \vdash G_i\ ^{(i=1,2)}}{u : ?(E_1 + E_2), \Gamma \vdash G_1 + G_2}\ \text{[T-BRANCH]}$$

Figure 2 Typing rules.

Judgments for processes have the form $\Gamma \vdash P :: \varphi$, meaning that P is well typed in Γ and yields the dependency graph φ. Judgments for guards have the form $\Gamma \vdash G$, meaning that G is well typed in Γ. We say that a judgment $\Gamma \vdash P :: \varphi$ is well formed if $\mathsf{fn}(\varphi) \subseteq \mathsf{dom}(\Gamma)$ and φ is acyclic. Each process typing rule has an implicit side condition requiring that its conclusion be well formed. For each global process definition $\mathsf{X}(\overline{x}) \triangleq P$ we assume that there is a corresponding *global process declaration* of the form $\mathsf{X} : (\overline{x} : \overline{\tau}; \varphi)$. We say that the definition is *consistent* with the corresponding declaration if $\overline{x} : \overline{\tau} \vdash P :: \varphi$, where we require the dependency graph yielded by P to be the same φ associated with the definition. Hereafter, all process definitions are assumed to be consistent. We now discuss the typing rules in detail, introducing auxiliary notions and notation as we go along.

Terminated process. According to the rule [T-DONE], the terminated process `done` is well typed in the empty type environment and yields no dependencies. This is motivated by the fact that `done` does not use any mailbox. Later on we will introduce a subsumption rule [T-SUB] that allows us to type `done` in any type environment with irrelevant names.

Message. Rule [T-MSG] establishes that a message $u!\mathtt{m}[\overline{v}]$ is well typed provided that the mailbox u allows the storing of an $\mathtt{m}$-tagged message with arguments of type $\overline{\tau}$ and the types of $\overline{v}$ are indeed $\overline{\tau}$. The subsumption rule [T-SUB] will make it possible to use arguments whose type is a *subtype* of the expected ones. A message $u!\mathtt{m}[\overline{v}]$ establishes dependencies between the target mailbox u and all of the arguments $\overline{v}$. We write $\{u, \{v_1, \dots, v_n\}\}$ for the dependency graph $\{u, v_1\} \sqcup \dots \sqcup \{u, v_n\}$ and use $\emptyset$ for the empty graph union.

Names with output capability are introduced by the $\parallel$ operator that combines type environments and that will be defined later on, when discussing rule [T-PAR].

Process invocation. The typing rule for a process invocation $\mathsf{X}[\overline{u}]$ checks that there exists a global definition for X which expects exactly the given number and type of parameters. Again, rule [T-SUB] will make it possible to use parameters whose types are subtypes of the expected ones. A process invocation yields the same dependencies as the corresponding process definition, with the appropriate substitutions applied.

Guards. Guards are used to match the content of a mailbox and retrieve messages from it. According to rule [T-FAIL], the action $\mathtt{fail}\ u$ matches a mailbox u with type $\mathtt{?}\mathbb{0}$, indicating that an unexpected message has been found in the mailbox. The type environment may contain arbitrary associations, since the $\mathtt{fail}\ u$ action causes a runtime error. Rule [T-FREE] states that the action $\mathtt{free}\ u.P$ matches a mailbox u with type $\mathtt{?}\mathbb{1}$, indicating that the mailbox is empty. The continuation is well typed in the residual type environment Γ. An input action $u\mathtt{?m}(\overline{x}).P$ matches a mailbox u with type $\mathtt{?}(\mathtt{m}[\overline{\tau}] \cdot E)$ that guarantees the presence of an $\mathtt{m}$-tagged message possibly along with other messages as specified by E. The continuation P must be well typed in an environment where the mailbox has type $\mathtt{?}E$, which describes the content of the mailbox after the $\mathtt{m}$-tagged message has been removed. Associations for the received arguments $\overline{x}$ are also added to the type environment. A compound guard $G_1 + G_2$ offers the actions offered by G_1 and G_2 and therefore matches a mailbox u with type $\mathtt{?}(E_1 + E_2)$, where E_i is the pattern that describes the mailbox matched by G_i. Note that the residual type environment Γ is the same in both branches, indicating that the type of other mailboxes used by the guard cannot depend on that of u.

The judgments for guards do not yield any dependency graph. This is compensated by the rule [T-GUARD], which we describe next.

Guarded processes. Rule [T-GUARD] is used to type a guarded process G, which matches some mailbox u of type $\mathtt{?}E$ and possibly retrieves messages from it. As we have seen while discussing guards, E is supposed to be a pattern of the form $E_1 + \cdots + E_n$ where each E_i is either $\mathbb{0}$, $\mathbb{1}$ or of the form $M \cdot F$. However, only the patterns E that are in *normal form* (defined below) are suitable to be used in this typing rule and the side condition $\models E$ checks that this is indeed the case. Before defining the notion of pattern normal form we motivate its need by means of a simple example.

Suppose that our aim is to type a process $u\mathtt{?A}.P + u\mathtt{?B}.Q$ that consumes either an $\mathtt{A}$ message or a $\mathtt{B}$ message from u, whichever of these two messages is matched first in u, and then continues as P or Q correspondingly. Suppose also that the type of u is $\mathtt{?}E$ with $E \overset{\text{def}}{=} \mathtt{A} \cdot \mathtt{C} + \mathtt{B} \cdot \mathtt{A}$, which allows the rules for guards to successfully type check the process. As we have seen while discussing rule [T-IN], P and Q must be typed in an environment where the type of u has been updated so as to reflect the fact that the consumed message is no longer in the mailbox. In this particular case, we might be tempted to infer that the type of u in P is $\mathtt{?C}$ and that the type of u in Q is $\mathtt{?A}$. Unfortunately, the type $\mathtt{?C}$ does not accurately describe the content of the mailbox after $\mathtt{A}$ has been consumed because, according to E, the $\mathtt{A}$ message may be accompanied by *either* a $\mathtt{B}$ message *or* by a $\mathtt{C}$ message, whereas $\mathtt{?C}$ only accounts for the second possibility. Thus, the appropriate pattern to be used for typing this process is $\mathtt{A} \cdot (\mathtt{B} + \mathtt{C}) + \mathtt{B} \cdot \mathtt{A}$, where the fact that $\mathtt{B}$ may be found after consuming $\mathtt{A}$ is made explicit. This pattern and E are equivalent as they generate exactly the same set of valid configurations. Yet, $\mathtt{A} \cdot (\mathtt{B} + \mathtt{C}) + \mathtt{B} \cdot \mathtt{A}$ is in normal form whereas E is not. In general the normal form is not unique. For example, also the patterns $\mathtt{B} \cdot \mathtt{A} + \mathtt{C} \cdot \mathtt{A}$ and $\mathtt{A} \cdot (\mathtt{B} + \mathtt{C})$ are in normal form and equivalent to E and can be used for typing processes that consume messages from u in different orders or with different priorities.

The first ingredient for defining the notion of pattern normal form is that of pattern residual E/M, which describes the content of a mailbox that initially contains a configuration of messages described by E and from which we remove a single message with type M:

▶ **Definition 14** (pattern residual)**.** The *residual* of a pattern E with respect to an atom M, written E/M, is inductively defined by the following equations:

$$\begin{array}{lll} \mathbb{0}/M = \mathbb{1}/M \stackrel{\text{def}}{=} \mathbb{0} & \mathtt{m}[\overline{\tau}]/\mathtt{m}[\overline{\sigma}] \stackrel{\text{def}}{=} \mathbb{1} \quad \text{if } \overline{\tau} \leqslant \overline{\sigma} & (E+F)/M \stackrel{\text{def}}{=} E/M + F/M \\ (E^*)/M \stackrel{\text{def}}{=} E/M \cdot E^* & \mathtt{m}[\overline{\tau}]/\mathtt{m}'[\overline{\sigma}] \stackrel{\text{def}}{=} \mathbb{0} \quad \text{if } \mathtt{m} \neq \mathtt{m}' & (E \cdot F)/M \stackrel{\text{def}}{=} E/M \cdot F + E \cdot F/M \end{array}$$

If we take the pattern E discussed earlier we have $E/\mathtt{A} = \mathbb{1} \cdot \mathtt{C} + \mathtt{A} \cdot \mathbb{0} + \mathbb{0} \cdot \mathbb{1} + \mathtt{B} \cdot \mathbb{1} \simeq \mathtt{B} + \mathtt{C}$. The pattern residual operator is closely related to Brzozowski's derivative in a commutative Kleene algebra [4, 28]. Unlike Brzozowski's derivative, the pattern residual is a partial operator: $E/\mathtt{m}[\overline{\sigma}]$ is defined provided that the $\overline{\sigma}$ are supertypes of all types $\overline{\tau}$ found in $\mathtt{m}$-tagged atoms within E. This condition has a natural justification: when choosing the message to remove from a mailbox containing a configuration of messages described by E, only the tag $\mathtt{m}$ of the message – and not the type of its arguments – matters. Thus, $\overline{\sigma}$ faithfully describe the received arguments provided that they are supertypes of *all* argument types of *all* $\mathtt{m}$-tagged message types in E. For example, assuming $\mathtt{nat} \leqslant \mathtt{int}$, we have that $(\mathtt{m}[\mathtt{int}] + \mathtt{m}[\mathtt{nat}])/\mathtt{m}[\mathtt{int}]$ is defined whereas $(\mathtt{m}[\mathtt{int}] + \mathtt{m}[\mathtt{nat}])/\mathtt{m}[\mathtt{nat}]$ is not.

We use the notion of pattern residual to define pattern normal forms:

▶ **Definition 15** (pattern normal form)**.** We say that a pattern E is in *normal form*, written $\vDash E$, if $E \vDash E$ is derivable by the following axioms and rules:

$$E \vDash \mathbb{0} \qquad E \vDash \mathbb{1} \qquad \frac{F \simeq E/M}{E \vDash M \cdot F} \qquad \frac{E \vDash F_1 \qquad E \vDash F_2}{E \vDash F_1 + F_2}$$

Essentially, the judgment $\vDash E$ verifies that E is expressed as a sum of $\mathbb{0}$, $\mathbb{1}$ and $M \cdot F$ terms where F is (equivalent to) the residual of E with respect to M.

A guarded process yields all the dependencies between the mailbox u being used and the names occurring free in the continuations, because the process will not be able to exercise the capabilities on these names until the message from u has been received.

Parallel composition. Rule [T-PAR] deals with parallel compositions of the form $P_1 \mid P_2$. This rule accounts for the fact that the same mailbox u may be used in both P_1 and P_2 according to different types. For example, P_1 might store an $\mathtt{A}$ message into u and P_2 might store a $\mathtt{B}$ message into u. In the type environment for the parallel composition as a whole we must be able to express with a single type the combined usages of u in P_1 and P_2. This is accomplished by introducing an operator that combines types:

▶ **Definition 16** (type combination)**.** We write $\tau \parallel \sigma$ for the *combination* of τ and σ, where $\parallel$ is the partial symmetric operator defined as follows:

$$\mathtt{!}E \parallel \mathtt{!}F \stackrel{\text{def}}{=} \mathtt{!}(E \cdot F) \qquad \mathtt{!}E \parallel \mathtt{?}(E \cdot F) \stackrel{\text{def}}{=} \mathtt{?}F \qquad \mathtt{?}(E \cdot F) \parallel \mathtt{!}E \stackrel{\text{def}}{=} \mathtt{?}F$$

Continuing the previous example, we have $\mathtt{!A} \parallel \mathtt{!B} = \mathtt{!}(\mathtt{A} \cdot \mathtt{B})$ because storing one $\mathtt{A}$ message and one $\mathtt{B}$ message in u means storing an overall configuration of messages described by the pattern $\mathtt{A} \cdot \mathtt{B}$. When u is used for both input and output operations, the combined type of u describes the overall balance of the mailbox. For example, we have $\mathtt{!A} \parallel \mathtt{?}(\mathtt{A} \cdot \mathtt{B}) = \mathtt{?B}$: if we combine a process that stores an $\mathtt{A}$ message into u with another process that consumes both an $\mathtt{A}$ message and a $\mathtt{B}$ message from the same mailbox in some unspecified order, then we end up with a process that consumes a $\mathtt{B}$ message from u.

Notice that $\|$ is a partial operator in that not all type combinations are defined. It might be tempting to relax $\|$ in such a way that $\mathtt{!}(\mathtt{A} \cdot \mathtt{B}) \parallel \mathtt{?A} = \mathtt{!B}$, so as to represent the fact that the combination of two processes results in an excess of messages that must be consumed by some other process. However, this would mean allowing different processes to consume messages from the same mailbox, which is not safe in general (see Example 17). For the same reason, the combination of $\mathtt{?}E$ and $\mathtt{?}F$ is always undefined regardless of E and F. Operators akin to $\|$ for the combination of channel types are commonly found in substructural type systems for the (linear) π-calculus [51, 44]. Unlike these systems, in our case the combination concerns also the content of a mailbox in addition to the capabilities for accessing it.

▶ **Example 17.** Suppose that we extend the type combination operator so that $\mathtt{?}(E \cdot F) = \mathtt{?}E \parallel \mathtt{?}F$. To see why this extension would be dangerous, consider the process

$$\begin{array}{l}(u\mathtt{!A}[\mathtt{True}] \mid u\mathtt{?A}(x).(\mathit{system}\mathtt{!print\_bool}[x] \mid \mathtt{free}\ u.\mathtt{done})) \mid \\ (u\mathtt{!A}[2] \mid u\mathtt{?A}(y).(\mathit{system}\mathtt{!print\_int}[y] \mid \mathtt{free}\ u.\mathtt{done}))\end{array}$$

Overall, this process stores into u a combination of messages that matches the pattern $\mathtt{A}[\mathtt{bool}] \cdot \mathtt{A}[\mathtt{int}]$ and retrieves from u the same combination of messages. Apparently, u is used in a balanced way. However, there is no guarantee that the $u\mathtt{!A}[\mathtt{True}]$ message is received by the process at the top and that the $u\mathtt{!A}[2]$ message is received by the process at the bottom. In fact, the converse may happen because only the tag of a message – not the type or value of its arguments – is used for matching messages in the mailbox calculus.

We now extend type combination to type environments in the expected way:

▶ **Definition 18** (type environment combination)**.** We write $\Gamma \parallel \Delta$ for the *combination* of Γ and Δ, where $\|$ is the partial operator inductively defined by the equations:

$$\Gamma \parallel \Delta \stackrel{\text{def}}{=} \Gamma, \Delta \quad \text{if } \mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Delta) = \emptyset \qquad (u : \tau, \Gamma) \parallel (u : \sigma, \Delta) \stackrel{\text{def}}{=} u : \tau \parallel \sigma, (\Gamma \parallel \Delta)$$

With this machinery in place, rule [T-PAR] is straightforward to understand and the dependency graph of $P_1 \mid P_2$ is simply the union of the dependency graphs of P_1 and P_2.

Mailbox restriction. Rule [T-NEW] establishes that the process creating a new mailbox a with scope P is well typed provided that the type of a is $\mathtt{?}\mathbb{1}$. This means that every message stored in the mailbox a by (a sub-process of) P is also consumed by (a sub-process of) P. The dependency graph of the process is the same as that of P, except that a is restricted.

Subsumption. As we have anticipated earlier in a few occasions, the subsumption rule [T-SUB] allows us to rewrite types in the type environment and to introduce associations for irrelevant names. The rule makes use of the following notion of subtyping for type environments:

▶ **Definition 19** (subtyping for type environments)**.** We say that Γ is a *subtype environment* of Δ if $\Gamma \leqslant \Delta$, where $\leqslant$ is the least preorder on type environments such that:

$$\frac{}{u : \mathtt{!}\mathbb{1}, \Gamma \leqslant \Gamma} \qquad \frac{\tau \leqslant \sigma}{u : \tau, \Gamma \leqslant u : \sigma, \Gamma}$$

Intuitively, $\Gamma \leqslant \Delta$ means that Γ provides more capabilities than Δ. For example, $u : \mathtt{!}(\mathtt{A}+\mathtt{B}), v : \mathtt{!}\mathbb{1} \leqslant u : \mathtt{!A}$ since a process that is well typed in the environment $u : \mathtt{!A}$ stores an $\mathtt{A}$ message into u, which is also a valid behavior in the environment $u : \mathtt{!}(\mathtt{A}+\mathtt{B}), v : \mathtt{!}\mathbb{1}$ where u has more capabilities (it is also possible to store a $\mathtt{B}$ message into u) and there is an irrelevant name v not used by the process.

Rule [t-sub] also allows us to replace the dependency graph yielded by P with another one that generates a superset of dependencies. In general, the dependency graph should be kept as small as possible to minimize the possibility of yielding mutual dependencies (see [t-par]). The replacement allowed by [t-sub] is handy for technical reasons, but not necessary. The point is that the residual of a process typically yields fewer dependencies than the process itself, so we use [t-sub] to enforce the invariance of dependency graphs across reductions.

▶ **Example 20.** We show the full typing derivation for FreeLock and BusyLock defined in Example 1. Our objective is to show the consistency of the global process declarations

$$\mathsf{FreeLock} : (\mathit{self} : \tau; \emptyset) \qquad \mathsf{BusyLock} : (\mathit{self} : \tau, \mathit{owner} : \rho; \{\mathit{self}, \mathit{owner}\})$$

where $\tau \overset{\text{def}}{=} \mathtt{?acquire}[\rho]^*$ and $\rho \overset{\text{def}}{=} \mathtt{!reply}[\mathtt{!release}]$. In the derivation trees below we rename *self* as x and *owner* and y to resonably fit the derivations within the page limits. We start from the body of BusyLock, which is simpler, and obtain

$$\dfrac{\dfrac{}{x : \mathtt{!release}, y : \rho \vdash y\mathtt{!reply}[x] :: \{y, x\}}\,\text{[t-msg]} \qquad \dfrac{\dfrac{\dfrac{}{x : \tau \vdash \mathsf{FreeLock}[x] :: \emptyset}\,\text{[t-def]}}{x : \sigma \vdash x\mathtt{?release}.\mathsf{FreeLock}[x]}\,\text{[t-in]}}{x : \sigma \vdash x\mathtt{?release}.\mathsf{FreeLock}[x] :: \emptyset}\,\text{[t-guard]}}{x : \tau, y : \rho \vdash y\mathtt{!reply}[x] \mid x\mathtt{?release}.\mathsf{Lock}[x] :: \{y, x\} \sqcup \emptyset}\,\text{[t-par]}$$

where $\sigma \overset{\text{def}}{=} \mathtt{?}(\mathtt{release} \cdot \mathtt{acquire}[\rho]^*)$.

Concerning FreeLock, the key step is rewriting the pattern of τ in a normal form that matches the branching structure of the process. To this aim, we use the property $E^* \simeq \mathbb{1} + E \cdot E^*$ and the fact that $\mathbb{0}$ is absorbing for the product connective:

$$\dfrac{\dfrac{\vdots \qquad \dfrac{\dfrac{\dfrac{}{x : \mathtt{?}\mathbb{0} \vdash \mathtt{fail}\ x}\,\text{[t-fail]}}{x : \mathtt{?}\mathbb{0} \vdash \mathtt{fail}\ x :: \emptyset}\,\text{[t-guard]}}{x : \mathtt{?}(\mathtt{release} \cdot \mathbb{0}) \vdash x\mathtt{?release}.\mathtt{fail}\ x}\,\text{[t-in]}}{x : \mathtt{?}(\mathbb{1} + \mathtt{acquire}[\rho] \cdot \mathtt{acquire}[\rho]^* + \mathtt{release} \cdot \mathbb{0}) \vdash \cdots + x\mathtt{?release}.\mathtt{fail}\ x}\,\text{[t-branch]}}{\dfrac{x : \mathtt{?}(\mathbb{1} + \mathtt{acquire}[\rho] \cdot \mathtt{acquire}[\rho]^* + \mathtt{release} \cdot \mathbb{0}) \vdash \cdots + x\mathtt{?release}.\mathtt{fail}\ x :: \emptyset}{x : \tau \vdash \mathtt{free}\ x.\mathtt{done} + \cdots + x\mathtt{?release}.\mathtt{fail}\ x :: \emptyset}\,\text{[t-sub]}}\,\text{[t-guard]}$$

The elided sub-derivation concerns the first two branches of FreeLock and is as follows:

$$\dfrac{\dfrac{\dfrac{}{\emptyset \vdash \mathtt{done} :: \emptyset}\,\text{[t-done]}}{x : \mathtt{?}\mathbb{1} \vdash \mathtt{free}\ x.\mathtt{done}}\,\text{[t-free]} \qquad \dfrac{\dfrac{}{x : \tau, y : \rho \vdash \mathsf{BusyLock}[x, y] :: \{x, y\}}\,\text{[t-def]}}{x : \mathtt{?acquire}[\rho] \cdot \mathtt{acquire}[\rho]^* \vdash x\mathtt{?acquire}(y).\mathsf{BusyLock}[x, y]}}{x : \mathtt{?}(\mathbb{1} + \mathtt{acquire}[\rho] \cdot \mathtt{acquire}[\rho]^*) \vdash \mathtt{free}\ x.\mathtt{done} + x\mathtt{?acquire}(y).\mathsf{BusyLock}[x, y]}$$

The process (1), combining an instance of the lock and the users *alice* and *carol*, is also well typed. As we will see at the end of Section 3.4, this implies that both *alice* and *carol* are able to acquire the lock, albeit in some unspecified order.

▶ **Example 21.** In this example we show that the process (2) of Example 6 is ill typed. In order to do so, we assume the global process declaration

$$\mathsf{Future} : (\mathit{self} : \texttt{?}(\texttt{put}[\texttt{int}] \cdot \texttt{get}[\texttt{!reply}[\texttt{int}]]^*); \emptyset)$$

which can be shown to be consistent with the given definition for Future. In the derivation below we use the pattern $F \overset{\text{def}}{=} \texttt{put}[\texttt{int}] \cdot \texttt{get}[\rho]^*$ and the types $\tau \overset{\text{def}}{=} \texttt{!put}[\texttt{int}]$, $\sigma \overset{\text{def}}{=} \texttt{?}(\texttt{reply}[\texttt{int}] \cdot \mathbb{1})$ and $\rho \overset{\text{def}}{=} \texttt{!reply}[\texttt{int}]$:

$$\dfrac{\dfrac{\dfrac{}{f : \texttt{!get}[\rho], c : \rho \vdash f\texttt{!get}[c] :: \{f, c\}} \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{f : \tau, x : \texttt{int} \vdash f\texttt{!put}[x] :: \emptyset}}{f : \tau, c : \texttt{?}\mathbb{1}, x : \texttt{int} \vdash \texttt{free}\ c.f\texttt{!put}[x]}}{f : \tau, c : \texttt{?}\mathbb{1}, x : \texttt{int} \vdash \texttt{free}\ c.f\texttt{!put}[x] :: \{c, f\}}}{f : \tau, c : \sigma \vdash c\texttt{?reply}(x).\texttt{free}\ c.f\texttt{!put}[x]}}{f : \tau, c : \sigma \vdash c\texttt{?reply}(x).\texttt{free}\ c.f\texttt{!put}[x] :: \{c, f\}}}{\dfrac{f : \texttt{!}(\texttt{get}[\rho] \cdot \texttt{put}[\texttt{int}]), c : \texttt{?}\mathbb{1} \vdash f\texttt{!get}[c] \mid c\texttt{?reply}(x).\texttt{free}\ c.f\texttt{!put}[x] :: -}{f : \texttt{!}F, c : \texttt{?}\mathbb{1} \vdash f\texttt{!get}[c] \mid c\texttt{?reply}(x).\texttt{free}\ c.f\texttt{!put}[x] :: -}\ \text{[T-SUB]}}}{f : \texttt{!}F \vdash (\nu c)(f\texttt{!get}[c] \mid c\texttt{?reply}(x).\texttt{free}\ c.f\texttt{!put}[x]) :: -}$$

In attempting this derivation we have implicitly extended the typing rules so that names with type $\texttt{int}$ do not contribute in generating any significant dependency. The critical point of the derivation is the application of [T-PAR], where we are composing two parallel processes that yield a circular dependency between c and f. In the process on the left hand side, the dependency $\{f, c\}$ arises because c is sent as a reference in a message targeted to f. In the process on the right hand side, the dependency $\{c, f\}$ arises because there are guards concerning the mailbox c that block an output operation on the mailbox f.

▶ **Example 22** (non-deterministic choice)**.** The same input action can occur multiple times in the same guarded process. This feature can be used to encode in the mailbox calculus the non-deterministic choice between P_1 and P_2 as the process

$$(\nu a)(\texttt{free}\ a.P_1 + \texttt{free}\ a.P_2) \tag{4}$$

provided that $\Gamma \vdash P_i :: \varphi_i$ for $i = 1, 2$. That is, P_1 and P_2 must be well typed in the same type environment. Below is the typing derivation for (4)

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash P_1 :: \varphi_1}{\Gamma, a : \texttt{?}\mathbb{1} \vdash \texttt{free}\ a.P_1}\ \text{[T-FREE]} \quad \dfrac{\Gamma \vdash P_2 :: \varphi_2}{\Gamma, a : \texttt{?}\mathbb{1} \vdash \texttt{free}\ a.P_2}\ \text{[T-FREE]}}{\Gamma, a : \texttt{?}(\mathbb{1} + \mathbb{1}) \vdash \texttt{free}\ a.P_1 + \texttt{free}\ a.P_2}\ \text{[T-BRANCH]}}{\Gamma, a : \texttt{?}(\mathbb{1} + \mathbb{1}) \vdash \texttt{free}\ a.P_1 + \texttt{free}\ a.P_2 :: \varphi}\ \text{[T-GUARD]}}{\Gamma, a : \texttt{?}\mathbb{1} \vdash \texttt{free}\ a.P_1 + \texttt{free}\ a.P_2 :: \varphi}\ \text{[T-SUB]}}{\Gamma \vdash (\nu a)(\texttt{free}\ a.P_1 + \texttt{free}\ a.P_2) :: (\nu a)\varphi}\ \text{[T-NEW]}$$

where $\varphi \overset{\text{def}}{=} \{a, \mathsf{dom}(\Gamma)\}$. The key step is the application of [T-SUB], which exploits the idempotency of $+$ (in patterns) to rewrite $\mathbb{1}$ as the equivalent pattern $\mathbb{1} + \mathbb{1}$.

3.4 Properties of well-typed processes

In this section we state the main properties enjoyed by well-typed processes. As usual, subject reduction is instrumental for all of the results that follow as it guarantees that typing is preserved by reductions:

▶ **Theorem 23.** *If Γ is reliable and $\Gamma \vdash P :: \varphi$ and $P \rightarrow Q$, then $\Gamma \vdash Q :: \varphi$.*

Interestingly, Theorem 23 seems to imply that the types of the mailboxes used by a process do not change. In sharp contrast, other popular behavioral typing disciplines (session types in particular), are characterized by a subject reduction result in which types reduce along with processes. Theorem 23 also seems to contradict the observations made earlier concerning the fact that the mailboxes used by a process may have different types (Example 11). The type preservation guarantee assured by Theorem 23 can be explained by recalling that the type environment Γ in a judgment $\Gamma \vdash P :: \varphi$ already takes into account the overall balance between the messages stored into and consumed from the mailbox used by P (see Definition 18). In light of this observation, Theorem 23 simply asserts that well-typed processes are steady state: they never produce more messages than those that are consumed, nor do they ever try to consume more messages than those that are produced.

A practically relevant consequence of Theorem 23 is that, by looking at the type $?E$ of the mailbox a used by a guarded process P (rule [T-GUARD]), it is possible to determine *bounds* to the number of messages that can be found in the mailbox as P waits for a message to receive. In particular, if every configuration of E contains at most k `m`-tagged atoms, then at runtime a contains at most `m`-tagged messages. As a special case, a mailbox of type $?\mathbb{1}$ is guaranteed to be empty and can be statically deallocated. Note that the bounds may change after P receives a message. For example, a free lock is guaranteed to have no `release` messages in its mailbox, and will have at most one when it is busy (see Example 20).

The main result concerns the soundness of the type system, guaranteeing that well-typed (closed) processes are both mailbox conformant and deadlock free (Definitions 4 and 5):

▶ **Theorem 24.** *If $\emptyset \vdash P :: \varphi$, then P is mailbox conformant and deadlock free.*

Fair termination and junk freedom are not enforced by our typing discipline in general. The usual counterexamples include processes that postpone indefinitely the use of a mailbox with a relevant type. For instance, the `m` message in the well-typed process $(\nu a)(a!\mathtt{m} \mid \mathsf{X}[a])$ where $\mathsf{X}(x) \triangleq \mathsf{X}[x]$ is never consumed because a is never used for an input operation. Nevertheless, fair termination is guaranteed for the class of finitely unfolding processes:

▶ **Theorem 25.** *We say that P is* finitely unfolding *if all maximal reductions of P use* [R-DEF] *finitely many times. If $\emptyset \vdash P :: \varphi$ and P is finitely unfolding, then P is fairly terminating.*

The class of finitely unfolding processes obviously includes all finite processes (those not using process invocations) but also many recursive processes. For example, every process of the form $(\nu a)(a!\mathtt{m} \mid \cdots \mid a!\mathtt{m} \mid \mathsf{X}[a])$ where $\mathsf{X}(x) \triangleq x?\mathtt{m}.\mathsf{X}[x] + \mathtt{free}\ x.\mathtt{done}$ is closed, well typed and finitely unfolding regardless of the number of `m` messages stored in a, hence is fairly terminating and junk free by Theorem 25.

4 Examples

In this section we discuss a few more examples that illustrate the expressiveness of the mailbox calculus and of its type system. We consider a variant of the bank account shown in Listing 1 (Section 4.1), the case of master-workers parallelism (Section 4.2) and the encoding of binary sessions extended with forks and joins (Sections 4.3 and 4.4).

4.1 Actors using futures

Many Scala programs combine actors with futures [53]. As an example, Listing 2 shows an alternative version of the `Account` actor in Akka that differes from Listing 1 in the

```
 1 class Account(var balance: Double) extends AkkaActor[AnyRef] {
 2   override def process(msg: AnyRef) {
 3     msg match {
 4       case dm: DebitMessage =>
 5         balance += dm.amount
 6         sender() ! ReplyMessage.ONLY
 7       case cm: CreditMessage =>
 8         balance -= cm.amount
 9         val recipient = cm.recipient.asInstanceOf[ActorRef]
10         val future = ask(recipient, new DebitMessage(self,cm.amount))
11         Await.result(future, Duration.Inf)
12         sender() ! ReplyMessage.ONLY
13       case _: StopMessage => exit()
14       case message =>
15         val ex = new IllegalArgumentException("Unsupported␣message")
16         ex.printStackTrace(System.err)
17     }
18   }
19 }
```

Listing 2 An Akka actor using futures from the `Savina` benchmark suite [32].

handling of `CreditMessages` (lines 10–11). The `future` variable created here is initialized asynchronously with the result of the debit operation invoked on `recipient`. To make sure that each transaction is atomic, the actor waits for the variable to be resolved (line 11) before notifying `sender` that the operation has been completed.

This version of `Account` is arguably simpler than the one in Listing 1, if only because the actor has a unique top-level behavior. One way of modeling this implementation of `Account` in the mailbox calculus is to use Future, discussed in Example 2. A simpler modeling stems from the observation that `future` in Listing 2 is used for a one-shot synchronization. A future variable with this property is akin to a mailbox from which the value of the resolved variable is retrieved exactly once. Following this approach we obtain the process below:

$$
\begin{array}{rcl}
\mathsf{Account}(\mathit{self}, \mathit{balance}) & \triangleq & \mathit{self}?\mathtt{debit}(\mathit{amount}, \mathit{sender})\,.\\
& & \mathit{sender}\,!\mathtt{reply} \mid \mathsf{Account}[\mathit{self}, \mathit{balance} + \mathit{amount}]\\
& + & \mathit{self}?\mathtt{credit}(\mathit{amount}, \mathit{recipient}, \mathit{sender})\,.\\
& & (\nu \mathit{future}) \left(\begin{array}{l} \mathit{recipient}\,!\mathtt{debit}[\mathit{amount}, \mathit{future}] \mid \\ \mathit{future}?\mathtt{reply}\,.\,\mathtt{free}\ \mathit{future}\,.\\ (\mathit{sender}\,!\mathtt{reply} \mid \mathsf{Account}[\mathit{self}, \mathit{balance} - \mathit{amount}]) \end{array}\right)\\
& + & \mathit{self}?\mathtt{stop}\,.\,\mathtt{free}\ \mathit{self}\,.\,\mathtt{done}\\
& + & \mathit{self}?\mathtt{reply}\,.\,\mathtt{fail}\ \mathit{self}
\end{array}
$$

Compared to the process in Example 3, here the notification from the *recipient* account is received from the mailbox *future*, which is created locally during the handling of the `credit` message. The rest of the process is the same as before. This definition of Account and the one in Example 3 can both be shown to be consistent with the declaration

$$\mathsf{Account} : (\mathit{self} : ?(\mathtt{debit}[\mathtt{int}, \rho]^* \cdot \mathtt{credit}[\mathtt{int}, !\mathtt{debit}[\mathtt{int}, \rho], \rho]^* + \mathtt{stop}), \mathit{balance} : \mathtt{int}; \emptyset)$$

where $\rho \overset{\text{def}}{=} !\mathtt{reply}$. In particular, the dependencies between *self* and *future* that originate in this version of Account are not observable from outside Account itself.

The use of multiple mailboxes and the interleaving of blocking operations on them may increase the likelyhood of programming mistakes causing mismatched communications and/or deadlocks. However, these errors can be detected by a suitable typing discipline such the one proposed in this paper. Types can also be used to mitigate the runtime overhead resulting from the use of multiple mailboxes. Here, for example, the typing of *future* guarantees that this mailbox is used for receiving a *single* message and that *future* is empty by the time `free` *future* is performed. A clever compiler can take advantage of this information to statically optimize both the allocation and the deallocation of this mailbox.

4.2 Master-workers parallelism

In this example we model a *master* process that receives tasks to perform from a *client*. For each task, the master creates a pool of *workers* and assigns each worker a share of work. The master waits for all partial results from the workers before sending the final result back to the client and making itself available again. The number of workers may depend on some quantity possibly related to the task to be performed and that is known at runtime only.

Below we define three processes corresponding to the three states in which the master process can be, and we leave Worker unspecified:

$$\begin{array}{rcl}
\mathsf{Available}(\mathit{self}) & \triangleq & \mathit{self}?\mathtt{task}(\mathit{client}).(\nu \mathit{pool})\mathsf{CreatePool}[\mathit{self},\mathit{pool},\mathit{client}] \\
& & +\ \mathtt{free}\ \mathit{self}.\mathtt{done} \\
\mathsf{CreatePool}(\mathit{self},\mathit{pool},\mathit{client}) & \triangleq & \mathtt{if}\ \mathit{more\ workers\ needed}\ \mathtt{then} \\
& & \quad (\nu \mathit{worker})(\mathit{worker}!\mathtt{work}[\mathit{pool}] \mid \mathsf{Worker}[\mathit{worker}]) \mid \\
& & \quad \mathsf{CreatePool}[\mathit{self},\mathit{pool},\mathit{client}] \\
& & \mathtt{else} \\
& & \quad \mathsf{CollectResults}[\mathit{self},\mathit{pool},\mathit{client}] \\
\mathsf{CollectResults}(\mathit{self},\mathit{pool},\mathit{client}) & \triangleq & \mathit{pool}?\mathtt{result}.\mathsf{CollectResults}[\mathit{self},\mathit{pool},\mathit{client}] \\
& & +\ \mathtt{free}\ \mathit{pool}.(\mathit{client}!\mathtt{result} \mid \mathsf{Available}[\mathit{self}])
\end{array}$$

The "`if` *condition* `then` P `else` Q" form used here can be encoded in the mailbox calculus and is typed similarly to the non-deterministic choice of Example 22. These definitions can be shown to be consistent with the following declarations:

$$\begin{array}{rl}
\mathsf{Available} : & (\mathit{self} : ?\mathtt{task}[!\mathtt{result}]^*; \emptyset) \\
\mathsf{CreatePool}, \mathsf{CollectResults} : & (\mathit{self} : ?\mathtt{task}[!\mathtt{result}]^*, \mathit{pool} : ?\mathtt{result}^*, \mathit{client} : !\mathtt{result}; \\
& \{\mathit{pool},\mathit{self}\} \sqcup \{\mathit{pool},\mathit{client}\})
\end{array}$$

The usual implementation of this coordination pattern requires the programmer to keep track of the number of active workers using a counter that is decremented each time a partial result is collected [32]. When the counter reaches zero, the master knows that all the workers have finished their job and notifies the client. In the mailbox calculus, we can model the counter using a dedicated mailbox *pool* from which the partial results are collected: when *pool* becomes disposable, it means that no more active workers remain.

4.3 Encoding of binary sessions

Session types [27, 29] have become a popular formalism for the specification and enforcement of structured protocols through static analysis. A session is a private communication channel shared by processes that interact through one of its *endpoints*. Each endpoint is associated with a *session type* that specifies the type, direction and order of messages that are supposed to be exchanged through that endpoint. A typical syntax for session types in the case of *binary sessions* (those connecting exactly two peer processes) is shown below:

$$T, S \quad ::= \quad \texttt{end} \quad | \quad ?[\tau].T \quad | \quad ![\tau].T \quad | \quad T \,\&\, S \quad | \quad T \oplus S$$

A session type $?[\tau].T$ describes an endpoint used for receiving a message of type τ and then according to T. Dually, a session type $![\tau].T$ describes an endpoint used for sending a message of type τ and then according to T. An external choice $T \,\&\, S$ describes an endpoint used for receiving a selection (either `left` or `right`) and then according to the corresponding continuation (either T or S). Dually, an internal choice $T \oplus S$ describes an endpoint used for making a selection and then according to the corresponding continuation. Communication safety and progress of a binary session are guaranteed by the fact that its two endpoints are linear resources typed by *dual* session types, where the dual of T is obtained by swapping inputs with outputs and internal with external choices.

In this example we encode sessions and session types using mailboxes and mailbox types. We encode a session as a non-uniform, concurrent object. The object is "concurrent" because it is accessed concurrently by the two peers of the session. It is "non-uniform" because its interface changes over time, as the session progresses. The object uses a mailbox *self* and its behavior is defined by the equations for $\mathsf{Session}_T(\mathit{self})$ shown below, where T is the session type according to which it must be used by one of the peers:

$$\begin{array}{rl}
\mathsf{Session}_{\texttt{end}}(\mathit{self}) \triangleq & \texttt{free}\ \mathit{self}\,.\,\texttt{done} \\
\mathsf{Session}_{?[\tau].T}(\mathit{self}) \triangleq & \mathit{self}?\texttt{send}(x, s)\,.\,\mathit{self}?\texttt{receive}(r)\,. \\
& (s!\texttt{reply}[\mathit{self}] \mid r!\texttt{reply}[x, \mathit{self}] \mid \mathsf{Session}_T[\mathit{self}]) \\
\mathsf{Session}_{![\tau].T}(\mathit{self}) \triangleq & \mathsf{Session}_{?[\tau].T}[\mathit{self}] \\
\mathsf{Session}_{T\&S}(\mathit{self}) \triangleq & \mathit{self}?\texttt{left}(s)\,.\,\mathit{self}?\texttt{receive}(r)\,. \\
& (s!\texttt{reply}[\mathit{self}] \mid r!\texttt{left}[\mathit{self}] \mid \mathsf{Session}_T[\mathit{self}]) \\
+ & \mathit{self}?\texttt{right}(s)\,.\,\mathit{self}?\texttt{receive}(r)\,. \\
& (s!\texttt{reply}[\mathit{self}] \mid r!\texttt{right}[\mathit{self}] \mid \mathsf{Session}_S[\mathit{self}]) \\
\mathsf{Session}_{T\oplus S}(\mathit{self}) \triangleq & \mathsf{Session}_{T\&S}[\mathit{self}]
\end{array}$$

To grasp the intuition behind the definition of $\mathsf{Session}_T(\mathit{self})$, it helps to recall that each stage of a session corresponds to an interaction between the two peers, where one process plays the role of "sender" and its peer that of "receiver". Both peers manifest their willingness to interact by storing a message into the session's mailbox. The receiver always stores a `receive` message, while the sender stores either `send`, `left` or `right` according to T. All messages contain a reference to the mailbox owned by sender and receiver (respectively s and r) where they will be notified once the interaction is completed. A `send` message also carries actual payload x being exchanged. The role of $\mathsf{Session}_T(\mathit{self})$ is simply to forward each message from the sender to the receiver. The notifications stored in s and r contain a reference to the session's mailbox so that its type reflects the session's updated interface corresponding to the rest of the conversation.

Interestingly, the encoding of a session with type T is undistinguishable from that of a session with the dual type $\overline{T}$. This is natural by recalling that each stage of a session corresponds to a single interaction between the two peers: the order in which they store the respective messages in the session's mailbox is in general unpredictable but also unimportant, for both messages are necessary to complete each interaction.

As an example, suppose we want to model a system where Alice asks Carol to compute the sum of two numbers exchanged through a session s. Alice and Carol use the session according to the session types $T \overset{\text{def}}{=} ![\texttt{int}].![\texttt{int}].?[\texttt{int}].\texttt{end}$ and $\overline{T} \overset{\text{def}}{=} ?[\texttt{int}].?[\texttt{int}].![\texttt{int}].\texttt{end}$, respectively. The system is modeled as the process

$$(\nu\, alice)(\nu\, carol)(\nu s)(\mathsf{Alice}[alice, s] \mid \mathsf{Carol}[carol, s] \mid \mathsf{Session}_T[s]) \tag{5}$$

where Alice and Carol are defined as follows:

$$\begin{aligned}
\mathsf{Alice}(self, s) &\triangleq s\,!\,\mathtt{send}[4, self] \mid self?\mathtt{reply}(s)\,.\\
&\quad(s\,!\,\mathtt{send}[2, self] \mid self?\mathtt{reply}(s)\,.\\
&\quad(s\,!\,\mathtt{receive}[self] \mid self?\mathtt{reply}(x, s)\,.\\
&\quad(system\,!\,\mathtt{print\_int}[x] \mid \mathtt{free}\ self\,.\mathtt{done})))\\
\mathsf{Carol}(self, s) &\triangleq s\,!\,\mathtt{receive}[self] \mid self?\mathtt{reply}(x, s)\,.\\
&\quad(s\,!\,\mathtt{receive}[self] \mid self?\mathtt{reply}(y, s)\,.\\
&\quad(s\,!\,\mathtt{send}[x + y, self] \mid self?\mathtt{reply}(s)\,.\mathtt{free}\ self\,.\mathtt{done}))
\end{aligned}$$

The process (5) and the definitions of Alice and Carol are well typed. In general, $\mathsf{Session}_T$ is consistent with the declaration $\mathsf{Session}_T : (self : ?(\mathscr{E}(T) \cdot \mathscr{E}(\overline{T})); \emptyset)$ where $\mathscr{E}(T)$ is the pattern defined by the following equations:

$$\begin{aligned}
\mathscr{E}(\mathtt{end}) &\stackrel{\text{def}}{=} \mathbb{1}\\
\mathscr{E}(?[\tau].T) &\stackrel{\text{def}}{=} \mathtt{receive}[!\mathtt{reply}[\tau, !\mathscr{E}(T)]]\\
\mathscr{E}(![\tau].T) &\stackrel{\text{def}}{=} \mathtt{send}[\tau, !\mathtt{reply}[!\mathscr{E}(T)]]\\
\mathscr{E}(T \,\&\, S) &\stackrel{\text{def}}{=} \mathtt{receive}[!(\mathtt{left}[!\mathscr{E}(T)] + \mathtt{right}[!\mathscr{E}(S)])]\\
\mathscr{E}(T \oplus S) &\stackrel{\text{def}}{=} \mathtt{left}[!\mathtt{reply}[!\mathscr{E}(T)]] + \mathtt{right}[!\mathtt{reply}[!\mathscr{E}(S)]]
\end{aligned}$$

By interpreting both the syntax of T and the definition of $\mathsf{Session}_T$ coinductively, it is easy to see that this encoding of binary sessions extends to internal and external choices with arbitrary labels and also to recursive session types. The usual regularity condition ensures that $\mathsf{Session}_T$ is finitely representable. Finally, note that the notion of subtyping for encoded session types induced by Definition 9 coincides with the conventional one [21]. Thus, the mailbox type system subsumes a rich session type system where Theorem 24 corresponds to the well-known communication safety and progress properties of sessions.

4.4 Encoding of sessions with forks and joins

We have seen that it is possible to share the output capability on a mailbox among several processes. We can take advantage of this feature to extend session types with forks and joins:

$$T, S \ ::= \ \mathtt{end} \ \mid \ ?[\tau].T \ \mid \ ![\tau].T \ \mid \ T \,\&\, S \ \mid \ T \oplus S \ \mid \ \invamp_{i\in I}\mathtt{m}_i[\tau_i]; T \ \mid \ \otimes_{i\in I}\mathtt{m}_i[\tau_i]; T$$

The idea is that the session type $\otimes_{1\leq i\leq n}\mathtt{m}_i[\tau_i]; T$ describes an endpoint that can be used for sending *all* of the $\mathtt{m}_i$ messages, and then according to T. The difference between $\otimes_{1\leq i\leq n}\mathtt{m}_i[\tau_i]; T$ and a session type of the form $![\tau_1] \dots ![\tau_n].T$ is that the $\mathtt{m}_i$ messages can be sent by independent processes (for example, by parallel workers) in whatever order instead of by a single sender. Dually, the session type $\invamp_{1\leq i\leq n}\mathtt{m}_i[\tau_i]; T$ describes an endpoint that can be used for collecting *all* of the $\mathtt{m}_i$ messages, and then according to T. Forks and joins are dual to each other, just like simple outputs are dual to simple inputs. The tags $\mathtt{m}_i$ need not be distinct, but equal tags must correspond to equal argument types.

The extension of $\mathsf{Session}_T$ to forks and joins is shown below:

$$\begin{aligned}
\mathsf{Session}_{\otimes_{i\in I}\mathtt{m}_i[\tau_i];T}(self) &\triangleq self?\mathtt{send}(s)\,.\,self?\mathtt{receive}(r)\,.\,\mathsf{Join}_{\otimes_{i\in I}\mathtt{m}_i[\tau_i];T}[self, s, r]\\
\mathsf{Session}_{\invamp_{i\in I}\mathtt{m}_i[\tau_i];T}(self) &\triangleq \mathsf{Session}_{\otimes_{i\in I}\mathtt{m}_i[\tau_i];T}[self]\\
\mathsf{Join}_{\otimes_{i\in I}\mathtt{m}_i[\tau_i];T}(self, s, r) &\triangleq \begin{cases} s\,!\,\mathtt{reply}[self] \mid r\,!\,\mathtt{reply}[self] \mid \mathsf{Session}_T[self] & \text{if } I = \emptyset\\ self?\mathtt{m}_i(x_i)\,.\,(r\,!\,\mathtt{m}_i[x_i] \mid \mathsf{Join}_{\otimes_{i\in I\setminus\{i\}}\mathtt{m}_i[\tau_i];T}[self, s, r]) & \text{if } i \in I \end{cases}
\end{aligned}$$

As in the case of simple interactions, sender and receiver manifest their willingness to interact by storing `send` and `receive` messages into the session's mailbox *self*. At that point, $\mathsf{Join}_T[\mathit{self}, s, r]$ forwards all the $\mathtt{m}_i$ messages coming from the sender side to the receiver side, in some arbitrary order (case $i \in I$). When there are no more messages to forward (case $I = \emptyset$) both sender and receiver are notified with a `reply` message that carries a reference to the session's endpoint, with its type updated according to the rest of the continuation.

The encoding of session types extended to forks and joins follows easily:

$$\mathscr{E}(\otimes_{i\in I}\mathtt{m}_i[\tau_i]; T) \stackrel{\text{def}}{=} \mathtt{send}[\mathtt{!reply}[\mathtt{!}\mathscr{E}(T)]] \cdot \textstyle\prod_{i\in I} \mathtt{m}_i[\tau_i]$$
$$\mathscr{E}(\invamp_{i\in I}\mathtt{m}_i[\tau_i]; T) \stackrel{\text{def}}{=} \mathtt{receive}[\mathtt{!}(\textstyle\prod_{i\in I} \mathtt{m}_i[\tau_i]) \cdot \mathtt{reply}[\mathtt{!}\mathscr{E}(T)]]$$

An alternative definition of Join_T that fowards messages as soon as they become available can be obtained by providing suitable input actions for each $i \in I$ instead of picking an arbitrary $i \in I$.

5 Related Work

Concurrent Objects. There are analogies between actors and concurrent objects. Both entities are equipped with a unique identifier through which they receive messages, they may interact with several concurrent clients and their behavior may vary over time, as the entity interacts with its clients. Therefore, static analysis techniques developed for concurrent objects may be applicable to actors (and vice versa). Relevant works exploring behavioral type systems for concurrent objects include those of Najim *et al.* [42], Ravara and Vasconcelos [50], and Puntigam *et al.* [48, 49]. As in the pure actor model, each object has a unique mailbox and the input capability on that mailbox cannot be transferred. The mailbox calculus does not have these restrictions. A notable variation is the model studied by Ravara and Vasconcelos [50], which accounts for *distributed objects*: there can be several copies of an object that react to messages targeted to the same mailbox. Another common trait of these works is that the type discipline focuses on sequences of method invocations and types contain (abstract) information on the internal state of objects and on state transitions. Indeed, types are either finite-state automata [42], or terms of a process algebra [50] or tokens annotated with state transitions [49]. In contrast, mailbox types focus on the content of a mailbox and sequencing is expressed in the type of explicit continuations. The properties enforced by the type systems in these works differ significantly. Some do not consider deadlock freedom [50, 48], others do not account for out-of-order message processing [48]. Details on the enforced properties also vary. For example, the notion of protocol conformance used by Ravara and Vasconcelos [50] allows sending to an object any message that can be handled *by some future state* of the object. In our setting, this would mean allowing to send a `release` message to a free lock if the lock is acquired later on, or allowing to send a `reply` message to an account if the account will later be involved in a transaction.

The most closely related work among those addressing concurrent objects is the one by Crafa and Padovani [15], who propose the use of the Objective Join Calculus as a model for non-uniform, concurrent objects and develop a type discipline that can be used for enforcing concurrent object protocols. While mailbox types have been directly inspired by their types of concurrent objects, there are two major differences with our work. First, in the Objective Join Calculus every object is associated with a single mailbox, just like in the pure actor model [26, 1], meaning that mailboxes are not first class. As a consequence, the types considered by Crafa and Padovani [15] all have an (implicit) output capability. Second, in the Objective Join Calculus input operations are defined atomically on molecules of messages, whereas in the mailbox calculus messages are received one at a time. As a consequence, the type of a mailbox in the work of Crafa and Padovani [15] is invariant, whereas the same mailbox may have different types at different times in the mailbox calculus (Example 11).

Static analysis of actors. Srinivasan and Mycroft [52] define a type discipline for controlling the ownership of messages and ensuring actor isolation, but consider only uniformly typed mailboxes and do not address mailbox conformance or deadlock freedom.

Christakis and Sagonas [10] describe a static analysis technique whose aim is to ensure matching between send and receive operations in actors. The technique, which is described only informally and does not account for deadlocks, has been implemented in a tool called dialyzer and used for the analysis of Erlang programs.

Crafa [14] defines a behavioural type system for actors aimed at ensuring that the order of messages produced and consumed by an actor follows a prescribed protocol. Protocols are expressed as types and describe the behavior of actors rather than the content of the mailboxes they use. Deadlock freedom is not addressed.

Charousset *et al.* [8] describe the design and implementation of CAF, the C++ Actor Framework. Among the features of CAF is the use of *type-safe message passing interfaces* that makes it possible to statically detect a number of protocol violations by piggybacking on the C++ type system. There are close analogies between CAF's message passing interfaces and mailbox types with output capability: both are equipped with a subset semantics and report only those messages that can be stored into the mailbox through a mailbox reference with that type. Charousset *et al.* [8] point out that this feature fosters the decoupling of actors and enables incremental program recompilation.

Giachino *et al.* [22, 40] define a type system for the deadlock analysis of actors making use of implicit futures. Mailbox conformance and deadlocks due to communications are not taken into account.

He *et al.* [25] discuss a typed extension of Akka [24] ensuring that, in well-typed programs, messages sent to an actor are understood by the actor. The type system is not behavioral though, meaning that it is not possible to reason on which *configurations* of messages are legal. In particular, behavior upgrades are monotonic and actors can only increase the type of messages they understand. This is in sharp contrast with our typing disipline, which allows behavior upgrades with possibly unrelated mailbox types (Examples 1 and 2).

Fowler *et al.* [20] formalize channel-based and mailbox-based communicating systems, highlighting the differences between the two models and studying type-preserving encodings between them. Mailboxes in their work are uniformly typed, but the availability of union types make it possible to host heterogeneous values within the same mailbox. This however may lead to a loss of precision in typing. This phenomenon, called *type pollution* by He *et al.* [25] and Fowler *et al.* [20], is observable to some extent also in our typing discipline and can be mitigated by the use of multiple mailboxes (*cf.* Section 4.2). Finally, Fowler *et al.* [20] leave the extension of their investigation to behaviorally-typed language of actors as future work. Our typing discipline is a potential candidate for this investigation and addresses a more general setting thanks to the support for first-class mailboxes.

Sessions and actors. The encoding of binary sessions into actors discussed in Section 4.3 is new and has been inspired by the encoding of binary sessions into the linear π-calculus [35, 16], whereby each message is paired with a continuation. In our case, the continuation, instead of being a fresh (linear) channel, is either the mailbox of the peer or that of the session. This style of communication with explicit continuation passing is idiomatic in the actor model, which is based on asynchronous communications. The encoding discussed in Section 4.3 can be generalized to multiparty sessions by defining $\mathsf{Session}_T$ as a *medium process* through which messages are exchanged between the parties of the session. This idea has been put forward by Caires and Pérez [5] to encode multiparty sessions using binary sessions.

Mostrous and Vasconcelos [41] study a session type system for enforcing ordered dyadic interactions in core Erlang. They use *references* for distinguishing messages pertaining to different sessions, making use of the advanced pattern matching capabilities of Erlang. Their type system guarantees a weaker form of mailbox conformance, whereby junk messages may be present at the end of a computation, and does not consider deadlock freedom. Compared to our encoding of binary sessions, their approach does not require a medium process representing the session itself.

Neykova and Yoshida [43] propose a framework based on multiparty session types for the specification and implementation of actor systems with guarantees on the order of interactions. This approach is applicable when designing an entire system and both the network topology and the communication protocol can be established in advance. Fowler [19] builds upon the work of Neykova and Yoshida to obtain a runtime protocol monitoring mechanism for Erlang. Charalambides *et al.* [7] extend the multiparty session approach with a protocol specification language that is parametric in the number of actors participating in the system. In contrast to these approaches based on multiparty/global session types, our approach ensures mailbox conformance and deadlock freedom of a system compositionally, as the system is assembled out of smaller components, and permits the modeling of systems with a dynamic network topology or with a varying number of interacting processes.

Linear logic. Shortly after its introduction, linear logic has been proposed as a specification language suitable for concurrency. Following this idea, Kobayashi and Yonezawa [37, 38] have studied formal models of concurrent objects and actors based on linear logic. More recently, a direct correspondence between propositions of linear logic and session types has been discovered [6, 55, 39]. There are several analogies between the mailbox type system and the proof system of linear logic. Mailbox types with output capability are akin to positive propositions, with $\mathtt{!}\mathbb{0}$ and $\mathtt{!}\mathbb{1}$ respectively playing the roles of 0 and 1 in linear logic and $\mathtt{!}(E+F)$ and $\mathtt{!}(E \cdot F)$ corresponding to $\oplus$ and $\otimes$. Mailbox types with input capability are akin to negative propositions, with $\mathtt{?}\mathbb{0}$ and $\mathtt{?}\mathbb{1}$ corresponding to $\top$ and $\bot$ and $\mathtt{?}(E+F)$ and $\mathtt{?}(E \cdot F)$ corresponding to & and $\parr$. Rules [T-FAIL], [T-FREE] and [T-BRANCH] have been directly inspired from the rules for $\top$, $\bot$ and & in the classical sequent calculus for linear logic. Subtyping corresponds to inverse linear implication and its properties are consistent with those of the logic connectives according to the above interpretation.

Deadlock freedom. There is a vast literature on type systems ensuring deadlock freedom (or stronger properties) of communicating processes and/or concurrent objects. These are based on various mechanisms, including dependency relations between channels, sessions or objects [12, 36, 45], process types [30] and behavioral types [42, 34, 44]. Hüttel *et al.* [29] survey most of these techniques. Our approach is based on the idea of enforcing an acyclic network topology and has been inspired by the session type systems based on linear logic [6, 55, 39]. Interestingly, these works do not require any additional mechanism to ensure deadlock freedom. There are two reasons that call for dependency graphs in our setting. First, the rule [T-PAR] is akin to a symmetric cut rule. Dependency graphs are necessary to detect mutual dependencies that may consequently arise (Example 21). Second, unlike session endpoints, mailbox references can be used non-linearly. Thus, the multiplicity of dependencies, and not just the presence or lack thereof, is relevant (Example 13).

Igarashi and Kobayashi [30] study a *generic* type system for the π-calculus that allows the enforcement of various safety properties, among which deadlock freedom. Unlike our approach, which is based on typing mailboxes, their approach associates types with processes. Process types collect information about *both* the messages exchanged over channels *as well as* the dependencies between them, potentially achieving better precision in the analysis. This also means, however, that the properties of a system are established by a global check on the type of the system as a whole, which may hinder compositional reasoning. Determining whether their type system subsumes our own is non-trivial and left for future work.

As a final consideration, it is easy to see from the typing rules and the structure of judgments that dependency graphs are completely orthogonal to the other components of the type system. This makes it possible to remove or replace them with more fine-grained mechanisms if desired/appropriate. For example, some of the aforementioned works [36, 45] are able to establish deadlock freedom of some cyclic network topologies. It might be interesting to see whether and how these may be applied in our setting.

6 Concluding Remarks

We have presented a mailbox type system for reasoning about processes that communicate through first-class, unordered mailboxes. The type system enforces mailbox conformance, deadlock freedom and, for a significant class of processes, junk freedom as well. In sharp contrast with session types, mailbox types embody the unordered nature of mailboxes and enable the description of mailboxes concurrently accessed by several processes, abstracting away from the state and behavior of the processes using these mailboxes. The fact that a mailbox may have different types during its lifetime is entirely encapsulated by the typing rules and not apparent from mailbox types themselves. The mailbox calculus subsumes the actor model and allows us to analyze systems with a dynamic network topology and a varying number of processes mixing different concurrency abstractions.

In the associated technical report [17] we informally discuss how to relax the syntax of guarded processes to accommodate actions referring to different mailboxes as well as actions representing timeouts. This extension makes the typing rules for guards more complex to formulate but enhances expressiveness and precision of typing. As a further extension, it is also possible to allow multiple processes to receive messages from the same mailbox.

Concerning further developments, the intriguing analogies between the mailbox type system and linear logic pointed out in Section 5 surely deserve a more thorough investigation. On the practical side, a primary goal is the application of the proposed typing discipline to real-world programs based on the actor model and extensions thereof. In this respect, one promising approach is the development of a tool for the analysis of Java bytecode along the lines of what has already been done for Kilim [52]. Meanwhile, we have derived an algorithmic version of the typing rules (Table 2) and developed a proof-of-concept tool that applies the proposed typing discipline to the mailbox calculus [46]. The fact that each occurrence of a name might be typed differently calls for a non-trivial amount of type inference as well. To this aim, we make use of *pattern variables* to denote unknown patterns, we *generate constraints* involving these variables from the structure of the process being analyzed, and finally we look for a *solution* of the obtained constraints. This latter phase requires solving systems of inequations in a commutative Kleene algebra, for which we appeal to a particular instance of Newtonian program analysis [18] first introduced by Hopkins and Kozen [28].

References

1 Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

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*, 3:95–230, 2016. doi:10.1561/2500000031.

3 Joe Armstrong. *Programming Erlang: Software for a Concurrent World.* Pragmatic Bookshelf, 2013.

4 Janusz A. Brzozowski. Derivatives of Regular Expressions. *Journal of ACM*, 11(4):481–494, 1964. doi:10.1145/321239.321249.

5 Luís Caires and Jorge A. Pérez. Multiparty session types within a canonical binary theory, and beyond. In *Proceedings of FORTE'16*, LNCS 9688, pages 74–95. Springer, 2016. doi:10.1007/978-3-319-39570-8_6.

6 Luís Caires and Frank Pfenning. Session Types as Intuitionistic Linear Propositions. In *Proceedings of CONCUR'10*, LNCS 6269, pages 222–236. Springer, 2010. doi:10.1007/978-3-642-15375-4_16.

7 Minas Charalambides, Peter Dinges, and Gul A. Agha. Parameterized, concurrent session types for asynchronous multi-actor interactions. *Science of Computer Programming*, 115-116:100–126, 2016. doi:10.1016/j.scico.2015.10.006.

8 Dominik Charousset, Raphael Hiesgen, and Thomas C. Schmidt. Revisiting actor programming in C++. *Computer Languages, Systems & Structures*, 45:105–131, 2016. doi:10.1016/j.cl.2016.01.002.

9 Arghya Chatterjee, Branko Gvoka, Bing Xue, Zoran Budimlic, Shams Imam, and Vivek Sarkar. A distributed selectors runtime system for java applications. In *Proceedings of PPPJ'16*, pages 3:1–3:11. ACM, 2016. doi:10.1145/2972206.2972215.

10 Maria Christakis and Konstantinos Sagonas. Detection of asynchronous message passing errors using static analysis. In *Proceedings of PADL'11*, LNCS 6539, pages 5–18. Springer, 2011. doi:10.1007/978-3-642-18378-2_3.

11 John Conway. *Regular Algebra and Finite Machines.* William Clowes & Sons Ltd, 1971.

12 Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. Global Progress for Dynamically Interleaved Multiparty Sessions. *Mathematical Structures in Computer Science*, 26:238–302, 2016. doi:10.1017/S0960129514000188.

13 Bruno Courcelle. Fundamental Properties of Infinite Trees. *Theoretical Computer Science*, 25:95–169, 1983. doi:10.1016/0304-3975(83)90059-2.

14 Silvia Crafa. Behavioural types for actor systems. Technical Report 1206.1687, arXiv, 2012. URL: http://arxiv.org/abs/1206.1687.

15 Silvia Crafa and Luca Padovani. The Chemical Approach to Typestate-Oriented Programming. *ACM Transactions on Programming Languages and Systems*, 39:13:1–13:45, 2017. doi:10.1145/3064849.

16 Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. *Information and Computation*, 256:253–286, 2017. doi:10.1016/j.ic.2017.06.002.

17 Ugo de'Liguoro and Luca Padovani. Mailbox types for unordered interactions. *CoRR*, abs/1801.04167, 2018. arXiv:1801.04167.

18 Javier Esparza, Stefan Kiefer, and Michael Luttenberger. Newtonian program analysis. *Journal of the ACM*, 57(6):33:1–33:47, 2010. doi:10.1145/1857914.1857917.

19 Simon Fowler. An Erlang implementation of multiparty session actors. In *Proceedings of ICE'16*, EPTCS 223, pages 36–50, 2016. doi:10.4204/EPTCS.223.3.

20 Simon Fowler, Sam Lindley, and Philip Wadler. Mixing metaphors: Actors as channels and channels as actors. In *Proceedings of ECOOP'17*, LIPIcs 74, pages 11:1–11:28, 2017. `doi:10.4230/LIPIcs.ECOOP.2017.11`.

21 Simon J. Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2-3):191–225, 2005. `doi:10.1007/s00236-005-0177-z`.

22 Elena Giachino, Ludovic Henrio, Cosimo Laneve, and Vincenzo Mastandrea. Actors may synchronize, safely! In *Proceedings PPDP'16*, pages 118–131. ACM, 2016. `doi:10.1145/2967973.2968599`.

23 Philipp Haller. On the integration of the actor model in mainstream technologies: the scala perspective. In *Proceedings of AGERE! 2012*, pages 1–6. ACM, 2012. `doi:10.1145/2414639.2414641`.

24 Philipp Haller and Frank Sommers. *Actors in Scala - concurrent programming for the multi-core era*. Artima, 2011.

25 Jiansen He, Philip Wadler, and Philip Trinder. Typecasting actors: From akka to takka. In *Proceedings of the Fifth Annual Scala Workshop (SCALA'14)*, pages 23–33. ACM, 2014. `doi:10.1145/2637647.2637651`.

26 Carl Hewitt, Peter Bishop, and Richard Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of IJCAI'73*, pages 235–245. William Kaufmann, 1973.

27 Kohei Honda. Types for Dyadic Interaction. In *Proceedings of CONCUR'93*, volume LNCS 715, pages 509–523. Springer, 1993. `doi:10.1007/3-540-57208-2_35`.

28 Mark W. Hopkins and Dexter Kozen. Parikh's Theorem in Commutative Kleene Algebra. In *Proceedings of LICS'99*, pages 394–401. IEEE, 1999. `doi:10.1109/LICS.1999.782634`.

29 Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of Session Types and Behavioural Contracts. *ACM Computing Surveys*, 49(1):3:1–3:36, 2016. `doi:10.1145/2873052`.

30 Atsushi Igarashi and Naoki Kobayashi. A generic type system for the pi-calculus. *Theoretical Computer Science*, 311(1-3):121–163, 2004. `doi:10.1016/S0304-3975(03)00325-6`.

31 Shams Mahmood Imam and Vivek Sarkar. Integrating task parallelism with actors. *SIGPLAN Notices*, 47(10):753–772, 2012. `doi:10.1145/2398857.2384671`.

32 Shams Mahmood Imam and Vivek Sarkar. Savina - an actor benchmark suite: Enabling empirical evaluation of actor libraries. In *Proceedings of AGERE! 2014*, pages 67–80. ACM, 2014. `doi:10.1145/2687357.2687368`.

33 Shams Mahmood Imam and Vivek Sarkar. Selectors: Actors with multiple guarded mailboxes. In *Proceedings of AGERE! 2014*, pages 1–14. ACM, 2014. `doi:10.1145/2687357.2687360`.

34 Naoki Kobayashi. A Type System for Lock-Free Processes. *Information and Computation*, 177(2):122–159, 2002. `doi:10.1006/inco.2002.3171`.

35 Naoki Kobayashi. Type systems for concurrent programs. Technical report, Tohoku University, 2007. Short version appeared in 10th Anniversary Colloquium of UNU/IIST, 2002. URL: `http://www.kb.ecei.tohoku.ac.jp/~koba/papers/tutorial-type-extended.pdf`.

36 Naoki Kobayashi and Cosimo Laneve. Deadlock analysis of unbounded process networks. *Information and Computation*, 252:48–70, 2017. `doi:10.1016/j.ic.2016.03.004`.

37 Naoki Kobayashi and Akinori Yonezawa. Type-theoretic foundations for concurrent object-oriented programming. In *Proceedings of OOPSLA'94*, pages 31–45. ACM, 1994. `doi:10.1145/191080.191088`.

38 Naoki Kobayashi and Akinori Yonezawa. Asynchronous communication model based on linear logic. *Formal Aspects of Computing*, 7(2):113–149, 1995. `doi:10.1007/BF01211602`.

39 Sam Lindley and J. Garrett Morris. A semantics for propositions as sessions. In *Proceedings of ESOP'15*, LNCS 9032, pages 560–584. Springer, 2015. doi:10.1007/978-3-662-46669-8_23.

40 Vincenzo Mastandrea. Deadlock analysis with behavioral types for actors. In *Proceedings of ICTCS'16*, volume 1720 of *CEUR Workshop Proceedings*, pages 257–262, 2016. URL: http://ceur-ws.org/Vol-1720/short7.pdf.

41 Dimitris Mostrous and Vasco T. Vasconcelos. Session typing for a featherweight Erlang. In *Proceedings of COORDINATION'11*, LNCS 6721, pages 95–109. Springer, 2011. doi:10.1007/978-3-642-21464-6_7.

42 Elie Najm, Abdelkrim Nimour, and Jean-Bernard Stefani. Guaranteeing liveness in an object calculus through behavioural typing. In *Proceedings of FORTE'99*, volume 156, pages 203–221. Kluwer, 1999.

43 Rumyana Neykova and Nobuko Yoshida. Multiparty session actors. *Logical Methods in Computer Science*, 13(1), 2017. doi:10.23638/LMCS-13(1:17)2017.

44 Luca Padovani. Deadlock and Lock Freedom in the Linear π-Calculus. In *Proceedings of CSL-LICS'14*, pages 72:1–72:10. ACM, 2014. doi:10.1145/2603088.2603116.

45 Luca Padovani. Deadlock-Free Typestate-Oriented Programming. *Programming Journal*, 2, 2018. doi:10.22152/programming-journal.org/2018/2/15.

46 Luca Padovani. MC^2, the Mailbox Calculus Checker, 2018. URL: http://www.di.unito.it/~padovani/Software/MCC/index.html.

47 Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–453, 1996.

48 Franz Puntigam. Strong types for coordinating active objects. *Concurrency and Computation: Practice and Experience*, 13(4):293–326, 2001. doi:10.1002/cpe.570.

49 Franz Puntigam and Christof Peter. Types for active objects with static deadlock prevention. *Fundamenta Informaticae*, 48(4):315–341, 2001. URL: http://content.iospress.com/articles/fundamenta-informaticae/fi48-4-02.

50 António Ravara and Vasco T. Vasconcelos. Typing non-uniform concurrent objects. In *Proceedings of CONCUR'00*, LNCS 1877, pages 474–488. Springer, 2000. doi:10.1007/3-540-44618-4_34.

51 Davide Sangiorgi and David Walker. *The Pi-Calculus - A theory of mobile processes.* Cambridge University Press, 2001.

52 Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-typed actors for java. In *Proceedings of ECOOP'08*, LNCS 5142, pages 104–128. Springer, 2008. doi:10.1007/978-3-540-70592-5_6.

53 Samira Tasharofi, Peter Dinges, and Ralph E. Johnson. Why do scala developers mix the actor model with other concurrency models? In *Proceedings of ECOOP'13*, LNCS 7920, pages 302–326. Springer, 2013. doi:10.1007/978-3-642-39038-8_13.

54 Carlos A. Varela and Gul Agha. Programming dynamically reconfigurable open systems with SALSA. *SIGPLAN Notices*, 36(12):20–34, 2001. doi:10.1145/583960.583964.

55 Philip Wadler. Propositions as sessions. *Journal of Functional Programming*, 24(2-3):384–418, 2014. doi:10.1017/S095679681400001X.