Special Delivery

Programming with Mailbox Types

SIMON FOWLER, University of Glasgow, UK
DUNCAN PAUL ATTARD, University of Glasgow, UK
FRANCISZEK SOWUL, University of Glasgow, UK
SIMON J. GAY, University of Glasgow, UK
PHIL TRINDER, University of Glasgow, UK

The asynchronous and unidirectional communication model supported by *mailboxes* is a key reason for the success of actor languages like Erlang and Elixir for implementing reliable and scalable distributed systems. While many actors may *send* messages to some actor, only the actor may (selectively) *receive* from its mailbox. Although actors eliminate many of the issues stemming from shared memory concurrency, they remain vulnerable to communication errors such as protocol violations and deadlocks.

Mailbox types are a novel behavioural type system for mailboxes first introduced for a process calculus by de'Liguoro and Padovani in 2018, which capture the contents of a mailbox as a commutative regular expression. Due to aliasing and nested evaluation contexts, moving from a process calculus to a programming language is challenging. This paper presents Pat, the first programming language design incorporating mailbox types, and describes an algorithmic type system. We make essential use of quasi-linear typing to tame some of the complexity introduced by aliasing. Our algorithmic type system is necessarily co-contextual, achieved through a novel use of backwards bidirectional typing, and we prove it sound and complete with respect to our declarative type system. We implement a prototype type checker, and use it to demonstrate the expressiveness of Pat on a factory automation case study and a series of examples from the Savina actor benchmark suite.

CCS Concepts: • **Software and its engineering → Concurrent programming languages**; **Functional languages**.

Additional Key Words and Phrases: mailbox types, quasi-linear types, actor languages

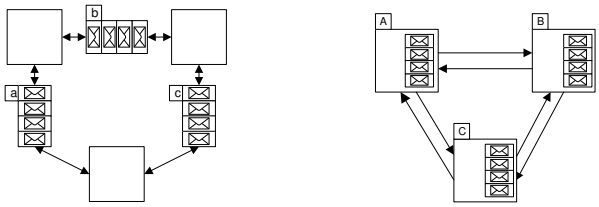
ACM Reference Format:
Simon Fowler, Duncan Paul Attard, Franciszek Sowul, Simon J. Gay, and Phil Trinder. 2023. Special Delivery: Programming with Mailbox Types. *Proc. ACM Program. Lang.* 7, ICFP, Article 191 (August 2023), 30 pages. https://doi.org/10.1145/3607832

1 INTRODUCTION

Software is increasingly concurrent and distributed, but coordinating concurrent computations introduces a host of additional correctness issues like communication mismatches and deadlocks. Communication-centric languages such as Go, Erlang, and Elixir make it possible to avoid many of the issues stemming from shared memory concurrency by structuring applications as lightweight processes that communicate through explicit message passing. There are two main classes of

Authors' addresses: Simon Fowler, University of Glasgow, UK, simon.fowler@glasgow.ac.uk; Duncan Paul Attard, University of Glasgow, UK, duncan.attard@glasgow.ac.uk; Franciszek Sowul, University of Glasgow, UK, 2482997S@student.gla.ac.uk; Simon J. Gay, simon.gay@glasgow.ac.uk, University of Glasgow, UK; Phil Trinder, University of Glasgow, UK, phil.trinder@glasgow.ac.uk.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2023 Copyright held by the owner/author(s).
2475-1421/2023/8-ART191
https://doi.org/10.1145/3607832



(a) Channels (b) Actors

Fig. 1. Channel- and actor-based languages [Fowler et al. 2017]

communication-centric language. In *channel-based* languages like Go or Rust, processes communicate over channels, where a `send` in one process is paired with a `receive` in the recipient process. In *actor* languages like Erlang or Elixir, a message is sent to the *mailbox* of the recipient process, which is an incoming message queue; in certain actor languages, the recipient can choose which message from the mailbox to handle next.

Although communication-centric languages eliminate many coordination issues, some remain. For example, a process may still receive a message that it is not equipped to handle, or wait for a message that it will never receive. Such communication errors often occur sporadically and unpredictably after deployment, making them difficult to locate and fix.

Behavioural type systems [Hüttel et al. 2016] encode correct communication behaviour to support *correct-by-construction* concurrency. Behavioural type systems, in particular *session types* [Honda 1993; Honda et al. 1998; Takeuchi et al. 1994], have been extensively applied to specify communication protocols in channel-based languages [Ancona et al. 2016]. There has, however, been far less application of behavioural typing to actor languages. Existing work either imposes restrictions on the actor model to retrofit session types [Harvey et al. 2021; Mostrous and Vasconcelos 2011; Tabone and Francalanza 2021, 2022] or relies on dynamic typing [Neykova and Yoshida 2017b]. We discuss these systems further in §7.

Our approach is based on *mailbox types*, a behavioural type system for mailboxes first introduced in the context of a process calculus [de'Liguoro and Padovani 2018]. We present the first programming language design incorporating mailbox types and we detail an algorithmic type system, an implementation, and a range of benchmarks and a factory case study. Due to aliasing and nested evaluation contexts, the move from a process calculus to a programming language is challenging. We make essential and novel use of quasi-linear typing [Ennals et al. 2004; Kobayashi 1999] to tame some of the complexity introduced by aliasing, and our algorithmic type system is necessarily co-contextual [Erdweg et al. 2015; Kuci et al. 2017], achieved through a novel use of backwards bidirectional typing [Zeilberger 2015].

1.1 Channel vs. Actor Communication

Channel-based languages comprise anonymous processes that communicate over named channels, whereas actor-based languages comprise named processes each equipped with a mailbox. Figure 1 contrasts the approaches, and is taken from a detailed comparison [Fowler et al. 2017].

Actor languages have proven to be effective for implementing reliable and scalable distributed systems [Trinder et al. 2017]. Communication in actor languages is asynchronous and unidirectional: many actors may *send* messages to an actor A, whereas only A may *receive* from its mailbox. Mailboxes provide *data locality* as each message is stored with the process that will handle it. Since channel-based languages allow channel names to be sent, they must either sacrifice locality and reduce performance, or rely on complex distributed algorithms [Chaudhuri 2009; Hu et al. 2008].

Although it is straightforward to add a type system to channel-based languages, adding a type system to actor languages is less straightforward, as process names (process IDs or PIDs) must be parameterised by a type that supports all messages that can be received. The type is therefore less precise, requiring subtyping [He et al. 2014] or synchronisation [de Boer et al. 2007; Tasharofi et al. 2013] to avoid a total loss of modularity [Fowler et al. 2017].

The situation becomes even more pronounced when considering behavioural type systems: communication errors might be prevented by giving one end of a channel the session type !Int.!Int.?Bool.End (send two integers, and receive a Boolean), and the other end the *dual* type ?Int.?Int.!Bool.End. Behavioural type systems for actor languages are much less straightforward due to the asymmetric communication model. In practice, designers of session type systems for actor languages either emulate session-typed channels [Mostrous and Vasconcelos 2011], or use *multiparty session types* to govern the communication actions performed by a process, requiring a fixed communication topology [Neykova and Yoshida 2017b].

1.2 Mailbox Types

de'Liguoro and Padovani [2018] observe that session types require a *strict ordering* of messages, whereas most actor systems use *selective receive* to process messages out-of-order. Concentrating on *unordered* interactions enables behavioural typing for mailboxes with many writers.

Mailbox typing by example: a future variable. Rather than reasoning about the *behaviour of a process*, mailbox types reason about the *contents of a mailbox*. Consider a *future variable*, which is a placeholder in a concurrent computation. A future can receive many `get` messages that are only fulfilled *after* a `put` message initialises the future with a value. After the future is initialised, it fulfils all `get` messages by sending its value; a second `put` message is explicitly disallowed. We can implement a future straightforwardly in Erlang:

```
1  empty_future() ->
2    receive
3      { put, X } -> full_future(X)
4    end.
5  full_future(X) ->
6    receive
7      { get, Pid } ->
8        Pid ! { reply, X },
9        full_future(X);
10     { put, _ } ->
11       erlang:error("Multiple writes")
12   end.
13 client() ->
14   Future = spawn(future, empty_future, []),
15   Future ! { put, 5 },
16   Future ! { get, self() },
17   receive
18     { reply, Result } ->
19       io:fwrite("~w~n", [Result])
20   end.
```

The `empty_future` function awaits a `put` message to set the value of the future (lines 2–4), and transitions to the `full_future` state. A `full_future` receives `get` messages (lines 6–12) containing a process ID used to reply with the future's value. The `client` function spawns a future (line 14), sends a `put` message followed by a `get` message (lines 15–16), and awaits the result (lines 17–20). The program prints the number 5.

Several communication errors can arise in this example:

- **Protocol violation.** Sending two `put` messages to the future will result in a *runtime* error.
- **Unexpected message.** Sending a message other than `get` or `put` to the future will silently succeed, but the message will never be retrieved, resulting in a memory leak.
- **Forgotten reply.** If the future fails to send a `reply` message after receiving a `get` message the client will be left waiting forever.
- **Self-deadlock.** If the client attempts to receive a `reply` message before sending a `get` message it will be left waiting forever.

All of the above issues can be solved by mailbox typing. We can write the following types:

$$\begin{aligned} \mathsf{EmptyFuture} &\triangleq ?(\mathtt{Put}[\mathsf{Int}] \odot \star\mathtt{Get}[\mathsf{ClientSend}]) \\ \mathsf{FullFuture} &\triangleq ?\star\mathtt{Get}[\mathsf{ClientSend}] \end{aligned} \qquad \begin{aligned} \mathsf{ClientSend} &\triangleq !\mathtt{Reply}[\mathsf{Int}] \\ \mathsf{ClientRecv} &\triangleq ?\mathtt{Reply}[\mathsf{Int}] \end{aligned}$$

A mailbox type combines a *capability* (either ! for an output capability, analogous to a PID in Erlang; or ? for an input capability) with a *pattern*. A pattern is a *commutative regular expression*: in the context of a send mailbox type, the pattern will describe the messages that must be sent; in the context of a receive mailbox type, it describes the messages that the mailbox may contain.

A mailbox name (*e.g.*, `Future`) may have different types at different points in the program. EmptyFuture types an input capability of an empty future mailbox, and denotes that the mailbox may contain a single **Put** message with an Int payload, and potentially many ($\star$) **Get** messages each with a ClientSend payload. FullFuture types an input capability of the future after a **Put** message has been received, and requires that the mailbox *only* contains **Get** messages. ClientSend is an output mailbox type which requires that a **Reply** message must be sent; ClientRecv is an input capability for receiving the **Reply**. For each mailbox name, sends and receives must "balance out": if a message is sent, it must eventually be received.

de'Liguoro and Padovani [2018] introduce a small extension of the asynchronous π-calculus [Amadio et al. 1998], which they call the *mailbox calculus*, and endow it with mailbox types. They express the Future example in the mailbox calculus as follows, where the mailbox is denoted *self*.

$$\begin{aligned} \mathsf{emptyFuture}(\mathit{self}) &\triangleq \mathit{self}?\mathtt{Put}(x)\,.\,\mathsf{fullFuture}(\mathit{self}, x) \\ \mathsf{fullFuture}(\mathit{self}, x) &\triangleq \mathbf{free}\ \mathit{self}\,.\,\mathbf{done} \\ &\quad + \mathit{self}?\mathtt{Get}(\mathit{sender})\,.\,(\mathit{sender}!\mathtt{Reply}[x] \parallel \mathsf{fullFuture}(\mathit{self}, x)) \\ &\quad + \mathit{self}?\,\mathtt{Put}(x)\,.\,\mathbf{fail}\ \mathit{self} \end{aligned}$$

$$\begin{aligned} &(\nu \mathit{future})(\mathsf{emptyFuture}(\mathit{future}) \parallel \mathit{future}!\mathtt{Put}[5] \parallel \\ &\qquad (\nu \mathit{self})(\mathit{future}!\mathtt{Get}[\mathit{self}] \parallel (\mathit{self}?\,\mathtt{Reply}(x)\,.\,\mathbf{free}\ \mathit{self}.\,\mathsf{print}(\mathsf{intToString}(x))) \end{aligned}$$

A process calculus is useful for expressing the essence of concurrent computation, but there is a large gap between a process calculus and a programming language design, the biggest being the separation of static and dynamic terms. A programming language specifies the *program* that a user writes, whereas a process calculus provides a snapshot of the system at a given time. A particular difference comes with name generation: in a process calculus, we can write name restrictions directly; in a programming language, we instead have a language construct (like **new**) which is evaluated to create a fresh name at runtime. Further complexities come with nested evaluation contexts, sequential evaluation, and aliasing. We explore these challenges in greater detail in §2.

We propose Pat[1], a first-order programming language that supports mailbox types, in which we express the *future* example as follows (*self* is again the mailbox).

```
def emptyFuture(self: EmptyFuture): 1 {
  guard self: Put ⊙ ★Get {
    receive Put[x] from self ↦ fullFuture(self, x)
  }
}

def fullFuture(self: FullFuture, value : Int): 1 {
  guard self: ★Get {
    free ↦ ()
    receive Get[user] from self ↦
      user!Reply[value];
      fullFuture(self, value)
  }
}
```

```
def client(): 1 {
  let future = new in
  spawn emptyFuture(future);
  let self = new in
  future!Put[5];
  future!Get[self];
  guard self: Reply {
    receive Reply[result] from self ↦
      free self;
      print(intToString(result))
  }
}
```

[1] https://en.wikipedia.org/wiki/Postman_Pat

The Pat program has a similar structure to the Erlang example with client, emptyFuture and fullFuture functions, and the mailbox types are similar to those in the mailbox calculus specification. There are, however, some differences compared with the Erlang future. The first is that in Pat mailboxes are first-class: we create a new mailbox with **new**, and receive from it using the **guard** expression. A **guard** acts on a mailbox and may contain several guards: **free** $\mapsto M$ frees the mailbox if there are no other references to it and evaluates M; and **receive** $\mathtt{m}[\overrightarrow{x}]$ **from** $y \mapsto M$ retrieves a message with tag $\mathtt{m}$ from the mailbox, binding its payloads to $\overrightarrow{x}$ and re-binding the mailbox variable (with an updated type) to y in continuation M. There is also **fail** denoting that a mailbox is in an invalid state, but the type system ensures that this guard is never evaluated. In the above code, **free** *self* is syntactic sugar (see §3).

Pat has all of the characteristics of a programming language, unlike the mailbox calculus. Static and dynamic terms are distinguished, *i.e.*, we *do not* need to write name restrictions with dynamic names known *a priori*. Pat provides **let**-bindings, which enable full sequential composition along with nested evaluation contexts; and we have data types and return types. Crucially *all of the concurrency errors described earlier result in a type error*, i.e. protocol violations, unexpected messages, forgotten replies, and self-deadlocks are all detected statically.

Contributions. Despite being a convincing proposal for behavioural typing for actor languages, mailbox typing has received little attention since its introduction in 2018. The overarching contribution of this paper, therefore, is the first design and implementation of a concurrent programming language with support for mailbox types. Concretely, we make four main contributions:

(1) We introduce a declarative type system for Pat (§3), a first-order programming language with support for mailbox types, making essential and novel use of quasi-linear types. We show type preservation, mailbox conformance, and a progress result.
(2) We introduce a co-contextual algorithmic type system for Pat (§4), making use of backwards bidirectional typing. We prove that the algorithmic type system is sound and complete with respect to the declarative type system.
(3) We extend Pat with sum and product types; interfaces; and higher-order functions (§5).
(4) We detail our implementation (§6), and demonstrate the expressiveness of Pat by encoding all of the examples from de'Liguoro and Padovani [2018], and 10 of the 11 Savina benchmarks [Imam and Sarkar 2014] used by Neykova and Yoshida [2017b] in their evaluation of multiparty session types for actor languages (§6.2).

Our tool is available as an artifact and on GitHub (https://www.github.com/SimonJF/mbcheck). The extended version [Fowler et al. 2023b] contains further details and full proofs.

2 MAILBOX TYPES IN A PROGRAMMING LANGUAGE: WHAT ARE THE ISSUES?

```
1  def client(): 1 {
2    let future = new in
3    spawn emptyFuture(future);
4    let self = new in
5    future!Put[5]; future!Get[self];
6    guard self: Reply {
7      receive Reply[result] from self ↦
8        free self;
9        print(intToString(result))
10   }
11 }
```

Fig. 2. Send and receive uses of *future*

Session typing was originally studied in the context of process calculi (e.g., [Honda et al. 1998; Vasconcelos 2012]), but later work [Fowler et al. 2023c; Gay and Vasconcelos 2010; Wadler 2014] introduced session types for languages based on the linear λ-calculus. The more relaxed view of linearity in the mailbox calculus makes language integration far more challenging. A mailbox name may be used several times to *send* messages, but only once to *receive* a message. The intuition is that while sends simply add messages to a mailbox, it is a receive that determines the future behaviour of the actor. To illustrate, Fig. 2 shows a fragment of the future example from §1 with two sends to the *future* mailbox (lines 5 and

```
def useAfterFree1(x : ?★Msg[1]): 1 {
  guard x : ★Msg {
    receive Msg[y] from z ↦
      x!Msg[];
      useAfterFree1(z)
    free ↦ x!Msg[]
  }
}
```

(a) Using old name

```
def useAfterFree2(x : ?★Msg[1]): 1 {
  let a = x in
  guard a : ★Msg {
    receive Msg[y] from z ↦
      x!Msg[];
      useAfterFree2(z)
    free ↦ x!Msg[]
  }
}
```

(b) Renaming

```
def useAfterFree3(x : ?★Msg[1]): 1 {
  let _ =
    guard x : ★Msg {
      receive Msg[y] from z ↦
        x!Msg[];
        useAfterFree3(z)
      free ↦ x!Msg[]
    } in x!Msg[]
}
```

(c) Evaluation contexts

Fig. 3. Use-after-free via aliasing

6), and a single receive (line 8). In the mailbox calculus, a name remains constant and cannot be aliased; this is at odds with idiomatic programming where expressions are aliased with **let** bindings or function application. Moreover functional languages provide nested evaluation contexts and sequential evaluation.

2.1 Challenge: Mailbox Name Aliasing

Ensuring appropriate mailbox use is challenging in the presence of aliasing: for example, we can write a function that attempts to use a mailbox after it has been freed (Fig. 3a). Unlike in our situation, use-after-free errors are not an issue with a fully linear type system, since we *cannot* use a resource after it has been consumed.

We could require that a name cannot be used after it has been guarded upon by insisting that the subject and body of a **guard** expression are typable under disjoint type environments. Indeed, such an approach correctly rules out the error in Fig. 3a, but the check can easily be circumvented. Fig. 3b aliases the output capability for the mailbox, and the new name prevents the typechecker from realising that it has been used in the body of the guard. Similarly, Fig. 3c uses nested evaluation contexts, meaning that the next use of a mailbox variable is not necessarily contained within a subexpression of the guard.

Much of the intricacy arises from using a mailbox name many times as an output capability. In each process, we can avoid the problems above using three principles:

(1) No two distinct variables should represent the same underlying mailbox name.
(2) Once let-bound to a different name, a mailbox variable is considered out-of scope.
(3) A mailbox name cannot be used after it has been used in a **guard** expression.

These principles ensure syntactic hygiene: the first and second handle the disconnect between static names and their dynamic counterparts, allowing us to reason that two syntactically distinct variables indeed refer to different mailboxes. The third ensures that a mailbox name is correctly 'consumed' by a **guard** expression, allowing us to correctly update its type.

Aliasing through communication. Consider the following example, where mailbox a receives the message $\mathtt{m}[b]$, where b is already free in the continuation of the **receive** clause:

$$a \leftarrow \mathtt{m}[b] \quad \| \quad \begin{array}{l} \textbf{guard}\ a : \mathtt{m}\ \{ \\ \quad \textbf{receive}\ \mathtt{m}[x]\ \textbf{from}\ y \mapsto \\ \quad\quad b\,!\,\mathtt{n}[x]; \\ \quad\quad \textbf{free}\ y \\ \} \end{array} \quad \longrightarrow \quad \begin{array}{l} b\,!\,\mathtt{n}[b]; \\ \textbf{free}\ a \end{array}$$

Here, although the code suggests that x and b are distinct, aliasing is introduced through communication (violating principle 1).

2.2 Mailbox Calculus Solution: Dependency Graphs

The mailbox calculus uses a *dependency graph* (DG) both to avoid issues with aliasing and to eliminate cyclic dependencies and hence deadlocks. As an example, the mailbox calculus process $(\nu a)(\nu b)(a\,!\,\mathtt{m}[b] \parallel \textbf{free}\ b \parallel a?\mathtt{m}[x]\,.\,\textbf{free}\ a)$ would have DG $(\nu a)(\nu b)(\{a, b\})$ due to the dependency arising from sending b over a.

```
let a = new in
let b = new in
a!m[b]; spawn (free b);
guard a : m {
  receive m[x] from a ↦ free a
}
```

Alas, a language implementation *cannot* use this approach as it relies on knowing runtime names directly. To see why, consider the Pat program on the left, which evaluates to an analogous configuration. The first issue is how to create a scoped DG from **new**: one option is to introduce a scoped construct **let mailbox** x **in** M, but this approach fails as soon as we rename x using a **let**-binder. A more robust approach is to follow Ahmed et al. [2007] and Padovani [2019] and endow mailbox types with a type-level identity by giving **new** an existential type and introducing a scoped **unpack** construct. However there still remain two issues: first, it is unclear how to extend DGs to capture the more complex scoping and sequencing induced by nested contexts. Second, each mailbox type would require an identity (*e.g.* $!'\mathtt{Msg}$) which becomes too restrictive, since we would need to include identities in message payload types when communicating names. As an example, each client of the Future example from §1 would require a separate message type.

2.3 The Pat Solution: Quasi-Linear Typing

The many-sender, single-receiver pattern is closely linked to *quasi-linear typing* [Kobayashi 1999]; our formulation is closer to that of Ennals et al. [2004]. Quasi-linear types were originally designed to overcome some limitations of full linear types in the context of memory management and programming convenience and allow a value to be used once as a *first-class* (returnable) value, but several times as a *second-class* value [Osvald et al. 2016]. A second-class value can be *consumed* within an expression, for example as the subject of a send operation, but cannot escape the scope in which it is defined.

This distinction maps directly onto the many-writer, single-reader communication model used by the mailbox calculus. We augment mailbox types with a *usage*: either $\bullet$, a *returnable* reference that allows a type to appear in the return type of an expression; or $\circ$, a 'second-class' reference. The subject of a **guard** must be returnable. With usage information we can ensure that:

(1) there is *only one* returnable reference for each mailbox name in a process
(2) only returnable references can be renamed, avoiding problems with aliasing
(3) the returnable reference is the final lexical use of a mailbox name in a process

Quasi-linear types rule out all three of the previous examples. In useAfterFree, x is consumed by the **guard** expression and cannot be used thereafter. In useAfterFree2, since x is the subject of a **let** binding, it must be returnable and therefore cannot be used in the body of the binding. In useAfterFree3, since x is used as the subject of a **guard** expression, that use must be first-class and therefore the last lexical occurrence of x, ruling out the use of x in the outer evaluation context. Quasi-linear typing cannot account for inter-process deadlocks, but can still rule out self-deadlocks.

Ruling out aliasing through communication. Quasi-linear types alone do not safeguard against introducing aliasing through communication, and we cannot use DGs for the reasons stated above. However, treating all received names as second-class, coupled with some simple syntactic restrictions (*e.g.* by ensuring that either all message payloads or all variables free in the body of the **receive** clause have base types) eliminates unsafe aliasing.

Mailbox types	J, K	::=	$!E \mid ?E$
Mailbox patterns	E, F	::=	$\mathbb{0} \mid \mathbb{1} \mid \mathtt{m} \mid E \oplus F$
		$\mid$	$E \odot F \mid \star E$

Base types	C	::=	$\mathbf{1} \mid \mathsf{Int} \mid \mathsf{String} \mid \cdots$
Types	T, U	::=	$C \mid J$
Usage annotations	η	::=	$\circ \mid \bullet$
Usage-annotated types	A, B	::=	$C \mid J^{\eta}$

Variables	x, y, z		
Definition names	f		
Definitions	D	::=	$\mathbf{def}\, f(\overrightarrow{x : A}) : B\, \{M\}$
Values	V, W	::=	$x \mid c$
Terms	L, M, N	::=	$V \mid \mathbf{let}\, x : T = M\, \mathbf{in}\, N \mid f(\overrightarrow{V})$
		$\mid$	$\mathbf{spawn}\, M \mid \mathbf{new} \mid V\,!\,\mathtt{m}[\overrightarrow{W}] \mid \mathbf{guard}\, V : E\, \{\overrightarrow{G}\}$
Guards	G	::=	$\mathbf{fail} \mid \mathbf{free} \mapsto M \mid \mathbf{receive}\, \mathtt{m}[\overrightarrow{x}]\, \mathbf{from}\, y \mapsto M$
Type environments	Γ	::=	$\cdot \mid \Gamma, x : A$

Fig. 4. The syntax of Pat, a core language with mailbox types

Summary. Quasi-linear types and the lightweight syntactic checks outlined above ensure that mailboxes are used safely in a concurrent language that allows aliasing, and obviate the need for the static global dependency graph used in the mailbox calculus. We show that the checks are not excessively restrictive by expressing all of the examples shown by de'Liguoro and Padovani [2018], and 10 of the 11 Savina benchmarks [Imam and Sarkar 2014] used by Neykova and Yoshida [2017b] to demonstrate expressiveness of behavioural type systems for actor languages (§6.2).

3 PAT: A CORE LANGUAGE WITH MAILBOX TYPES

This section introduces Pat, a core first-order programming language with mailbox types, along with a declarative type system and an operational semantics.

3.1 Syntax

Figure 4 shows the syntax for Pat. We defer discussion of types to §3.2.

Programs and definitions. A program $(\mathcal{S}, \overrightarrow{D}, M)$ consists of a *signature* $\mathcal{S}$ which maps message tags to payload types; a set of *definitions* D; and an *initial term* M. Each definition $\mathbf{def}\, f(\overrightarrow{x : A}) : B\{M\}$ is a function with name f, annotated arguments $\overrightarrow{x : A}$, return type B, and body M. We write $\mathcal{P}(f)$ to retrieve the definition for function f, and $\mathcal{P}(\mathtt{m})$ to retrieve the payload types for message $\mathtt{m}$.

Values. It is convenient for typing to introduce a syntactic distinction between values and computations, inspired by *fine-grain call-by-value* [Levy et al. 2003]. Values V, W include variables x and constants c; we assume that the set of constants includes at least the unit value () of type $\mathbf{1}$.

Terms. The functional fragment of the language is largely standard. Every value is a term. The only evaluation context is $\mathbf{let}\, x : T = M\, \mathbf{in}\, N$, which evaluates term M of type T, binding its result to x in continuation N. The type annotation is a technical convenience and is not necessary in our implementation (§3). Function application $f(\overrightarrow{V})$ applies function f to arguments $\overrightarrow{V}$. As usual, we use $M; N$ as sugar for $\mathbf{let}\, x : \mathbf{1} = M\, \mathbf{in}\, N$, where x does not occur in N.

In the concurrent fragment of the language, $\mathbf{spawn}\, M$ spawns term M as a separate process, and $\mathbf{new}$ creates a fresh mailbox name. Term $V\,!\,\mathtt{m}[\overrightarrow{W}]$ sends message $\mathtt{m}$ with payloads $\overrightarrow{W}$ to mailbox V.

The $\mathbf{guard}\, V : E\, \{\overrightarrow{G}\}$ expression asserts that mailbox V contains pattern E, and invokes a guard in $\overrightarrow{G}$. The $\mathbf{fail}$ guard is triggered when an unexpected message has arrived; $\mathbf{free} \mapsto M$

is triggered when a mailbox is empty and there are no more references to it in the system; and **receive** $\mathtt{m}[\overrightarrow{x}]$ **from** $y \mapsto M$ is triggered when the mailbox contains a message with tag $\mathtt{m}$, binding its payloads to $\overrightarrow{x}$ and continuation mailbox with updated mailbox type to y in continuation term M. We write **free** V as syntactic sugar for **guard** $V : \mathbb{1}\ \{\textbf{free} \mapsto ()\}$, and **fail** V as syntactic sugar for **guard** $V : \mathbb{0}\ \{\textbf{fail}\}$. We require that each clause within a **guard** expression is unique.

3.2 Type System

This section describes a declarative type system for Pat. We begin by discussing mailbox types in more depth, in particular showing how to define subtyping and equivalence.

3.2.1 Types. A mailbox type consists of a *capability*, either *output* ! or *input* ?, and a *pattern*. A system can contain multiple references to a mailbox as an output capability, but only one as an input capability. A *pattern* is a *commutative* regular expression, *i.e.*, a regular expression where composition is unordered. The $\mathbb{1}$ pattern is the unit of pattern composition $\odot$, denoting the empty mailbox. The $\mathbb{0}$ pattern denotes the *unreliable* mailbox, which has received an unexpected message. It is not possible to send to, or receive from, an unreliable mailbox, but we will show that reduction does not cause a mailbox to become unreliable. The pattern $\mathtt{m}$ denotes a mailbox containing a single message $\mathtt{m}$[2]. Pattern choice $E \oplus F$ denotes that the mailbox contains either messages conforming to pattern E or F. Pattern composition $E \odot F$ denotes that the mailbox contains messages pertaining to E and F (in either order). Finally, $\star E$ denotes replication of E, so $\star\mathtt{m}$ denotes that the mailbox can contain zero or more instances of message $\mathtt{m}$. Mailbox patterns obey the usual laws of commutative regular expressions: $\mathbb{1}$ is the unit for $\odot$, while $\mathbb{0}$ is the unit for $\oplus$ and is cancelling for $\odot$. Composition $\odot$ is associative, commutative, and distributes over $\oplus$; and $\oplus$ is associative and commutative.

Pattern semantics. It follows that different syntactic representations of patterns may have the same meaning, *e.g.* patterns $\mathbb{1} \oplus \mathbb{0} \oplus (\mathtt{m} \odot \mathtt{n})$ and $\mathbb{1} \oplus (\mathtt{n} \odot \mathtt{m})$. Following [de'Liguoro and Padovani 2018], we define a set-of-multisets semantics for mailbox patterns; the intuition is that each multiset defines a configuration of messages that could be present in the mailbox. For example the semantic representation of both of the patterns above is $\{\langle\rangle, \langle \mathtt{m}, \mathtt{n}\rangle\}$. We let A, B range over multisets.

$$[\![\mathbb{0}]\!] = \emptyset \qquad [\![\mathbb{1}]\!] = \{\langle\rangle\} \qquad [\![E \oplus F]\!] = [\![E]\!] \cup [\![F]\!] \qquad [\![E \odot F]\!] = \{\mathsf{A} \uplus \mathsf{B} \mid \mathsf{A} \in [\![E]\!], \mathsf{B} \in [\![F]\!]\} \qquad [\![\mathtt{m}]\!] = \{\langle \mathtt{m}\rangle\}$$

$$[\![\star E]\!] = [\![\mathbb{1}]\!] \cup [\![E]\!] \cup [\![E \odot E]\!] \cup \cdots$$

The pattern $\mathbb{0}$ is interpreted as an empty set; $\mathbb{1}$ as the empty multiset; $\oplus$ as set union; $\odot$ as pointwise multiset union; $\mathtt{m}$ as the singleton multiset; and $\star E$ as the infinite set containing any number of concatenations of interpretations of E.

Usage annotations. A type T can be a *base type* C, or a mailbox type J. As discussed in §2, *quasi-linearity* is used to avoid aliasing issues. *Usage-annotated* types A, B annotate mailbox types with a usage: either second class ($\circ$), or returnable ($\bullet$). There are no restrictions on the use of a base type. Only values with a returnable type can be returned from an evaluation frame.

3.2.2 Operations on Types. We say that a type is *returnable*, written returnable(A), if A is a base type C or a returnable mailbox type $J^\bullet$. The $\lfloor - \rfloor$ operator ensures that a type is returnable, while the $\lceil - \rceil$ operator ensures that a mailbox type is second-class:

$$\lfloor C \rfloor = C \qquad \lfloor T \rfloor = T^\bullet \qquad \lceil C \rceil = C \qquad \lceil T \rceil = T^\circ$$

We also extend the operators to usage-annotated types (*e.g.* $\lceil J^\bullet \rceil = J^\circ$) and type environments.

[2]Unlike in §1, our formalism does not pair a message tag with its payload; instead, tags are associated with payload types via the program signature. This design choice allows us to more easily compare the declarative system with the algorithmic system in §4, and unlike [de'Liguoro and Padovani 2018] means we need not define types and subtyping coinductively.

Subtyping. With a semantics defined, we can consider subtyping. A pattern E is *included* in a pattern F, written $E \sqsubseteq F$, if every multiset in the semantics of E also occurs in the semantics of pattern F, *i.e.*, $E \sqsubseteq F \triangleq [\![E]\!] \subseteq [\![F]\!]$.

Definition 3.1 (Subtyping). The *subtyping* relation is defined by the following rules:

$$\frac{}{C \leq C} \qquad \frac{E \sqsubseteq F \qquad \eta_1 \leq \eta_2}{?E^{\eta_1} \leq ?F^{\eta_2}} \qquad \frac{F \sqsubseteq E \qquad \eta_1 \leq \eta_2}{!E^{\eta_1} \leq !F^{\eta_2}}$$

Usage subtyping is defined as the smallest reflexive operator defined by axioms $\eta \leq \eta$ and $\bullet \leq \circ$. We write $A \simeq B$ if both $A \leq B$ and $B \leq A$, *i.e.* either A, B are the same base type, or are mailbox types with the same capability and pattern semantics.

Base types are subtypes of themselves. As with previous accounts of subtyping in actor languages [He et al. 2014], subtyping is *covariant* for mailbox types with a receive capability: a mailbox can safely be replaced with another that can receive more messages. Likewise subtyping is *contravariant* for mailboxes with a send capability: a mailbox can safely be replaced with another that can send a smaller set of messages. Intuitively, as returnable usages are more powerful than second-class usages, returnable types can be used when only a second-class type is required.

Following de'Liguoro and Padovani [2018] we introduce names for particular classes of mailbox types. Intuitively, relevant mailbox names *must* be used, whereas irrelevant names need not be. Likewise reliable and usable names *can* be used, whereas unreliable and unusable names cannot.

Definition 3.2 (Relevant, Reliable, Usable). A mailbox type J is *relevant* if $J \not\leq\ !\mathbb{1}$, and *irrelevant* otherwise; *reliable* if $J \not\leq\ ?\mathbb{0}$ and *unreliable* otherwise; and *usable* if $J \not\leq\ !\mathbb{0}$ and *unusable* otherwise.

Definition 3.3 (Unrestricted and Linear Types). We say that a type A is *unrestricted*, written $\mathsf{un}(A)$, if $A = C$, or $A = !\mathbb{1}^{\circ}$. Otherwise, we say that T is *linear*.

Our type system ensures that variables with a linear type must be used, whereas variables with an unrestricted type can be discarded. We extend subtyping to type environments, making it possible to combine type environments, as in [Crafa and Padovani 2017; de'Liguoro and Padovani 2018].

Definition 3.4 (Environment subtyping). Environment subtyping $\Gamma_1 \leq \Gamma_2$ is defined as follows:

$$\frac{}{\cdot \leq \cdot} \qquad \frac{\mathsf{un}(A) \qquad x \notin \mathsf{dom}(\Gamma') \qquad \Gamma \leq \Gamma'}{\Gamma, x : A \leq \Gamma'} \qquad \frac{A \leq B \qquad \Gamma \leq \Gamma'}{\Gamma, x : A \leq \Gamma', x : B}$$

We include a notion of weakening into the subtyping relation, so an environment Γ can be a subtype environment of Γ' if it contains additional entries of unrestricted type.

Type combination. Mailbox types ensure that sends and receives "balance out", meaning that every send is matched with a receive. For example, using a mailbox at type $!\mathtt{Put}$ and $?(\mathtt{Put} \odot \star\mathtt{Get})$ results in a mailbox type $?(\star\mathtt{Get})$. The key technical device used to achieve this goal is *type combination*: combining a mailbox type $!E$ and a mailbox type $!F$ results in an output mailbox type which must send *both E and F*; combining an input and an output capability results in an input capability that no longer needs to receive the output pattern. We can also combine identical base types. Note that it is *not* possible to combine two input capabilities as this would permit simultaneous reads of the same mailbox.

Definition 3.5 (Type combination). Type combination $T \boxplus U$ is the commutative partial binary operator defined by the following axioms:

$$C \boxplus C = C \qquad !E \boxplus !F = !(E \odot F) \qquad !E \boxplus ?(E \odot F) = ?F \qquad ?(E \odot F) \boxplus !E = ?F$$

Following Crafa and Padovani [2017], it is convenient to identify types up to commutativity and associativity, *e.g.* we do not distinguish between $?(\mathsf{A} \odot \mathsf{B})^{\bullet}$ and $?(\mathsf{B} \odot \mathsf{A})^{\bullet}$. We may however need to use subtyping to rewrite a type into a form that allows two mailbox types to be combined (*e.g.* to combine $!\mathsf{A}$ and $?(\star\mathsf{A})$, we would need to use subtyping to rewrite the latter type to $?(\mathsf{A} \odot \star\mathsf{A})$).

The following *usage combination* is not commutative as a $\circ$ variable use must occur *before* a $\bullet$ use (ensuring that the returnable use is the variable's last lexical occurrence). Furthermore, note that $\bullet \triangleright \bullet$ is undefined (ensuring that there is only one returnable instance of a variable per thread).

Definition 3.6 (Usage combination). The *usage combination* operator is the partial binary operator defined by the axioms $\circ \triangleright \circ = \circ$ and $\circ \triangleright \bullet = \bullet$.

We can now define usage-annotated type and environment combination.

Definition 3.7 (Usage-annotated type combination). The *usage-annotated type combination* operator $A \triangleright B$ is the binary operator defined by the axioms $C \triangleright C = C$ and $J^{\eta_1} \triangleright K^{\eta_2} = (J \boxplus K)^{\eta_1 \triangleright \eta_2}$.

Definition 3.8 (Environment combination (Γ)). Usage-annotated environment combination $\Gamma_1 \triangleright \Gamma_2$ is the smallest partial operator on type environments closed under the following rules:

$$\frac{}{\cdot \triangleright \cdot = \cdot} \qquad \frac{x \notin \mathsf{dom}(\Gamma_2) \qquad \Gamma_1 \triangleright \Gamma_2 = \Gamma}{(\Gamma_1, x : A) \triangleright \Gamma_2 = \Gamma, x : A} \qquad \frac{x \notin \mathsf{dom}(\Gamma_1) \qquad \Gamma_1 \triangleright \Gamma_2 = \Gamma}{\Gamma_1 \triangleright (\Gamma_2, x : A) = \Gamma, x : A} \qquad \frac{\Gamma_1 \triangleright \Gamma_2 = \Gamma}{(\Gamma_1, x : A) \triangleright (\Gamma_2, x : B) = \Gamma, x : (A \triangleright B)}$$

We use usage-annotated type combination when combining the types of two variables used in subsequent evaluation frames (*i.e.* in the subject and body of a **let** expression). We also require *disjoint* combination, where two environments are only able to share variables of base type:

Definition 3.9 (Disjoint environment combination). Disjoint environment combination $\Gamma_1 + \Gamma_2$ is the smallest partial operator on type environments closed under the following rules:

$$\frac{}{\cdot + \cdot = \cdot} \qquad \frac{x \notin \mathsf{dom}(\Gamma_2) \qquad \Gamma_1 + \Gamma_2 = \Gamma}{\Gamma_1, x : A + \Gamma_2 = \Gamma, x : A} \qquad \frac{x \notin \mathsf{dom}(\Gamma_1) \qquad \Gamma_1 + \Gamma_2 = \Gamma}{\Gamma_1 + \Gamma_2, x : A = \Gamma, x : A} \qquad \frac{\Gamma_1 + \Gamma_2 = \Gamma}{\Gamma_1, x : C + \Gamma_2, x : C = \Gamma, x : C}$$

3.2.3 Typing Rules. Fig. 5 shows the declarative typing rules for Pat. As the system is declarative it helps to read the rules top-down.

Programs and definitions. A program is typable if all of its definitions are typable, and its body has unit type. A definition **def** $f(\overrightarrow{x : A})$: B $\{M\}$ is typable if M has type B under environment $\overrightarrow{x : A}$.

Terms. Term typing has the judgement $\Gamma \vdash_{\mathcal{P}} M : A$, which states that when defined in the context of program $\mathcal{P}$, under environment Γ, term M has type A. We omit the $\mathcal{P}$ parameter in the rules for readability. Rule T-Var types a variable in a singleton environment; we account for weakening in T-Sub. Rule T-Const types a constant under an empty environment; we assume an implicit schema mapping constants to types, and assume at least the unit value () of type **1**. Rule T-App types function application according to the definition in $\mathcal{P}$. Each argument must be typable under a disjoint type environment to avoid aliasing mailbox names in the body of the function.

Rule T-Let types sequential composition. The subject of the **let** expression must be returnable; since $\Gamma_1 \triangleright \Gamma_2$ is defined, we know that if the subject (typable using Γ_1) contains a returnable variable, then it cannot appear in Γ_2. This avoids aliasing and use-after-free errors.

Rule T-Spawn types spawning a term M of unit type as a new process. The type environment used to type M can contain any number of returnable, but the conclusion of the rule 'masks' any returnable types as second-class. Intuitively, this is because there is no need to impose an ordering on how a variable is used in a separate process: while within a single process a **guard** on some name x should not precede a send on x, there is no such restriction if the two expressions are executing

Typing rules for programs and definitions $\boxed{\vdash \mathcal{P}}$ $\boxed{\vdash D}$

$$\frac{\mathcal{P} = (\mathcal{S}, \overrightarrow{D}, M) \qquad (\vdash_{\mathcal{P}} D_i)_i \qquad \cdot \vdash_{\mathcal{P}} M : \mathbf{1}}{\vdash \mathcal{P}} \qquad \frac{\overrightarrow{x : A} \vdash_{\mathcal{P}} M : B}{\vdash_{\mathcal{P}} \mathbf{def}\ f(\overrightarrow{x : A}) : B\ \{M\}}$$

Typing rules for terms $\boxed{\Gamma \vdash_{\mathcal{P}} M : A}$

$$\text{T-Var}\ \frac{}{x : A \vdash x : A} \qquad \text{T-Const}\ \frac{c \text{ has base type } C}{\cdot \vdash c : C} \qquad \text{T-App}\ \frac{\mathcal{P}(f) = \mathbf{def}\ f(\overrightarrow{x : A}) : B\ \{M\} \qquad (\Gamma_i \vdash V_i : A_i)_{i \in 1..n}}{\Gamma_1 + \cdots + \Gamma_n \vdash f(V_1, \ldots, V_n) : B}$$

$$\text{T-Let}\ \frac{\Gamma_1 \vdash M : \lfloor T \rfloor \qquad \Gamma_2, x : \lfloor T \rfloor \vdash N : B}{\Gamma_1 \triangleright \Gamma_2 \vdash \mathbf{let}\ x : T = M\ \mathbf{in}\ N : B} \qquad \text{T-Spawn}\ \frac{\Gamma \vdash M : \mathbf{1}}{\lceil \Gamma \rceil \vdash \mathbf{spawn}\ M : \mathbf{1}} \qquad \text{T-New}\ \frac{}{\cdot \vdash \mathbf{new} : ?\mathbb{1}^{\bullet}}$$

$$\text{T-Send}\ \frac{\mathcal{P}(\mathtt{m}) = \overrightarrow{T} \qquad \Gamma \vdash V : !\mathtt{m}^{\circ} \qquad (\Gamma'_i \vdash W_i : \lceil T_i \rceil)_{i \in 1..n}}{\Gamma + \Gamma'_1 + \ldots + \Gamma'_n \vdash V\,!\,\mathtt{m}[\overrightarrow{W}] : \mathbf{1}}$$

$$\text{T-Guard}\ \frac{\Gamma_1 \vdash V : ?F^{\bullet} \qquad \Gamma_2 \vdash \overrightarrow{G} : A :: F \qquad E \sqsubseteq F \qquad \vDash F}{\Gamma_1 + \Gamma_2 \vdash \mathbf{guard}\ V : E\ \{\overrightarrow{G}\} : A} \qquad \text{T-Sub}\ \frac{\Gamma \leq \Gamma' \qquad A \leq B \qquad \Gamma' \vdash M : A}{\Gamma \vdash M : B}$$

Typing rules for guards $\boxed{\Gamma \vdash_{\mathcal{P}} \overrightarrow{G} : A :: E}$ $\boxed{\Gamma \vdash_{\mathcal{P}} G : A :: E}$

$$\text{TG-GuardSeq}\ \frac{(\Gamma \vdash G_i : A :: E_i)_i}{\Gamma \vdash \overrightarrow{G} : A :: E_1 \oplus \ldots \oplus E_n} \qquad \text{TG-Fail}\ \frac{}{\Gamma \vdash \mathbf{fail} : A :: \mathbb{0}} \qquad \text{TG-Free}\ \frac{\Gamma \vdash M : A}{\Gamma \vdash \mathbf{free} \mapsto M : A :: \mathbb{1}}$$

$$\text{TG-Recv}\ \frac{\mathcal{P}(\mathtt{m}) = \overrightarrow{T} \qquad \mathrm{base}(\overrightarrow{T}) \vee \mathrm{base}(\Gamma) \qquad \Gamma, y : ?E^{\bullet}, \overrightarrow{x} : \overrightarrow{\lceil T \rceil} \vdash M : B}{\Gamma \vdash \mathbf{receive}\ \mathtt{m}[\overrightarrow{x}]\ \mathbf{from}\ y \mapsto M : B :: \mathtt{m} \odot E}$$

Pattern residual $\boxed{E \,/\, \mathtt{m}}$

$$\mathbb{0} \,/\, \mathtt{m} \triangleq \mathbb{0} \qquad \mathbb{1} \,/\, \mathtt{m} \triangleq \mathbb{0} \qquad \mathtt{m} \,/\, \mathtt{m} \triangleq \mathbb{1} \qquad \frac{\mathtt{m} \neq \mathtt{n}}{\mathtt{m} \,/\, \mathtt{n} \triangleq \mathbb{0}} \qquad (E \oplus F) \,/\, \mathtt{m} \triangleq (E \,/\, \mathtt{m}) \oplus (F \,/\, \mathtt{m})$$

$$(E \odot F) \,/\, \mathtt{m} \triangleq ((E \,/\, \mathtt{m}) \odot F) \oplus (E \odot (F \,/\, \mathtt{m}))$$

Pattern normal form (PNF) $\boxed{E \vDash_{\mathsf{lit}} F}$ $\boxed{E \vDash F}$

$$E \vDash_{\mathsf{lit}} \mathbb{0} \qquad E \vDash_{\mathsf{lit}} \mathbb{1} \qquad \frac{F \simeq E \,/\, \mathtt{m}}{E \vDash_{\mathsf{lit}} \mathtt{m} \odot F} \qquad \frac{E \vDash_{\mathsf{lit}} F_1 \qquad E \vDash_{\mathsf{lit}} F_2}{E \vDash F_1 \oplus F_2} \qquad \frac{E \vDash_{\mathsf{lit}} F}{E \vDash F}$$

Fig. 5. Pat declarative term typing

in concurrent processes. Rule T-New creates a fresh mailbox with type $?\mathbb{1}^{\bullet}$, since subsequent sends and receives must "balance out" to an empty mailbox.

Rule T-Send types a send expression $V\,!\,\mathtt{m}[\overrightarrow{W}]$, where a message $\mathtt{m}$ with payloads $\overrightarrow{W}$ is sent to a mailbox V. Value V must be a reference with type $!\mathtt{m}^{\circ}$, meaning that it can be used to send message $\mathtt{m}$. The mailbox only needs to be *second-class*, but subtyping means that we can also send to a first-class name. All payloads $\overrightarrow{V}$ must be subtypes of the types defined by the signature for message $\mathtt{m}$, and payloads must be typable under separate environments to avoid aliasing when receiving a message. Unlike in session-typed functional programming languages, sending is a side-effecting operation of type $\mathbf{1}$, and the behavioural typing is accounted for in environment composition.

Rule T-Guard types the expression $\mathbf{guard}\ V : E\ \{\overrightarrow{G}\}$, that retrieves from mailbox V with some pattern E using guards $\overrightarrow{G}$. The first premise ensures that under a type environment Γ_1, mailbox V has type $?F^{\bullet}$: the mailbox should have a receive capability with pattern F, and must be returnable.

Demanding that the mailbox is returnable rules out use-after-free errors since we cannot use the mailbox name in the continuation. The second premise states that under type environment Γ_2, guards $\overrightarrow{G}$ all return a value of type A and correspond to pattern F. The third premise requires that the pattern assertion E is contained within F. The final premise, $\vDash F$, ensures that F is in *pattern normal form*: the pattern should be a disjunction of pattern literals. That is $\mathbb{0}$, $\mathbb{1}$, or $\mathtt{m} \odot F$, where F is equivalent to E without message $\mathtt{m}$.

Finally, rule T-Sub allows the use of subtyping. Subtyping on type environments is crucial when constructing derivations, *e.g.* two patterns may have the same semantics but differ syntactically. Applying T-Sub makes it possible to rewrite mailbox types so that they can be combined by the type combination operators. We also allow the usual use of subsumption on return types, *e.g.* allowing a value with a subtype of a function argument to be used.

Guards. Rule TG-GuardSeq types a sequence of guards, ensuring that each guard is typable under the same type environment and with the same return type. Rule TG-Fail types a failure guard: since the type system will ensure that such a guard is never evaluated, it can have any type environment and any type, and is typable under pattern literal $\mathbb{0}$. Rule TG-Free types a guard of the form $\textbf{free} \mapsto M$, where M has type A. Finally, rule TG-Recv types a guard of the form $\textbf{receive}\ \mathtt{m}[\overrightarrow{x}]\ \textbf{from}\ y \mapsto M$, that retrieves a message with tag $\mathtt{m}$ from the mailbox, binding its payloads (whose types are retrieved from the signature for message $\mathtt{m}$) to $\overrightarrow{x}$, and re-binding the mailbox to y with an updated type in continuation M. The payloads are made *usable* rather than returnable, as otherwise the payloads could interfere with the names in the enclosing context.

Pattern residual. The *pattern residual* $E \,/\, \mathtt{m}$ calculates the pattern E after $\mathtt{m}$ is consumed, and corresponds to the Brzozowski derivative [Brzozowski 1964] over a commutative regular expression. The residual of $\mathbb{0}$, $\mathbb{1}$, or $\mathtt{n}$ (where $\mathtt{n} \neq \mathtt{m}$) with respect to a message tag $\mathtt{m}$ is the unreliable type $\mathbb{0}$. The derivative of $\mathtt{m}$ with respect to $\mathtt{m}$ is $\mathbb{1}$. The derivative operator distributes over $\oplus$, and the derivative of concatenation is the disjunction of the derivative of each subpattern.

Example. We end this section by showing the derivation for the client definition from the future example in §1, which creates a future and self mailbox, initialises the future with a number, and then requests and prints the result. In the following, we abbreviate *future* to f, *self* to s, and *result* to r. We assume that the program includes a signature $\mathcal{S} = [\mathtt{Put} \mapsto \mathsf{Int}, \mathtt{Get} \mapsto\ !\mathtt{Reply}, \mathtt{Reply} \mapsto \mathsf{Int}]$, and the emptyFuture and fullFuture definitions from §1. We split the derivation into three subderivations. Since it is easier to read derivations top-down, we start by typing the **guard** expression. In the following, we refer to the **receive** guard as G, and name the first derivation $\mathbf{D}_1$:

$$
\dfrac{
\dfrac{}{s : ?(\mathtt{Reply} \odot \mathbb{1})^{\bullet} \vdash s : ?(\mathtt{Reply} \odot \mathbb{1})^{\bullet}}
\qquad
\dfrac{
\dfrac{
\dfrac{\dfrac{}{s : ?\mathbb{1}^{\bullet} \vdash s : ?\mathbb{1}^{\bullet}}}{s : ?\mathbb{1}^{\bullet} \vdash \textbf{free}\ s : \mathbf{1}}
\qquad
r : \mathsf{Int} \vdash \mathsf{print}(\mathsf{intToString}(r)) : \mathbf{1}
}{s : ?\mathbb{1}^{\bullet}, r : \mathsf{Int} \vdash \textbf{free}\ s; \mathsf{print}(\mathsf{intToString}(r)) : \mathbf{1}}
}{\cdot \vdash \begin{array}{l}\textbf{receive}\ \mathtt{Reply}[r]\ \textbf{from}\ s \mapsto \\ \quad \textbf{free}\ s; \mathsf{print}(\mathsf{intToString}(r))\end{array} : \mathbf{1} :: \mathtt{Reply} \odot \mathbb{1}}
\qquad
\begin{array}{l}\mathtt{Reply} \sqsubseteq \mathtt{Reply} \odot \mathbb{1} \\ \vDash \mathtt{Reply} \odot \mathbb{1}\end{array}
}{s : ?(\mathtt{Reply} \odot \mathbb{1})^{\bullet} \vdash \textbf{guard}\ s : \mathtt{Reply}\ \{G\} : \mathbf{1}}
$$

The type of the s mailbox in the subject of the **guard** expression is $?(\mathtt{Reply} \odot \mathbb{1})^{\bullet}$ denoting that the mailbox can contain a $\mathtt{Reply}$ message and will then be empty. The **receive** guard binds s at type $?\mathbb{1}^{\bullet}$ and r at Int, freeing s and using r in the print expression. The $\mathtt{Reply}$ annotation on the guard is a subpattern of the pattern of s. The above derivation is used within derivation $\mathbf{D}_2$:

$$\dfrac{\dfrac{\overline{f\colon !\mathtt{Put}^\circ \vdash f\colon !\mathtt{Put}^\circ}}{f\colon !\mathtt{Put}^\circ \vdash f!\mathtt{Put}[5]:\mathbf{1}} \qquad \dfrac{\overline{\cdot \vdash 5:\mathsf{Int}}}{} \qquad \dfrac{\dfrac{\dfrac{\overline{f\colon !\mathtt{Get}^\circ \vdash f\colon !\mathtt{Get}^\circ}}{f\colon !\mathtt{Get}^\bullet \vdash f\colon !\mathtt{Get}^\circ} \qquad \overline{s\colon !\mathtt{Reply}^\circ \vdash s\colon !\mathtt{Reply}^\circ}}{f\colon !\mathtt{Get}^\bullet, s\colon !\mathtt{Reply}^\circ \vdash f!\mathtt{Get}[s]:\mathbf{1}}\ \mathbf{D}_1}{f\colon !\mathtt{Get}^\bullet, s\colon ?\mathbb{1}^\bullet \vdash \begin{array}{l} f!\mathtt{Get}[s]; \\ \textbf{guard}\ s\colon \mathtt{Reply}\ \{G\} \end{array} :\mathbf{1}}}{f\colon !(\mathtt{Put} \odot \mathtt{Get})^\bullet, s\colon ?\mathbb{1}^\bullet \vdash \begin{array}{l} f!\mathtt{Put}[5];\ f!\mathtt{Get}[s]; \\ \textbf{guard}\ s\colon \mathtt{Reply}\ \{\cdots\} \end{array} :\mathbf{1}}$$

Here f is used to send a $\mathtt{Put}$ and then a $\mathtt{Get}$ with s of type $!\mathtt{Reply}^\circ$ as payload. As the two sends to the f message are sequentially composed, the type of f at the root of the subderivation is $!(\mathtt{Put} \odot \mathtt{Get})^\bullet$. Since s is used at type $?(\mathtt{Reply} \odot \mathbb{1})^\bullet$ in $\mathbf{D}_1$, the send and receive patterns balance out to the empty mailbox type $?\mathbb{1}^\bullet$. Finally, we can construct the derivation for the entire term:

$$\dfrac{\overline{\cdot \vdash \textbf{new} : ?\mathbb{1}^\bullet} \qquad \dfrac{\dfrac{\dfrac{f\colon ?(\mathtt{Put} \odot \star\mathtt{Get})^\bullet \vdash \mathsf{emptyFuture}(f):\mathbf{1}}{f\colon ?(\mathtt{Put} \odot \star\mathtt{Get})^\circ \vdash \textbf{spawn}\ \mathsf{emptyFuture}(f):\mathbf{1}}}{f\colon ?((\mathtt{Put} \odot \mathtt{Get}) \odot \mathbb{1})^\circ \vdash \textbf{spawn}\ \mathsf{emptyFuture}(f):\mathbf{1}} \qquad \overline{\cdot \vdash \textbf{new} : ?\mathbb{1}^\bullet}\ \mathbf{D}_2}{f\colon !(\mathtt{Put} \odot \mathtt{Get})^\bullet \vdash \begin{array}{l} \textbf{let}\ s = \textbf{new in} \\ f!\mathtt{Put}[5];\cdots \end{array} :\mathbf{1}}}{f\colon ?\mathbb{1}^\bullet \vdash \begin{array}{l} \textbf{spawn}\ \mathsf{emptyFuture}(f); \\ \textbf{let}\ s = \textbf{new in} \\ f!\mathtt{Put}[5];\cdots \end{array} :\mathbf{1}}}{\cdot \vdash \begin{array}{l} \textbf{let}\ f = \textbf{new in} \\ \textbf{spawn}\ \mathsf{emptyFuture}(f); \\ \textbf{let}\ s = \textbf{new in} \\ f!\mathtt{Put}[5];\ f!\mathtt{Get}[s]; \\ \textbf{guard}\ s\colon \mathtt{Reply}\ \{ \\ \quad \textbf{receive}\ \mathtt{Reply}[r]\ \textbf{from}\ s \mapsto \\ \quad\quad \textbf{free}\ s; \\ \quad\quad \mathsf{print}(\mathsf{intToString}(r)) \\ \} \end{array} :\mathbf{1}}$$

Since we let-bind f to **new**, f must have type $?\mathbb{1}^\bullet$. Definition emptyFuture requires an argument of type $?(\mathtt{Put} \odot \star\mathtt{Get})^\bullet$; since the function application appears in the body of the **spawn** we can mask the usage annotation to $\circ$, and use environment subtyping to rewrite the type of f to $?((\mathtt{Put} \odot \mathtt{Get}) \odot \mathbb{1})^\bullet$. This then balances out with the use of f in $\mathbf{D}_2$, completing the derivation.

3.3 Operational Semantics

Figure 6 shows the runtime syntax and reduction rules for Pat. We extend values V with runtime names a. The concurrent semantics of the language is described as a nondeterministic reduction relation on a language of *configurations*, which resemble terms in the π-calculus. A thread $(\!| M, \Sigma |\!)$ evaluates term M with frame stack Σ (discussed shortly). Configuration $a \leftarrow \mathtt{m}[\overrightarrow{V}]$ is a message $\mathtt{m}[\overrightarrow{V}]$ in mailbox a; name restriction $(\nu a)\mathcal{C}$ binds name a in $\mathcal{C}$; and $\mathcal{C} \parallel \mathcal{D}$ denotes the parallel composition of $\mathcal{C}$ and $\mathcal{D}$. Structural congruence $\equiv$ (omitted) is standard, capturing scope extrusion and the associativity and commutativity of parallel composition. The semantics envisages a single static term M (*i.e.*, program text) to be evaluated in the context of an empty frame stack: $(\!| M, \epsilon |\!)$.

Frame stacks. We use frame stacks [Ennals et al. 2004; Pitts 1998] rather than evaluation contexts for technical convenience. A frame $\langle x, M\rangle$ is a pair of a variable x and a continuation M, where x is free in M. A frame stack is an ordered sequence of frames, where ϵ denotes the empty stack.

Reduction rules. Frame stacks are best demonstrated by the E-Let and E-Return rules: intuitively, **let** x: $T = M$ **in** N evaluates M, binding the result to x in N. The rule adds a fresh frame $\langle x, N\rangle$ to the top of a frame stack, and evaluates M. Conversely, E-Return returns V into the parent frame: if the top frame is $\langle x, M\rangle$, then we can evaluate the continuation M with V substituted for x. Rule E-App evaluates the body of function f with arguments $\overrightarrow{V}$ substituted for the parameters $\overrightarrow{x}$.

Runtime syntax

Runtime names	a			Guard contexts	$\mathcal{G}$	$::=$	$\overrightarrow{G_1} \cdot [\ ] \cdot \overrightarrow{G_2}$
Names	u, v, w	$::=$	$x \mid a$	Configurations	$\mathcal{C}, \mathcal{D}$	$::=$	$(\!\| M, \Sigma \|\!) \mid a \leftarrow \mathtt{m}[\overrightarrow{V}]$
Frames	σ	$::=$	$\langle x, M \rangle$			$\mid$	$\mathcal{C} \parallel \mathcal{D} \mid (\nu a)\mathcal{C}$
Frame stacks	Σ	$::=$	$\epsilon \mid \sigma \cdot \Sigma$	Runtime type environments	Δ	$::=$	$\cdot \mid \Delta, u : T$

Reduction rules $\boxed{\mathcal{C} \longrightarrow_{\mathcal{P}} \mathcal{D}}$

$$
\begin{array}{lrcl}
\textsc{E-Let} & (\!| \textbf{let}\ x{:}\ T = M\ \textbf{in}\ N, \Sigma |\!) & \longrightarrow & (\!| M, \langle x, N \rangle \cdot \Sigma |\!) \\
\textsc{E-Return} & (\!| V, \langle x, M \rangle \cdot \Sigma |\!) & \longrightarrow & (\!| M\{V/x\}, \Sigma |\!) \\
\textsc{E-App} & (\!| f(\overrightarrow{V}), \Sigma |\!) & \longrightarrow & (\!| M\{\overrightarrow{V}/\overrightarrow{x}\}, \Sigma |\!) \\
 & & & (\text{if } \mathcal{P}(f) = \textbf{def}\ f(\overrightarrow{x : A}) : B\ \{M\}) \\
\textsc{E-New} & (\!| \textbf{new}, \Sigma |\!) & \longrightarrow & (\nu a)((\!| a, \Sigma |\!)) \qquad (a \text{ is fresh}) \\
\textsc{E-Send} & (\!| a\,!\,\mathtt{m}[\overrightarrow{V}], \Sigma |\!) & \longrightarrow & (\!| (), \Sigma |\!) \parallel a \leftarrow \mathtt{m}[\overrightarrow{V}] \\
\textsc{E-Spawn} & (\!| \textbf{spawn}\ M, \Sigma |\!) & \longrightarrow & (\!| (), \Sigma |\!) \parallel (\!| M, \epsilon |\!) \\
\textsc{E-Free} & (\nu a)((\!| \textbf{guard}\ a : E\ \{\mathcal{G}[\textbf{free} \mapsto M]\}, \Sigma |\!)) & \longrightarrow & (\!| M, \Sigma |\!) \\
\textsc{E-Recv} & (\!| \textbf{guard}\ a : E\ \{\mathcal{G}[\textbf{receive}\ \mathtt{m}[\overrightarrow{x}]\ \textbf{from}\ y \mapsto M]\}, \Sigma |\!) \parallel a \leftarrow \mathtt{m}[\overrightarrow{V}] & \longrightarrow & (\!| M\{\overrightarrow{V}/\overrightarrow{x}, a/y\}, \Sigma |\!)
\end{array}
$$

$$
\textsc{E-Nu}\ \frac{\mathcal{C} \longrightarrow \mathcal{D}}{(\nu a)\mathcal{C} \longrightarrow (\nu a)\mathcal{D}} \qquad \textsc{E-Par}\ \frac{\mathcal{C} \longrightarrow \mathcal{C}'}{\mathcal{C} \parallel \mathcal{D} \longrightarrow \mathcal{C}' \parallel \mathcal{D}} \qquad \textsc{E-Struct}\ \frac{\mathcal{C} \equiv \mathcal{C}' \quad \mathcal{C}' \longrightarrow \mathcal{D}' \quad \mathcal{D}' \equiv \mathcal{D}}{\mathcal{C} \longrightarrow \mathcal{D}}
$$

Fig. 6. Pat operational semantics

Rule E-New creates a fresh mailbox name restriction and returns it into the calling context. Rule E-Send sends a message with tag $\mathtt{m}$ and payloads $\overrightarrow{V}$ to a mailbox a, returning () to the calling context and creating a sent message configuration $a \leftarrow \mathtt{m}[\overrightarrow{V}]$. Rule E-Spawn spawns a computation as a fresh process, with an empty frame stack. Rule E-Free allows a name a to be garbage collected if it is not contained in any other thread, evaluating the continuation M of the **free** guard. Finally, rule E-Recv handles receiving a message from a mailbox, binding the payload values to $\overrightarrow{x}$ and updated mailbox name to y in continuation M. The remaining rules are administrative.

3.4 Metatheory

3.4.1 Runtime Typing. To prove metatheoretical properties about Pat we introduce a type system on configurations; this type system is used only for reasoning and is not required for typechecking.

Runtime type environments. The runtime typing rules make use of a type environment Δ that maps variables to types that *do not* contain usage information. Usage information is inherently only useful in constraining *sequential* uses of a mailbox variable, where guards are blocking, whereas it makes little sense to constrain *concurrent* usages of a variable. Runtime type environment combination on $\Delta_1 \bowtie \Delta_2$ is similar to usage-annotated type environment combination but with two differences: it is *commutative* to account for the unordered nature of parallel threads, and type combination does not include usage information.

Definition 3.10 (Environment combination (Δ)). Environment combination $\Delta_1 \bowtie \Delta_2$ is the smallest partial commutative binary operator on type environments closed under the following rules:

$$
\frac{}{\cdot \bowtie \cdot = \cdot} \qquad \frac{\begin{array}{c} x \notin \mathsf{dom}(\Delta_2) \\ \Delta_1 \bowtie \Delta_2 = \Delta \end{array}}{(\Delta_1, x{:}T) \bowtie \Delta_2 = \Delta, x{:}T} \qquad \frac{\begin{array}{c} x \notin \mathsf{dom}(\Delta_1) \\ \Delta_1 \bowtie \Delta_2 = \Delta \end{array}}{\Delta_1 \bowtie (\Delta_2, x{:}T) = \Delta, x{:}T} \qquad \frac{\Delta_1 \bowtie \Delta_2 = \Delta}{(\Delta_1, x{:}T) \bowtie (\Delta_2, x{:}U) = \Delta, x{:}(T \boxplus U)}
$$

Disjoint combination on runtime type environments $\Delta_1 + \Delta_2$ (omitted) is defined analogously to disjoint combination on Γ.

Configuration Typing $\boxed{\Delta \vdash C}$

$$\text{TC-Nu}\quad \frac{\Delta, a : ?\mathbb{1} \vdash C}{\Delta \vdash (\nu a)C} \qquad \text{TC-Par}\quad \frac{\Delta_1 \vdash C \qquad \Delta_2 \vdash \mathcal{D}}{\Delta_1 \bowtie \Delta_2 \vdash C \parallel \mathcal{D}} \qquad \text{TC-Message}\quad \frac{(\lceil \Delta_i \rceil \vdash V_i : A_i)_{i \in 1..n} \qquad \overrightarrow{A} \leq \lceil \mathcal{P}(\mathtt{m}) \rceil}{\Delta_1 + \ldots + \Delta_n, a : !\mathtt{m} \vdash a \leftarrow \mathtt{m}[\overrightarrow{V}]}$$

$$\text{TC-Thread}\quad \frac{\lfloor \Delta \rfloor = \Gamma_1 \triangleright \Gamma_2 \qquad \Gamma_1 \vdash M : A \qquad \Gamma_2 \vdash A \blacktriangleright \Sigma}{\Delta \vdash (\!| M, \Sigma |\!)} \qquad \text{TC-Subs}\quad \frac{\Delta \leq \Delta' \qquad \Delta' \vdash C}{\Delta \vdash C}$$

Frame Stack Typing $\boxed{\Gamma \vdash A \blacktriangleright \Sigma}$

$$\frac{}{\cdot \vdash A \blacktriangleright \epsilon} \qquad \frac{\Gamma_1, x : A \vdash M : B \qquad \text{returnable}(B) \qquad \Gamma_2 \vdash B \blacktriangleright \Sigma}{\Gamma_1 \triangleright \Gamma_2 \vdash A \blacktriangleright \langle x, M \rangle \cdot \Sigma}$$

Fig. 7. Pat runtime typing

Runtime typing rules. Figure 7 shows the runtime typing rules. Rule TC-Nu types a name restriction if the name is of type $?\mathbb{1}$; in turn this ensures that sends and receives on the mailbox "balance out" across threads. Rule TC-Par allows configurations $\mathcal{C}$ and $\mathcal{D}$ to be composed in parallel if they are typable under combinable runtime type environments. Rule TC-Message types a message configuration $a \leftarrow \mathtt{m}[\overrightarrow{V}]$. Name a of type $!\mathtt{m}$ cannot appear in any of the values sent as a payload. Each payload value V must be a subtype of the type defined by the message signature, under the second-class lifting of a disjoint runtime type environment. Rule TC-Subs allows subtyping on runtime type environments; the subtyping relation $\Delta \leq \Delta'$ is analogous to subtyping on Γ.

Thread and frame stack typing. Rule TC-Thread types a thread, which is a pair of a currently-evaluating term, typable under an environment Γ_1, and a stack frame, typable under an environment Γ_2. The combination $\Gamma_1 \triangleright \Gamma_2$ should result in the returnable lifting of Δ: intuitively, we should be able to use every mailbox variable in Δ as returnable in the thread. TC-Thread makes use of the frame stack typing judgement $\Gamma \vdash A \blacktriangleright \Sigma$ (inspired by [Ennals et al. 2004]), which can be read "under type environment Γ, given a value of type A, frame stack Σ is well-typed". The empty frame stack is typable under the empty environment given any type. A non-empty frame stack $\langle x, M \rangle \cdot \Sigma$ is well-typed if M has some returnable type B, given a variable x of type A. The remainder of the stack must then be well-typed given B. We combine the environments used for typing the head term and the remainder of the stack using $\triangleright$ as we wish to account for sequential uses of a mailbox; for example, in the term $x\,!\,\mathtt{m}[V]; x\,!\,\mathtt{n}[W]$, x would have type $!(\mathtt{m} \odot \mathtt{n})^{\circ}$.

3.4.2 Properties. We can now state some metatheoretical results. Proofs can be found in the extended version [Fowler et al. 2023b]. Typability is preserved by reduction; the proof is nontrivial since we must do extensive reasoning about environment combination.

Theorem 3.11 (Preservation). *If* $\vdash \mathcal{P}$, *and* $\Gamma \vdash_{\mathcal{P}} \mathcal{C}$ *with* Γ *reliable, and* $\mathcal{C} \longrightarrow_{\mathcal{P}} \mathcal{D}$, *then* $\Gamma \vdash_{\mathcal{P}} \mathcal{D}$.

Preservation implies mailbox conformance: the property that a configuration will never evaluate to a singleton failure guard. To state mailbox conformance, it is useful to define the notion of a *configuration context* $\mathcal{H} ::= (\nu a)\mathcal{H} \mid \mathcal{H} \parallel \mathcal{C} \mid (\!| [\,], \Sigma |\!)$, that allows us to focus on a single thread.

Corollary 3.12 (Mailbox Conformance).
If $\vdash \mathcal{P}$ *and* $\Gamma \vdash_{\mathcal{P}} \mathcal{C}$ *with* Γ *reliable, then* $\mathcal{C} \not\longrightarrow^{*} \mathcal{H}[\textbf{fail}\ V]$.

Progress. To prove a progress result for Pat, we begin with some auxiliary definitions.

Definition 3.13 (Message set). A *message set* $\mathcal{M}$ is a configuration of the form: $a_1 \leftarrow \mathtt{m}_1[\overrightarrow{V_1}] \parallel \cdots \parallel a_n \leftarrow \mathtt{m}_n[\overrightarrow{V_n}]$. We say that a message set $\mathcal{M}$ *contains a message* $\mathtt{m}$ *for* a if $\mathcal{M} \equiv a \leftarrow \mathtt{m}[\overrightarrow{V}] \parallel \mathcal{M}'$.

Pattern variables	α, β		
Mailbox patterns	γ, δ	::=	$\mathbb{0} \mid \mathbb{1} \mid \mathtt{m} \mid \gamma \oplus \delta$
		$\mid$	$\gamma \odot \delta \mid \star\gamma \mid \alpha$
Mailbox types	ς	::=	$!\gamma \mid ?\gamma$
Types	π, ρ	::=	$C \mid \varsigma$
Usage-ann. types	τ, σ	::=	$C \mid \varsigma^{\eta}$

Augmented type envs.	Θ	::=	$\cdot \mid \Theta, x : \tau$
Nullable type envs.	Ψ	::=	$\Theta \mid \top$
Augmented definitions	D	::=	$\mathbf{def}\ f(\overrightarrow{x : \tau}) : \sigma\ \{M\}$
Constraints	ϕ	::=	$\gamma <: \delta$
Constraint sets	Φ		

Fig. 8. Pat syntax extended for algorithmic typing

Next, we classify *canonical forms*, which give us a global view of a configuration. Every well typed process is structurally congruent to a canonical form.

Definition 3.14 (Canonical form). A configuration C is in *canonical form* if it is of the form: $(\nu a_1) \cdots (\nu a_l)(\llparenthesis M_1, \Sigma_1 \rrparenthesis \parallel \cdots \parallel \llparenthesis M_m, \Sigma_m \rrparenthesis \parallel \mathcal{M})$

Definition 3.15 (Waiting). We say that a term M is *waiting on mailbox a for a message with tag* $\mathtt{m}$, written $\mathsf{waiting}(M, a, \mathtt{m})$, if M can be written $\mathbf{guard}\ a : E\ \{\mathcal{G}[\mathbf{receive}\ \mathtt{m}[x]\ \mathbf{from}\ y \mapsto N]\}$.

Let $\mathsf{fv}(-)$ denote the set of free variables in a term M or frame stack Σ. We can then use canonical forms to characterise a progress result: either each thread can reduce, has reduced to a value, or is waiting for a message which has not yet been sent by a different thread.

Theorem 3.16 (Partial Progress). *Suppose $\vdash \mathcal{P}$ and $\cdot \vdash_{\mathcal{P}} C$ where C is in canonical form:*

$$C = (\nu a_1) \cdots (\nu a_l)(\llparenthesis M_1, \Sigma_1 \rrparenthesis \parallel \cdots \llparenthesis M_m, \Sigma_m \rrparenthesis \parallel \mathcal{M})$$

Then for each M_i, either:

- *there exist M_i', Σ_i' such that $\llparenthesis M_i, \Sigma_i \rrparenthesis \longrightarrow \llparenthesis M_i', \Sigma_i' \rrparenthesis$; or*
- *M_i is a value and $\Sigma_i = \epsilon$; or*
- *$\mathit{waiting}(M_i, a_j, \mathtt{m}_j)$ where $\mathcal{M}$ does not contain a message $\mathtt{m}_j$ for a_j and $a_j \notin \mathit{fv}(\overrightarrow{G_i}) \cup \mathit{fv}(\Sigma_i)$, where $\overrightarrow{G_i}$ are the guard clauses of M_i.*

A key consequence of Theorem 3.16 is the absence of self-deadlocks: since we can only guard on a returnable mailbox, and a returnable name must be the last occurrence in the thread, it cannot be that the **guard** expression is blocking a send to the same mailbox in the same thread. As we cannot use dependency graphs (§2.2), our progress result does not rule out inter-process deadlocks.

4 ALGORITHMIC TYPING

Writing a typechecker based on Pat's declarative typing rules is challenging due to nondeterministic context splits, environment subtyping, and pattern inclusion. MC^2 [Padovani 2018b] is a typechecker for the mailbox calculus, based on a typechecker for concurrent object usage protocols [Padovani 2018c]. The MC^2 type system has, however, not been formalised. We adopt several ideas from MC^2, especially algorithmic type combination, and adapt the approach for a programming language.

Type system overview. Our algorithmic type system takes a *co-contextual* [Erdweg et al. 2015] approach: rather than taking a type environment as an *input* to the type-checking algorithm, we produce a type environment as an *output*. The intuition is that (read bottom-up), *splitting* an environment into two sub-environments is more difficult than *merging* two environments inferred from subexpressions. We also generate *inclusion constraints* on patterns to be solved later.

Bidirectional type systems [Dunfield and Krishnaswami 2022; Pierce and Turner 2000] split typing rules into two classes: those that *synthesise* a type A for a term M ($\Gamma \vdash M \Rightarrow A$), and those

Unrestrictedness $\boxed{\mathsf{unr}(\tau) \blacktriangleright \Phi}$

$$\frac{}{\mathsf{unr}(C) \blacktriangleright \emptyset} \qquad \frac{}{\mathsf{unr}(!\gamma^{\circ}) \blacktriangleright \{\mathbb{1} <: \gamma\}}$$

Subtyping $\boxed{\eta_1 \leq \eta_2}$ $\boxed{\tau \leq \sigma \blacktriangleright \Phi}$

$$\frac{}{\eta \leq \eta} \qquad \frac{}{\bullet \leq \circ} \qquad \frac{}{C \leq C \blacktriangleright \emptyset} \qquad \frac{\eta_1 \leq \eta_2}{!\gamma^{\eta_1} \leq !\delta^{\eta_2} \blacktriangleright \delta <: \gamma} \qquad \frac{\eta_1 \leq \eta_2}{?\gamma^{\eta_1} \leq ?\delta^{\eta_2} \blacktriangleright \gamma <: \delta}$$

Type join $\boxed{\varsigma_1 \fatsemi \varsigma_2 \blacktriangleright \varsigma; \Phi}$ $\boxed{\tau_1 \fatsemi \tau_2 \blacktriangleright \sigma; \Phi}$

$$\frac{}{!\gamma \fatsemi !\delta \blacktriangleright !(\gamma \odot \delta); \emptyset} \qquad \frac{\alpha \text{ fresh}}{!\gamma \fatsemi ?\delta \blacktriangleright ?\alpha; \{(\gamma \odot \alpha) <: \delta\}} \qquad \frac{\alpha \text{ fresh}}{?\gamma \fatsemi !\delta \blacktriangleright ?\alpha; \{(\delta \odot \alpha) <: \gamma\}}$$

$$\frac{\varsigma_1 \fatsemi \varsigma_2 \blacktriangleright \varsigma; \Phi}{\varsigma_1^{\eta_1} \fatsemi \varsigma_2^{\eta_2} \blacktriangleright \varsigma^{\eta_1 \triangleright \eta_2}; \Phi} \qquad \frac{}{C \fatsemi C \blacktriangleright C; \emptyset}$$

Type merge $\boxed{\varsigma_1 \sqcap \varsigma_2 \blacktriangleright \varsigma; \Phi}$ $\boxed{\tau_1 \sqcap \tau_2 \blacktriangleright \sigma; \Phi}$

$$\frac{}{!\gamma \sqcap !\delta \blacktriangleright !(\gamma \oplus \delta); \emptyset} \qquad \frac{\alpha \text{ fresh}}{?\gamma \sqcap ?\delta \blacktriangleright ?\alpha; \{\alpha <: \gamma, \alpha <: \delta\}} \qquad \frac{\varsigma_1 \sqcap \varsigma_2 \blacktriangleright \varsigma; \Phi}{\varsigma_1^{\eta_1} \sqcap \varsigma_2^{\eta_2} \blacktriangleright \varsigma^{\max(\eta_1, \eta_2)}; \Phi} \qquad \frac{}{C \sqcap C \blacktriangleright C; \emptyset}$$

Fig. 9. Pat algorithmic type operations

that *check* that a term M has type A ($\Gamma \vdash M \Leftarrow A$). Bidirectional type systems are syntax-directed and amenable to implementation.

We use a co-contextual variant of bidirectional typing first introduced by Zeilberger [2015]. The main twist is the variable rule, which becomes a *checking* rule and records the given variable-type mapping in the inferred environment.

4.1 Algorithmic Type System

Extended syntax and annotation. A key difference in comparison to the declarative type system is the addition of *pattern variables* α, that act as a placeholder for part of a pattern and are generated during typechecking. We can then generate and solve *inclusion constraints* ϕ on patterns. Figure 8 shows the extended syntax used in the algorithmic system.

Constraints. An important challenge for the algorithmic type system is determining whether one pattern is *included* within another: *e.g.* $\mathtt{m} \sqsubseteq \star\mathtt{m}$. Given that patterns may contain pattern variables, we may need to defer inclusion checking until more pattern variables are known, so we introduce inclusion constraints $\gamma <: \delta$ which require that pattern γ is included in pattern δ.

Fig. 9 shows the algorithmic type combination operators.

Unrestrictedness and subtyping. The algorithmic unrestrictedness operation $\mathsf{unr}(\tau) \blacktriangleright \Phi$ states that τ is unrestricted subject to constraints Φ, and the definition reflects the fact that a type is unrestricted in the declarative system if it is a base type or a subtype of $!\mathbb{1}^{\circ}$. Algorithmic subtyping is similar: a base type is a subtype of itself, and we check that two mailbox types with the same capability are subtypes of each other by generating a contravariant constraint for a send type, and a covariant constraint for a receive type.

Algorithmic type join. Declarative mailbox typing relies on the subtyping rule to manipulate types into a form where they can be combined with the type combination operators, *e.g.*, $!E \boxplus ?(E \odot F) = ?F$. The algorithmic type system cannot apply the same technique as it does not know, *a priori*, the form of each pattern. Instead, the algorithmic *type join* operation allows the combination of

two mailbox types irrespective of their syntactic form. Combining two send types is the same as in the declarative system, but combining a send type with a receive type (and vice versa) is more interesting: say we wish to combine $!\gamma$ and $?\delta$. In this case, we generate a fresh pattern variable α; the result is $?\alpha$ along with the constraint that $(\gamma \odot \alpha) <: \delta$: namely, that the send pattern concatenated with the fresh pattern variable is included in the pattern δ.

As an example, joining $!\mathtt{m}$ and $?(\mathtt{n} \odot \mathtt{m})$ produces a receive mailbox type $?\alpha$ and a constraint $(\mathtt{m} \odot \alpha) <: (\mathtt{n} \odot \mathtt{m})$, for which a valid solution is $\alpha \mapsto \mathtt{n}$, and hence the expected combined type $?\mathtt{n}$.

Algorithmic type merge. In the declarative type system branching control flow requires that each branch is typable under the same type environment (using the T-Subs rule). The algorithmic type system instead generates constraints that ensure that each type is used consistently across branches using the *algorithmic type merge* operation $\tau_1 \sqcap \tau_2 \blacktriangleright \sigma; \Phi$. Two base types are merged if they are identical. In the case of mailbox types, the function takes the maximum usage annotation, so $\max(\bullet, \circ) = \bullet$. It ensures that when merging two output capabilities the patterns are combined using pattern disjunction. Conversely merging two input capabilities generates a new pattern variable that must be included in both merged patterns.

Algorithmic environment combination. We can extend the algorithmic type operations to type environments; the (omitted) rules are adaptations of the corresponding declarative operations. Notably, when combining two environments used in branching control flow such as conditionals where an output mailbox $!\gamma$ is used in one environment but not the other, the resulting type is $!(\gamma \oplus \mathbb{1})$ to signify the *choice* of not sending on the mailbox name.

Nullable type environments. Checking a **fail** guard produces a *null* environment $\top$ which can be composed with *any other* type environment, as shown by the following definition:

Definition 4.1 (Nullable environment combination). For each combination operator $\star \in \{\fatsemi, \sqcap, +\}$ we extend environment combination to nullable type environments, $\Psi_1 \star \Psi_2 \blacktriangleright \Psi; \Phi$ by extending each environment combination operation with the following rules:

$$\frac{}{\top \star \top \blacktriangleright \top; \emptyset} \qquad \frac{}{\top \star \Theta \blacktriangleright \Theta; \emptyset} \qquad \frac{}{\Theta \star \top \blacktriangleright \Theta; \emptyset}$$

Nullable type environments are a supertype of every defined type environment: $\Theta \leq \top$.

Figure 10 shows the Pat algorithmic typing of programs, definitions and terms. The key idea is to remain in checking mode for as long as possible, in order to propagate type information to the variable rule and construct a type environment. We write $\Theta - \overrightarrow{x}$ for $\{y : \tau \mid y : \tau \in \Theta \wedge y \notin \overrightarrow{x}\}$.

Synthesis. Our synthesis judgement has the form $M \Rightarrow_{\mathcal{P}} \tau \blacktriangleright \Theta; \Phi$, which can be read "synthesise type τ for term M under program $\mathcal{P}$, producing type environment Θ and constraints Φ". Here, M and $\mathcal{P}$ are inputs of the judgement, whereas τ, Θ, and Φ are outputs. The checking judgement $M \Leftarrow_{\mathcal{P}} \tau \blacktriangleright \Theta; \Phi$ can be read "check that term M has type τ under program $\mathcal{P}$, producing type environment Θ and constraints Φ". Here, M, $\mathcal{P}$, and τ are inputs of the judgement, whereas Θ and Φ are outputs. As in the declarative system we omit the $\mathcal{P}$ annotation in the rules for readability.

Rule TS-Const assigns a known base type to a constant, and rule TS-New synthesises a type $?\mathbb{1}^{\bullet}$ (analogous to T-New); both rules produce an empty environment and constraint set.

Rule TS-Spawn checks that the given computation M has the unit type, synthesises type $\mathbf{1}$, and infers a type environment Θ and constraint set Φ. Like T-Spawn in the declarative system, the usability annotations are masked as usable since usability restrictions are process-local.

Message sending $V\,!\mathtt{m}[\overrightarrow{W}]$ is a side-effecting operation, and so we synthesise type $\mathbf{1}$. Rule TS-Send first looks up the payload types $\overrightarrow{\pi}$ in the signature, and *checks* that message target V has mailbox type $!\mathtt{m}^{\circ}$. In performing this check, the type system will produce environment Θ' that

Constraint generation for programs and definitions $\boxed{\vdash \mathcal{P} \triangleright \Phi}$ $\boxed{\vdash_{\mathcal{P}} \textbf{def}\ f(\overrightarrow{x:\tau}) : \sigma\ \{M\} \triangleright \Phi}$

$$\frac{\mathcal{P} = (\mathcal{S}, \overrightarrow{D}, M) \qquad (\vdash_{\mathcal{P}} D_i \triangleright \Phi_i)_i \qquad M \Leftarrow \mathbf{1} \blacktriangleright \cdot; \Phi}{\vdash \mathcal{P} \triangleright \Phi \cup \Phi_1 \cup \cdots \cup \Phi_n}$$

$$\frac{M \Leftarrow \sigma \blacktriangleright \Theta; \Phi_1 \qquad \mathsf{check}(\overrightarrow{x}, \overrightarrow{\tau}, \Theta) = \Phi_2 \qquad \Theta - \overrightarrow{x} = \cdot}{\vdash_{\mathcal{P}} \textbf{def}\ f(\overrightarrow{x:\tau}) : \sigma\ \{M\} \triangleright \Phi_1 \cup \Phi_2}$$

Constraint generation (synthesis) $\boxed{M \Rightarrow_{\mathcal{P}} \tau \blacktriangleright \Theta; \Phi}$

$$\textsc{TS-Const}\ \frac{c \text{ has base type } C}{c \Rightarrow C \blacktriangleright \cdot; \emptyset} \qquad \textsc{TS-New}\ \frac{}{\textbf{new} \Rightarrow ?\mathbb{1}^{\bullet} \blacktriangleright \cdot; \emptyset} \qquad \textsc{TS-Spawn}\ \frac{M \Leftarrow \mathbf{1} \blacktriangleright \Theta; \Phi}{\textbf{spawn}\ M \Rightarrow \mathbf{1} \blacktriangleright \lceil \Theta \rceil; \Phi}$$

$$\textsc{TS-Send}\ \frac{\mathcal{P}(\mathtt{m}) = \overrightarrow{\pi} \qquad V \Leftarrow !\mathtt{m}^{\circ} \blacktriangleright \Theta'; \Phi \qquad (W_i \Leftarrow \lceil \pi_i \rceil \blacktriangleright \Theta'_i; \Phi'_i)_{i \in 1..n} \qquad \Theta' + \Theta'_1 + \ldots + \Theta'_n \blacktriangleright \Theta; \Phi''}{V\,!\mathtt{m}[\overrightarrow{W}] \Rightarrow \mathbf{1} \blacktriangleright \Theta; \Phi \cup \Phi'_1 \cup \ldots \cup \Phi'_n \cup \Phi''}$$

$$\textsc{TS-App}\ \frac{\mathcal{P}(f) = \overrightarrow{\tau} \to \sigma \qquad (V_i \Leftarrow \tau_i \blacktriangleright \Theta_i; \Phi_i)_{i \in 1..n} \qquad \Theta_1 + \ldots + \Theta_n \blacktriangleright \Theta; \Phi}{f(V_1, \ldots, V_n) \Rightarrow \sigma \blacktriangleright \Theta; \Phi \cup \Phi_1 \cup \ldots \cup \Phi_n}$$

Constraint generation (checking) $\boxed{M \Leftarrow_{\mathcal{P}} \tau \blacktriangleright \Theta; \Phi}$

$$\textsc{TC-Var}\ \frac{}{x \Leftarrow \tau \blacktriangleright x : \tau; \emptyset} \qquad \textsc{TC-Let}\ \frac{M \Leftarrow \lfloor T \rfloor \blacktriangleright \Theta_1; \Phi_1 \qquad N \Leftarrow \tau \blacktriangleright \Theta_2; \Phi_2 \qquad \mathsf{check}(\Theta_2, x, \lfloor T \rfloor) = \Phi_3 \qquad \Theta_1 - x \fatsemi \Theta_2 \blacktriangleright \Theta; \Phi_4}{\textbf{let}\ x{:}\ T = M\ \textbf{in}\ N \Leftarrow \tau \blacktriangleright \Theta; \Phi_1 \cup \cdots \cup \Phi_4}$$

$$\textsc{TC-Guard}\ \frac{\{E\}\ \overrightarrow{G} \Leftarrow \tau \blacktriangleright \Psi; \Phi_1; F \qquad V \Leftarrow ?F^{\bullet} \blacktriangleright \Theta'; \Phi_2 \qquad \Psi + \Theta' \blacktriangleright \Theta; \Phi_3}{\textbf{guard}\ V : E\ \{\overrightarrow{G}\} \Leftarrow \tau \blacktriangleright \Theta; \Phi_1 \cup \Phi_2 \cup \Phi_3 \cup \{E <: F\}}$$

$$\textsc{TC-Sub}\ \frac{M \Rightarrow \tau \blacktriangleright \Theta; \Phi_1 \qquad \tau \leq \sigma \blacktriangleright \Phi_2}{M \Leftarrow \sigma \blacktriangleright \Theta; \Phi_1 \cup \Phi_2}$$

Environment lookup $\boxed{\mathsf{check}(\Theta, x, \tau) = \Phi}$ $\boxed{\mathsf{check}(\Theta, \overrightarrow{x}, \overrightarrow{\tau}) = \Phi}$

$$\frac{x \notin \mathsf{dom}(\Theta) \qquad \mathsf{unr}(\tau) \blacktriangleright \Phi}{\mathsf{check}(\Theta, x, \tau) = \Phi} \qquad \frac{\sigma \leq \tau \blacktriangleright \Phi}{\mathsf{check}((\Theta, x : \tau), x, \sigma) = \Phi} \qquad \frac{(\mathsf{check}(\Theta, x_i, \tau_i) = \Phi_i)_i}{\mathsf{check}(\Theta, \overrightarrow{x}, \overrightarrow{\tau}) = \Phi_1 \cup \cdots \cup \Phi_n}$$

Fig. 10. Pat algorithmic typing (programs, definitions, and terms)

contains an entry mapping the variable in V to the desired mailbox type $!\mathtt{m}^{\circ}$. Next, the algorithm checks each payload value against the payload type described by the signature. The resulting environment is the algorithmic disjoint combination of the environments produced by checking each payload, and the resulting constraint set is the union of all generated constraints.

Function application is similar: rule TS-App looks up the type signature for function f and checks that all arguments have the expected types. The resulting environment is again the disjoint combination of the environments, and the constraint set is the union of all generated constraints.

Checking. Rule TC-Var *checks* that a variable x has type τ, producing a type environment $x : \tau$. The TC-Let rule checks that a let-binding **let** x: $T = M$ **in** N has type τ: first, we check that M has type $\lfloor T \rfloor$ noting that only values of returnable type may be returned, producing environment Θ_1 and constraints Φ_1. Next we check that the body N has type τ, producing environment Θ_2 and Φ_2. The next step is to check whether the types of the variable inferred in Θ_2 corresponds with the annotation. The check meta-function ensures that if x is not contained within Θ_2, then the type of x is unrestricted; and conversely if x *is* contained within Θ_2, then the annotation is a subtype of the inferred type as the annotation is a *lower bound* on what the body can expect of x.

Rule TC-Guard checks that a guard expression **guard** $V : E\ \{\overrightarrow{G}\}$ has return type τ. First, the rule checks that the guard sequence $\overrightarrow{G}$ has type τ, producing nullable environment Ψ, constraint

Constraint generation for guards $\{E\}\ \overrightarrow{G} \Leftarrow_{\mathcal{P}} \tau \blacktriangleright \Psi;\ \Phi;\ F$ | $\{E\}\ G \Leftarrow_{\mathcal{P}} \tau \blacktriangleright \Psi;\ \Phi;\ F$

TCG-Guards
$$\frac{(\{E\}\ G_i \Leftarrow \tau \blacktriangleright \Psi_i;\ \Phi_i;\ F_i)_{i \in 1..n} \qquad F = F_1 \oplus \cdots \oplus F_n \qquad \Psi_1 \sqcap \ldots \sqcap \Psi_n \blacktriangleright \Psi; \Phi}{\{E\}\ \overrightarrow{G} \Leftarrow \tau \blacktriangleright \Psi;\ \Phi \cup \Phi_1 \cup \cdots \cup \Phi_n;\ F}$$

TCG-Fail
$$\frac{}{\{E\}\ \textbf{fail} \Leftarrow \tau \blacktriangleright \top;\ \emptyset;\ \mathbb{0}}$$

TCG-Free
$$\frac{M \Leftarrow \tau \blacktriangleright \Theta;\ \Phi}{\{E\}\ \textbf{free} \mapsto M \Leftarrow \tau \blacktriangleright \Theta;\ \Phi;\ \mathbb{1}}$$

TCG-Recv
$$\frac{M \Leftarrow \tau \blacktriangleright \Theta', y : ?\gamma^{\bullet};\ \Phi_1 \qquad \mathcal{P}(\mathtt{m}) = \overrightarrow{\pi} \qquad \Theta = \Theta' - \overrightarrow{x} \qquad \mathsf{base}(\overrightarrow{\pi}) \vee \mathsf{base}(\Theta) \qquad \mathsf{check}(\Theta', \overrightarrow{x}, \overrightarrow{\lceil \pi \rceil}) = \Phi_2}{\{E\}\ \textbf{receive}\ \mathtt{m}[\overrightarrow{x}]\ \textbf{from}\ y \mapsto M \Leftarrow \tau \blacktriangleright \Theta;\ \Phi_1 \cup \Phi_2 \cup \{E \,/\, \mathtt{m} <: \gamma\};\ \mathtt{m} \odot (E \,/\, \mathtt{m})}$$

Fig. 11. Pat algorithmic typing (guards)

set Φ_1, and pattern F in pattern normal form. Next, the rule checks that the mailbox name V has type $?F^{\bullet}$, producing environment Θ' and constraint set Φ_2. Finally, the rule calculates the disjoint combination of Ψ and Θ', producing final environment Θ and constraints Φ_3.

Finally, rule TC-Sub states that if a term M is synthesisable with type τ, where τ is a subtype of σ, then M is checkable with type σ. The resulting environment is that produced by synthesising the type for M, and the resulting constraint set is the union of the synthesis and subtyping constraints.

*Un-annotated **let** expressions.* Although our core calculus assumes an annotation on **let** expressions, this is unnecessary if the let-bound variable is used in the continuation N, or M has a synthesisable type. Specifically, TC-LetNoAnn1 allows us to check the type of the continuation and inspect the produced environment for the type of x, which can be used to check M. Similarly, TC-LetNoAnn2 allows us to type a **let**-binding where x is *not* used in the continuation, as long as the type of M is synthesisable and unrestricted.

TC-LetNoAnn1
$$\frac{N \Leftarrow \sigma \blacktriangleright \Theta_1, x : \tau;\ \Phi_1 \qquad \mathsf{returnable}(\tau) \qquad M \Leftarrow \tau \blacktriangleright \Theta_2;\ \Phi_2 \qquad \Theta_2 \fatsemi \Theta_1 \blacktriangleright \Theta; \Phi_3}{\textbf{let}\ x = M\ \textbf{in}\ N \Leftarrow \sigma \blacktriangleright \Theta;\ \Phi_1 \cup \Phi_2 \cup \Phi_3}$$

TC-LetNoAnn2
$$\frac{N \Leftarrow \sigma \blacktriangleright \Theta_1;\ \Phi_1 \qquad x \notin \mathsf{dom}(\Theta_1) \qquad M \Rightarrow \tau \blacktriangleright \Theta_2;\ \Phi_2 \qquad \mathsf{returnable}(\tau) \qquad \Theta_2 \fatsemi \Theta_1 \blacktriangleright \Theta; \Phi_3}{\textbf{let}\ x = M\ \textbf{in}\ N \Leftarrow \sigma \blacktriangleright \Theta;\ \Phi_1 \cup \Phi_2 \cup \Phi_3}$$

We use the explicitly-typed representation in the core calculus for simplicity and uniformity, however the implementation follows the above approach to avoid needless annotations.

Guards. Figure 11 shows the typing rules for guards; the judgement $\{E\}\ G \Leftarrow \tau \blacktriangleright \Psi;\ \Phi;\ F$ can be read "Check that guard G has type τ, producing environment Ψ, constraints Φ, and closed pattern literal F in pattern normal form with respect to E". Rule TCG-Guards types a guard sequence, producing the algorithmic merge of all environments and the sum of all produced patterns. Rule TCG-Fail types the **fail** guard with any type and produces a null type environment, empty constraint set, and pattern $\mathbb{0}$. Rule TCG-Free checks that guard **free** $\mapsto M$ has type τ by checking that M has type τ; the guard produces pattern $\mathbb{1}$.

Finally, rule TCG-Recv checks that a receive guard **receive** $\mathtt{m}[\overrightarrow{x}]$ **from** $y \mapsto M$ has type τ. First, the rule checks that M has type τ, producing environment $\Theta', y : ?\gamma^{\bullet}$ and constraint set Φ_1; since a mailbox type with input capability is linear, it *must* be present in the inferred environment. Next, the rule checks that the inferred types for $\overrightarrow{x}$ in Θ' are compatible with the payloads for $\mathtt{m}$ declared in the signature, producing constraint set Φ_2. As with the declarative rule, to rule out unsafe aliasing either the payloads or inferred environment must consist only of base types. The resulting environment is Θ (i.e., the inferred environment without the mailbox variable or any payloads). The resulting constraint set is the union of Φ_1 and Φ_2 along with an additional constraint which ensures that $E \,/\, \mathtt{m}$ is included in γ, allowing us to produce the closed PNF literal $\mathtt{m} \odot (E \,/\, \mathtt{m})$.

4.2 Metatheory

We can now establish that the algorithmic type system is sound and complete with respect to the declarative type system. We begin by introducing the notion of pattern substitutions and solutions.

A *pattern substitution* Ξ is a mapping from type variables α to (fully-defined) patterns E; applying Ξ to a pattern γ substitutes all occurrences of a type variable α for $\Xi(\alpha)$. We extend application of pattern substitutions to types and environments. We write $\mathsf{pv}(E)$ for the set of pattern variables in a pattern and extend it to types and environments.

Definition 4.2 (Pattern solution). A pattern substitution Ξ is a *pattern solution* for a constraint set Φ (or *solves* Φ) if $\mathsf{pv}(\Phi) \subseteq \mathsf{dom}(\Xi)$ and for each $\gamma <: \delta \in \Xi$, we have that $\Xi(\gamma) \sqsubseteq \Xi(\delta)$. A solution Ξ is a *usable* solution if its range does not contain any pattern equivalent to $\mathbb{0}$.

4.2.1 Algorithmic Soundness.

Definition 4.3 (Covering solution). We say that a pattern substitution Ξ is a *covering solution* for a derivation $M \Rightarrow_{\mathcal{P}} \tau \blacktriangleright \Theta; \Phi$ or $M \Leftarrow_{\mathcal{P}} \tau \blacktriangleright \Theta; \Phi$ if given $\vdash \mathcal{P} \triangleright \Phi'$, it is the case that Ξ is a usable solution for $\Phi \cup \Phi'$ such that $\mathsf{pv}(\tau) \cup \mathsf{pv}(\mathcal{P}) \subseteq \mathsf{dom}(\Xi)$.

If a term is well typed in the algorithmic system then, given a covering solution, the term is also well typed in the declarative system.

Theorem 4.4 (Algorithmic Soundness).

- *If Ξ is a covering solution for $M \Rightarrow_{\mathcal{P}} \tau \blacktriangleright \Theta; \Phi$, then $\Xi(\Theta) \vdash_{\Xi(\mathcal{P})} M : \Xi(\tau)$.*
- *If Ξ is a covering solution for $M \Leftarrow_{\mathcal{P}} \tau \blacktriangleright \Theta; \Phi$, then $\Xi(\Theta) \vdash_{\Xi(\mathcal{P})} M : \Xi(\tau)$.*

4.2.2 Algorithmic Completeness.

We also obtain a completeness result, but only for the checking direction. This is because the type system *requires* type information to construct a type environment. In practice the lack of a completeness result for synthesis is unproblematic since all functions have return type annotations, and therefore the only terms typable in the declarative system but unsynthesisable are top-level terms containing free variables. In the following we assume that program $\mathcal{P}$ is *closed*, i.e. no definitions or message payloads contain type variables.

Theorem 4.5 (Algorithmic Completeness). *If $\vdash \mathcal{P}$ where $\mathcal{P}$ is closed, and $\Gamma \vdash_{\mathcal{P}} M : A$, then there exist some Θ, Φ and usable solution Ξ of Φ such that $M \Leftarrow_{\mathcal{P}} A \blacktriangleright \Theta; \Phi$ where $\Gamma \leq \Xi(\Theta)$.*

An unannotated **let** binding **let** $x = M$ **in** N is also typable by the algorithmic type system if either x occurs free in N, or the type of M is synthesisable; in practice this encompasses both base types and linear usages of mailbox types, *i.e.* the vast majority of use cases.

4.3 Constraint Solving

Constraint solving is covered in depth by Padovani [2018c], so we provide an informal overview:

Identify and group bounds A *pattern bound* is of the form $\gamma <: \alpha$ *i.e.* a constraint whose right-hand-side is a pattern variable. We firstly group all pattern bounds using pattern disjunction, *e.g.* a constraint set $\{\gamma <: \alpha, \delta <: \alpha, \mathbb{1} <: \mathbb{1}\}$ would result in the constraint $\gamma \oplus \delta <: \alpha$.

Calculate closed-form solutions Hopkins and Kozen [1999] define a closed-form solution for a set of pattern bounds $(\gamma_i <: \alpha_i)_i$: there exists a solution δ_i for each γ_i such that $\alpha_i \notin \mathsf{pv}(\delta_i)$. We can then substitute each closed pattern through the system to eliminate all pattern variables in the remaining constraints and obtain a system of closed inclusion constraints.

Translate to Presburger formulae and check satisfiability Finally, we translate the closed constraints into Presburger formulae. Commutative regular expressions, and therefore patterns, can be expressed as semilinear sets [Parikh 1966] that describe Presburger formulae [Ginsburg and Spanier 1966]. Since checking the satisfiability of a Presburger formula is

```
guard mb : Arg ⊙ Arg {
  receive Arg[x] from mb′ ↦
    guard mb′ : Arg {
      receive Arg[y] from mb″ ↦
        free mb″;
        x+y
    }
}
```

(a) Nested **guard**s

```
let (x, mb′) =
  guard mb : Arg ⊙ Arg {
    receive Arg[x] from mb′ ↦
      (x, mb′)
  } in
guard mb′ : Arg {
  receive Arg[y] from mb″ ↦
    free mb″; x+y
}
```

(b) Unnesting using products

```
guard self : M1 ⊙ M2 {
  receive M1[x] from mb′ ↦
    guard mb′ : M2 {
      receive M2[y] from mb″ ↦
        x!Go[];
        y!Go[];
        free mb″
    }
}
```

(c) Term requiring interfaces

Fig. 12. Examples motivating extensions

decidable, an external solver like Z3 [de Moura and Bjørner 2008] can be used to determine whether each constraint holds. In our case, we use Z3's quantifier elimination pass and its quantifier-free linear integer arithmetic solver.

5 EXTENSIONS

It is straightforward to extend Pat with product and sum types, and by using contextual typing information prior to constraint generation, we can add higher-order functions and *interfaces* that allow finer-grained alias analysis. The formalisation can be found in the extended version.

5.1 Product and Sum Types

Product and sum constructors are checking cases, and must contain only returnable components since we must be able to safely substitute their contents in any context. As with **let** expressions we can omit annotations on elimination forms, *i.e.* **let** $(x, y) = M$ **in** N or **case** V **of** $\{x \mapsto M; y \mapsto N\}$, provided that x and y are used in their continuations, or the sum or product consists of base types.

An advantage of adding product types is that we can avoid nested **guard** clauses, as we can return both a received value and an updated mailbox name. Figure 12a receives two integers and returns their sum using nested **guard** expressions, whereas Figure 12b uses products instead.

Since product types can only contain returnable components, they cannot be used to replace n-ary argument sequences in function definitions and **receive** clauses.

5.2 Using Contextual Type Information

A co-contextual approach is required to generate the pattern inclusion constraints. Sometimes, however, it is useful to have contextual type information *before* the constraint generation pass. Consider applying a first-class function: $(\lambda(x : \mathsf{Int}) : \mathsf{Int} \,.\, x)(5)$. Although the annotated λ expression allows us to synthesise a type and use a rule similar to TS-App, the lack of contextual type information means that the approach fails as soon as we stray from applying function literals as in **let** $f = (\lambda(x : \mathsf{Int}) : \mathsf{Int} \,.\, x)$ **in** $f(5)$). A typical backwards bidirectional typing approach requires *synthesising* function argument types, but this is too inflexible in our setting as each mailbox name argument would need a type annotation.

In the base system a global signature maps message tags to payload types. While technically convenient, this is inflexible. First, distinct entities may wish to use the same mailbox tags with different payload types. For example, a client may send a `Login` message containing credentials to a server, which may then send a `Login` message containing the credentials and a timestamp to a session management server. Second, we need a syntactic check on a **receive** guard to avoid aliasing, as outlined in §2: either the received payloads or free variables in the guard body must be

Tbl. 1. Typechecking concurrent actor examples in Pat

#	Name	Description	Strict	Time (ms)
Original mailbox calculus models taken from de'Liguoro and Padovani [2018]				
1	Lock	Concurrent lock modelling mutual exclusion	•	28.5
2	Future	Future variable that is written to once and read multiple times	•	22.5
3	Account	Concurrent accounts exchanging debit and credit instructions	•	19.5
4	AccountF	Concurrent accounts where debit instructions are effected via futures	•	33.5
5	Master-Worker	Master-worker parallel network	•	29.0
6	Session Types	Session-typed communicating actors using one arbiter	○	75.5
Selected micro-benchmarks adapted from Imam and Sarkar [2014], based on Neykova and Yoshida [2017b]				
7	Ping Pong	Process pair exchanging k ping and pong messages	•	24.6
8	Thread Ring	Ring network where actors cyclically relay one token with counter k	○	37.2
9	Counter	One actor sending messages to a second that sums the count, k	○	29.8
10	K-Fork	Fork-join pattern where a central actor delegates k requests to workers	•	7.1
11	Fibonacci	Fibonacci server delegating terms $(k-1)$ and $(k-2)$ to parallel actors	•	27.1
12	Big	Peer-to-peer network where actors exchange k messages randomly	○	62.8
13	Philosopher	Dining philosophers problem	○	57.1
14	Smokers	Centralised network where one arbiter allocates k messages to actors	○	31.3
15	Log Map	Computes the term $x_{k+1} = r \cdot x_k(1 - x_k)$ by delegating to parallel actors	○	57.9
16	Transaction	Request-reply actor communication initiated by a central teller actor	○	46.7

base types. This conservative check rules out innocuous cases such as in Figure 12c, which waits for messages from two actors before signalling them to continue.

With contextual information we can associate each mailbox name with an *interface* I, which maps tags to payload types, and allows us to syntactically distinguish different kinds of mailboxes (*e.g.* a future and its client). Since a name cannot have two interfaces at once, we can loosen our syntactic check on **receive** guards to require only that the *interfaces* of mailbox names in the payloads and free variables differ, as typing guarantees that they will refer to different mailboxes.

We implement the above extensions via a contextual type-directed translation: we annotate function applications with the type of the function (*i.e.* $V\,\overrightarrow{\tau} \rightarrow \sigma\,(\overrightarrow{W})$) which allows us to synthesise the function type. Users specify an interface when creating a mailbox (**new**$[I]$); our pass then annotates sends and guards with interface information (*i.e.* $V\,!^{I}\,\mathtt{m}\,[\overrightarrow{W}]$ and **guard**$^{I}\,V : E\,\{\overrightarrow{G}\}$) for use in constraint generation.

6 IMPLEMENTATION AND EXPRESSIVENESS

We outline the implementation of a prototype type checker written in OCaml, and evidence the expressiveness of Pat via a selection of example programs taken from the literature. We first show that using quasi-linear typing in place of dependency graphs (cf. §2.3) does not prevent Pat from expressing *all* of the examples in [de'Liguoro and Padovani 2018]. The Savina benchmarks [Imam and Sarkar 2014] capture typical concurrent communication patterns and are used both to compare actor languages and to demonstrate expressiveness; we show that Pat can express 10 of the 11 Savina expressiveness benchmarks used by Neykova and Yoshida [2017b]. Finally, we encode a case study provided by an industrial partner that develops control software for factories.

6.1 Implementation Overview

Pat programs are type checked in a six-phase pipeline: lexing and parsing; desugaring, which expands the sugared form of guards (*i.e.* rewrites **free** V as **guard** $V : \mathbb{1}\,\{\textbf{free} \mapsto ()\}$ and **fail** V as **guard** $V : \mathbb{0}\,\{\textbf{fail}\}$) and adds omitted pattern variables; IR conversion, which transforms the surface

language (supporting nested expressions) to our explicitly-sequenced intermediate representation; contextual type-checking, which supports the contextual extensions from §5.2; constraint generation; and constraint solving.

The Pat typechecker operates in two modes that determine how receive guards are type checked. *Strict* mode uses the lightweight syntactic checks outlined in §3 and §4, whereas *interface* mode uses interface type information (§5.2) to relax these checks. This means that every Pat program accepted in strict mode is also accepted in interface mode.

6.2 Expressiveness and Typechecking Time

Tbl. 1 lists the examples implemented in Pat. Examples 1-6 are the mailbox calculus examples from [de'Liguoro and Padovani 2018, Ex. 1–3, and Sec. 4.1–4.3]. Examples 7-16 are the selection of Savina benchmarks [Imam and Sarkar 2014, Table 1, No. 1–4, 6, 7, 12, 14–16] used in [Neykova and Yoshida 2017b]. The table indicates whether a Pat program can be checked in strict (denoted by •), *in addition* to interface mode (denoted by ∘). We report the mean typechecking time, excluding phases 1–3 of the pipeline. Measurements are made on a MacBook M1 Pro with 16GB of memory, running macOS 13.2 and OCaml 5.0, and averaging over 1000 repetitions.

6.2.1 Benchmarks. Tbl. 1 shows that all but one of the mailbox calculus examples from [de'Liguoro and Padovani 2018] can be checked in strict mode. The Savina examples capture typical concurrent programming patterns, namely, master-worker (K-Fork, Fibonacci, Log Map), client-server (Ping Pong, Counter), and peer-to-peer (Big), and common network topologies such as star (Philosopher, Smokers, Transaction) and ring (Thread Ring). Most of these programs require contextual type information (8, 9, and 12–16) to type check. As Pat does not yet support recursive types, we instead emulate fixed collections using definition parameters in examples 8, 10, 12–16. We could not encode the Sleeping Barber [Neykova and Yoshida 2017b, Ex. 8] example since the number of collection elements varies throughout execution.

The examples reveal the benefits of mailbox typing. Runtime checks, such as manual error handling (§1.2) are unnecessary since errors (*e.g.* unexpected messages) are *statically* ruled out by the type system. Mailbox types also have an edge over session typing tools for actor systems, *e.g.* [Neykova and Yoshida 2017b; Tabone and Francalanza 2022] where developers typically specify protocols in external tools and write code to accommodate the session typing framework. In contrast, mailbox typing *naturally* fits idiomatic actor programming.

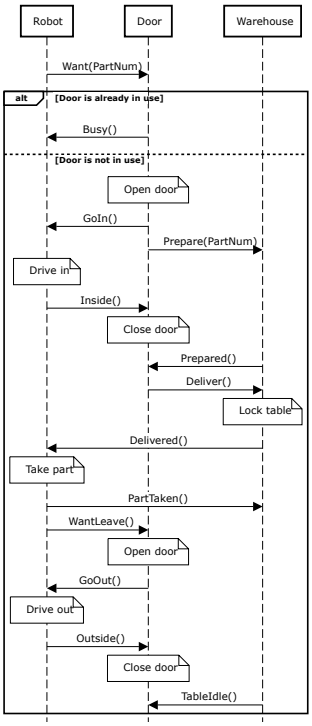


Fig. 13. Factory use case

This flexibility does not incur high typechecking runtime (see Tbl. 1). The aim of benchmarking typechecking time is to show that mailbox typechecking is not prohibitively expensive, rather than to claim comparative results. Comparisons with other implementations of (non-mailbox-typed versions of) the benchmarks written in other languages are unlikely to strengthen our results as the benchmark source code would be different, and we would be measuring *e.g.* Java's entire type system implementation rather than the essence of the typechecking algorithm.

6.2.2 Case Study. Finally we describe a real-world use case written by *Actyx AG*[3], who develop control software for factories. The use case captures a scenario where multiple robots on a factory floor acquire parts from a warehouse that provides access through a single door. Robots negotiate with the door to gain entry into the warehouse and obtain the part they require. The behaviour of our three entities, *Robot*, *Door*, and *Warehouse* is shown in Fig. 13. Our concrete syntax closely follows the core calculus of §3, without requiring that pattern variables in mailbox types are specified explicitly. Type checking our case study relies on contextual type information (see §5), and takes ≈89.6 ms.

We give an excerpt of our Warehouse process (below) that maps the interactions of its lifeline in Fig. 13. In its initial state, `empty`, the Warehouse expects a `Prepare` message (if there are Robots in the system), or none (if *no* Robot requests access), expressed as the guard `Prepare + 1` on line 2. When a part is requested, the Warehouse transitions to the state `engaged`, where it awaits a `Deliver` message from the Door and notifies the Robot via a `Delivered` message (lines 9–15). Subsequent interactions that the Warehouse undertakes with the Door and Robot are detailed in the extended version. Note that our type system enables us to be *precise* with respect to the messages mailboxes receive. Specifically, the guard on line 2 expects *at most* one `Prepare` message, capturing the mutual exclusion requirement between Robots, whereas the guard on line 10 expects *exactly* `Deliver`.

```
1  def empty(self: wh?): Unit {
2    guard self: Prepare + 1 {
3      free → ()
4      receive Prepare(partNum, door) from self →
5        door ! Prepared(self);
6        engaged(self)
7    }
8  }
9  def engaged(self: wh?): Unit {
10   guard self: Deliver {
11     receive Deliver(robot, door) from self →
12       robot ! Delivered(self, door);
13       given(self, door)
14   }
15 }
```

7 RELATED WORK

Behaviourally-typed actors. The asymmetric nature of mailboxes makes developing behavioural type systems for actor languages challenging. Mostrous and Vasconcelos [2011] investigate session typing for Core Erlang, using selective message reception and unique references to encode session-typed channels. Tabone and Francalanza [2021, 2022] develop a tool that statically checks Elixir [Jurić 2019] actors against binary session types to prove session fidelity. Neykova and Yoshida [2017b] propose a programming model for dynamically checking actor communication against multiparty session types [Honda et al. 2016], later implemented in Erlang by Fowler [2016]. Neykova and Yoshida [2017a] show how causality information in global types can support efficient recovery strategies. Harvey et al. [2021] use multiparty session types with explicit connection actions [Hu and Yoshida 2017] to give strong guarantees about actors that support runtime adaptation, but an actor can only participate in one session at a time. Using session types to structure communication requires specifying point-to-point interactions, typically using different libraries and formalisms. In contrast, our mailbox typing approach naturally fits idiomatic actor programming paradigms.

Bagherzadeh and Rajan [2017] define a type system for active objects [de Boer et al. 2007] which can rule out data races; this work targets an imperative calculus and is not validated via an implementation. Kamburjan et al. [2016] apply session-based reasoning to a core active object calculus where types encode remote calls and future resolutions; communication correctness is ensured by static checks against session automata [Bollig et al. 2013].

Mailbox types are inspired by behavioural type systems [Crafa and Padovani 2017] for the *objective join calculus* [Fournet and Gonthier 1996]. The technique can be implemented in Java using code generation via *matching automata* [Gerbo and Padovani 2019], and dependency graphs

[3] https://www.actyx.com

can rule out deadlocks [Padovani 2018a], but the authors do not consider a programming language design. Scalas et al. [2019] define a behavioural type system for Scala actors. Types are written in a domain-specific language, and type-level model checking determines safety and liveness properties. Their system focuses on the behaviour of a process, rather than the state of the mailbox.

Session-typed functional languages. Session types [Honda 1993; Honda et al. 1998] were originally considered in the setting of process calculi; Gay and Vasconcelos [2010] were first to integrate session types in a functional language by building on the linear λ-calculus, and their approach has been adopted by several other works (*e.g.* [Almeida et al. 2022; Lindley and Morris 2015]). Linear types are insufficient for mailbox typing since we require *multiple* uses of a mailbox name as a sender; we believe our use of quasi-linearity for behavioural typing is novel, and we conjecture that it could be used to support other paradigms (*e.g.* broadcast) that require non-linear variable use.

Co-contextual typing. Co-contextual typing [Erdweg et al. 2015] was originally introduced to support efficient incremental type-checking, and has also been used to support intrinsically-typed compilation [Rouvoet et al. 2021]. Padovani [2014] uses a co-contextual type algorithm for the linear π-calculus with sums, products, and recursive types; and Ciccone and Padovani [2022] use it when analysing fair termination properties. *Backwards bidirectional typing* [Zeilberger 2015] is a co-contextual formulation of bidirectional typing, and we are first to use it in a language implementation. Co-contextual typing has parallels with the co-de Bruijn nameless variable representation [McBride 2018], where subterms are annotated with the variables they contain.

Safety via static analysis. Christakis and Sagonas [2011] implement a static analyser for Erlang that detects errors such as receiving from an empty mailbox, payload mismatches, redundant patterns, and orphan messages. All of these issues can be detected with mailbox types, which also allow us to specify the mailbox state. Harrison [2018] implements an approach incorporating both typechecking and static analysis to detect errors such as orphan messages and redundant patterns.

8 CONCLUSION AND FUTURE WORK

Concurrent and distributed applications can harbour subtle and insidious bugs, including protocol violations and deadlocks. Behavioural types ensure *correct-by-construction* communication-centric software, but are difficult to apply to actor languages. We have proposed the *first* language design incorporating *mailbox types* which characterise mailbox communication. The multiple-writer, single-reader nature of mailbox-oriented messaging makes the integration of mailbox types in programming languages highly challenging. We have addressed these challenges through a novel use of quasi-linear types and have formalised and implemented an algorithmic type system based on backwards bidirectional typing (§4), proving it to be sound and complete with respect to the declarative type system (§3). Our approach can flexibly express common communication patterns (*e.g.* master-worker) and a real-world case study based on factory automation.

Future work. We are investigating implementing mailbox types in a tool for mainstream actor languages, *e.g.* Erlang; in parallel, we are investigating how languages with first-class mailboxes can be compiled to standard actor languages in order to leverage mature runtimes. We plan to consider finer-grained inter-process alias control, and co-contextual typing with *type* constraints (as well as pattern constraints), enabling us to study more advanced language features, *e.g.* polymorphism.

ACKNOWLEDGMENTS

We thank the anonymous ICFP reviewers and Artifact Evaluation Committee, as well as our STARDUST colleagues for their helpful comments. Thanks also to Roland Kuhn for discussion of the case study. This work was supported by EPSRC Grant EP/T014628/1 (STARDUST).

DATA AVAILABILITY STATEMENT

A typechecker for Pat, evaluated by the ICFP'23 Artifact Evaluation Committee, is available on Zenodo [Fowler et al. 2023a].

REFERENCES

Amal Ahmed, Matthew Fluet, and Greg Morrisett. 2007. L^3: A Linear Language with Locations. *Fundam. Informaticae* 77, 4 (2007), 397–449.

Bernardo Almeida, Andreia Mordido, Peter Thiemann, and Vasco T. Vasconcelos. 2022. Polymorphic lambda calculus with context-free session types. *Inf. Comput.* 289, Part (2022), 104948. https://doi.org/10.1016/0304-3975(83)90059-2

Roberto M. Amadio, Ilaria Castellani, and Davide Sangiorgi. 1998. On Bisimulations for the Asynchronous pi-Calculus. *Theor. Comput. Sci.* 195, 2 (1998), 291–324. https://doi.org/10.1016/S0304-3975(97)00223-5

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. 2016. Behavioral Types in Programming Languages. *Found. Trends Program. Lang.* 3, 2-3 (2016), 95–230. https://doi.org/10.1561/2500000031

Mehdi Bagherzadeh and Hridesh Rajan. 2017. Order types: static reasoning about message races in asynchronous message passing concurrency. In *AGERE!@SPLASH*. ACM, 21–30. https://doi.org/10.1145/3141834.3141837

Benedikt Bollig, Peter Habermehl, Martin Leucker, and Benjamin Monmege. 2013. A Fresh Approach to Learning Register Automata. In *Developments in Language Theory (LNCS, Vol. 7907)*. Springer, 118–130. https://doi.org/10.1007/978-3-642-38771-5_12

Janusz A Brzozowski. 1964. Derivatives of regular expressions. *Journal of the ACM (JACM)* 11, 4 (1964), 481–494. https://doi.org/10.1145/321239.321249

Avik Chaudhuri. 2009. A Concurrent ML library in Concurrent Haskell. In *ICFP*. ACM, 269–280. https://doi.org/10.1145/1596550.1596589

Maria Christakis and Konstantinos Sagonas. 2011. Detection of Asynchronous Message Passing Errors Using Static Analysis. In *PADL (Lecture Notes in Computer Science, Vol. 6539)*. Springer, 5–18. https://doi.org/10.1145/1140335.1140356

Luca Ciccone and Luca Padovani. 2022. Fair termination of binary sessions. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–30. https://doi.org/10.1145/3498666

Silvia Crafa and Luca Padovani. 2017. The Chemical Approach to Typestate-Oriented Programming. *ACM Trans. Program. Lang. Syst.* 39, 3 (2017), 13:1–13:45. https://doi.org/10.1145/3064849

Frank S. de Boer, Dave Clarke, and Einar Broch Johnsen. 2007. A Complete Guide to the Future. In *ESOP (Lecture Notes in Computer Science, Vol. 4421)*. Springer, 316–330. https://doi.org/10.1007/978-3-540-71316-6_22

Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (Lecture Notes in Computer Science, Vol. 4963)*. Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

Ugo de'Liguoro and Luca Padovani. 2018. Mailbox Types for Unordered Interactions. In *ECOOP (LIPIcs, Vol. 109)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 15:1–15:28. https://doi.org/10.4230/LIPIcs.ECOOP.2018.15

Jana Dunfield and Neel Krishnaswami. 2022. Bidirectional Typing. *ACM Comput. Surv.* 54, 5 (2022), 98:1–98:38. https://doi.org/10.1145/3450952

Robert Ennals, Richard Sharp, and Alan Mycroft. 2004. Linear Types for Packet Processing. In *ESOP (Lecture Notes in Computer Science, Vol. 2986)*. Springer, 204–218. https://doi.org/10.1007/978-3-540-24725-8_15

Sebastian Erdweg, Oliver Bracevac, Edlira Kuci, Matthias Krebs, and Mira Mezini. 2015. A co-contextual formulation of type rules and its application to incremental type checking. In *OOPSLA*. ACM, 880–897. https://doi.org/10.1145/2814270.2814277

Cédric Fournet and Georges Gonthier. 1996. The Reflexive CHAM and the Join-Calculus. In *POPL*. ACM Press, 372–385. https://doi.org/10.1145/237721.237805

Simon Fowler. 2016. An Erlang Implementation of Multiparty Session Actors. In *ICE (EPTCS, Vol. 223)*. 36–50. https://doi.org/10.4204/EPTCS.223.3

Simon Fowler, Duncan Paul Attard, Franciszek Sowul, Simon J. Gay, and Phil Trinder. 2023a. *Artifact for "Special Delivery: Programming with Mailbox Types"*. https://doi.org/10.5281/zenodo.8126809

Simon Fowler, Duncan Paul Attard, Franciszek Sowul, Simon J. Gay, and Phil Trinder. 2023b. Special Delivery: Programming with Mailbox Types (Extended Version). arXiv:2306.12935 [cs.PL]

Simon Fowler, Wen Kokke, Ornela Dardha, Sam Lindley, and J. Garrett Morris. 2023c. Separating Sessions Smoothly. *Logical Methods in Computer Science* 13, 3 (2023). https://doi.org/10.46298/lmcs-19(3:3)2023

Simon Fowler, Sam Lindley, and Philip Wadler. 2017. Mixing Metaphors: Actors as Channels and Channels as Actors. In *ECOOP (LIPIcs, Vol. 74)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 11:1–11:28. https://doi.org/10.4230/LIPIcs.ECOOP.2017.11

Simon J. Gay and Vasco Thudichum Vasconcelos. 2010. Linear type theory for asynchronous session types. *J. Funct. Program.* 20, 1 (2010), 19–50. https://doi.org/10.1017/S0956796809990268

Rosita Gerbo and Luca Padovani. 2019. Concurrent Typestate-Oriented Programming in Java. In *PLACES@ETAPS (EPTCS, Vol. 291)*. 24–34. https://doi.org/10.4204/EPTCS.291.3

Seymour Ginsburg and Edwin Spanier. 1966. Semigroups, Presburger formulas, and languages. *Pacific journal of Mathematics* 16, 2 (1966), 285–296. https://doi.org/10.2307/2271032

Joseph R. Harrison. 2018. Automatic detection of core Erlang message passing errors. In *Erlang Workshop*. ACM, 37–48. https://doi.org/10.1145/3239332.3242765

Paul Harvey, Simon Fowler, Ornela Dardha, and Simon J. Gay. 2021. Multiparty Session Types for Safe Runtime Adaptation in an Actor Language. In *ECOOP (LIPIcs, Vol. 194)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 10:1–10:30.

Jiansen He, Philip Wadler, and Philip W. Trinder. 2014. Typecasting actors: from Akka to TAkka. In *SCALA@ECOOP*. ACM, 23–33. https://doi.org/10.1145/2637647.2637651

Kohei Honda. 1993. Types for Dyadic Interaction. In *CONCUR (Lecture Notes in Computer Science, Vol. 715)*. Springer, 509–523. https://doi.org/10.1007/3-540-57208-2_35

Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. 1998. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *ESOP (Lecture Notes in Computer Science, Vol. 1381)*. Springer, 122–138. https://doi.org/10.1007/BFb0053567

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2016. Multiparty Asynchronous Session Types. *J. ACM* 63, 1 (2016), 9:1–9:67. https://doi.org/10.1145/2827695

Mark W. Hopkins and Dexter Kozen. 1999. Parikh's Theorem in Commutative Kleene Algebra. In *LICS*. IEEE Computer Society, 394–401. https://doi.org/10.1109/LICS.1999.782634

Raymond Hu and Nobuko Yoshida. 2017. Explicit Connection Actions in Multiparty Session Types. In *FASE (Lecture Notes in Computer Science, Vol. 10202)*. Springer, 116–133. https://doi.org/10.1007/978-3-642-21464-6_7

Raymond Hu, Nobuko Yoshida, and Kohei Honda. 2008. Session-Based Distributed Programming in Java. In *ECOOP (Lecture Notes in Computer Science, Vol. 5142)*. Springer, 516–541. https://doi.org/10.1007/978-3-540-70592-5_22

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. 2016. Foundations of Session Types and Behavioural Contracts. *ACM Comput. Surv.* 49, 1 (2016), 3:1–3:36. https://doi.org/10.1145/2873052

Shams Mahmood Imam and Vivek Sarkar. 2014. Savina - An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries. In *AGERE!@SPLASH*. ACM, 67–80. https://doi.org/10.1145/2687357.2687368

Saša Jurić. 2019. *Elixir in Action*. Manning.

Eduard Kamburjan, Crystal Chang Din, and Tzu-Chun Chen. 2016. Session-Based Compositional Analysis for Actor-Based Languages Using Futures. In *ICFEM (Lecture Notes in Computer Science, Vol. 10009)*. 296–312. https://doi.org/10.1007/978-3-319-47846-3_19

Naoki Kobayashi. 1999. Quasi-Linear Types. In *POPL*. ACM, 29–42. https://doi.org/10.1145/292540.292546

Edlira Kuci, Sebastian Erdweg, Oliver Bracevac, Andi Bejleri, and Mira Mezini. 2017. A Co-contextual Type Checker for Featherweight Java. In *ECOOP (LIPIcs, Vol. 74)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 18:1–18:26. https://doi.org/10.4230/LIPIcs.ECOOP.2017.18

Paul Blain Levy, John Power, and Hayo Thielecke. 2003. Modelling environments in call-by-value programming languages. *Information and Computation* 185, 2 (2003), 182–210. https://doi.org/10.1016/S0890-5401(03)00088-9

Sam Lindley and J. Garrett Morris. 2015. A Semantics for Propositions as Sessions. In *ESOP (Lecture Notes in Computer Science, Vol. 9032)*. Springer, 560–584. https://doi.org/10.1007/978-3-662-46669-8_23

Conor McBride. 2018. Everybody's Got To Be Somewhere. In *MSFP@FSCD (EPTCS, Vol. 275)*. 53–69. https://doi.org/10.4204/EPTCS.275.6

Dimitris Mostrous and Vasco Thudichum Vasconcelos. 2011. Session Typing for a Featherweight Erlang. In *COORDINATION (Lecture Notes in Computer Science, Vol. 6721)*. Springer, 95–109.

Rumyana Neykova and Nobuko Yoshida. 2017a. Let it recover: multiparty protocol-induced recovery. In *CC*. ACM, 98–108. https://doi.org/10.1145/3033019.3033031

Rumyana Neykova and Nobuko Yoshida. 2017b. Multiparty Session Actors. *Logical Methods in Computer Science* 13, 1 (2017). https://doi.org/10.23638/LMCS-13(1:17)2017

Leo Osvald, Grégory M. Essertel, Xilun Wu, Lilliam I. González Alayón, and Tiark Rompf. 2016. Gentrification gone too far? affordable 2nd-class values for fun and (co-)effect. In *OOPSLA*. ACM, 234–251.

Luca Padovani. 2014. Type Reconstruction for the Linear π-Calculus with Composite and Equi-Recursive Types. In *FoSSaCS (Lecture Notes in Computer Science, Vol. 8412)*. Springer, 88–102. https://doi.org/10.1007/978-3-642-54830-7_6

Luca Padovani. 2018a. Deadlock-Free Typestate-Oriented Programming. *Art Sci. Eng. Program.* 2, 3 (2018), 15. https://doi.org/10.22152/programming-journal.org/2018/2/15

Luca Padovani. 2018b. *Mailbox Calculus Checker*. https://boystrange.github.io/mcc/

Luca Padovani. 2018c. A type checking algorithm for concurrent object protocols. *Journal of Logical and Algebraic Methods in Programming* 100 (2018), 16–35. https://doi.org/10.1016/j.jlamp.2018.06.001

Luca Padovani. 2019. Context-Free Session Type Inference. *ACM Trans. Program. Lang. Syst.* 41, 2 (2019), 9:1–9:37. https://doi.org/10.1145/3229062

Rohit Parikh. 1966. On Context-Free Languages. *J. ACM* 13, 4 (1966), 570–581. https://doi.org/10.1145/321356.321364

Benjamin C. Pierce and David N. Turner. 2000. Local type inference. *ACM Trans. Program. Lang. Syst.* 22, 1 (2000), 1–44. https://doi.org/10.1145/345099.345100

Andrew M. Pitts. 1998. Existential Types: Logical Relations and Operational Equivalence. In *ICALP (Lecture Notes in Computer Science, Vol. 1443)*. Springer, 309–326. https://doi.org/10.1007/BFb0055063

Arjen Rouvoet, Robbert Krebbers, and Eelco Visser. 2021. Intrinsically typed compilation with nameless labels. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–28. https://doi.org/10.1145/3434303

Alceste Scalas, Nobuko Yoshida, and Elias Benussi. 2019. Verifying message-passing programs with dependent behavioural types. In *PLDI*. ACM, 502–516. https://doi.org/10.1145/3314221.3322484

Gerard Tabone and Adrian Francalanza. 2021. Session types in Elixir. In *AGERE!@SPLASH*. ACM, 12–23. https://doi.org/10.1145/3486601.3486708

Gerard Tabone and Adrian Francalanza. 2022. Session Fidelity for ElixirST: A Session-Based Type System for Elixir Modules. In *ICE (EPTCS, Vol. 365)*. 17–36. https://doi.org/10.4204/EPTCS.365.2

Kaku Takeuchi, Kohei Honda, and Makoto Kubo. 1994. An Interaction-based Language and its Typing System. In *PARLE (Lecture Notes in Computer Science, Vol. 817)*. Springer, 398–413. https://doi.org/10.1007/3-540-58184-7_118

Samira Tasharofi, Peter Dinges, and Ralph E. Johnson. 2013. Why Do Scala Developers Mix the Actor Model with other Concurrency Models?. In *ECOOP (Lecture Notes in Computer Science, Vol. 7920)*. Springer, 302–326. https://doi.org/10.1007/978-3-642-39038-8_13

Phil Trinder, Natalia Chechina, Nikolaos Papaspyrou, Konstantinos Sagonas, Simon Thompson, Stephen Adams, Stavros Aronis, Robert Baker, Eva Bihari, Olivier Boudeville, et al. 2017. Scaling reliably: Improving the scalability of the Erlang distributed actor platform. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 39, 4 (2017), 1–46. https://doi.org/10.1145/3107937

Vasco T. Vasconcelos. 2012. Fundamentals of session types. *Inf. Comput.* 217 (2012), 52–70. https://doi.org/10.1016/j.ic.2012.05.002

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

Noam Zeilberger. 2015. Balanced polymorphism and linear lambda calculus. Talk at TYPES. http://noamz.org/papers/linprin.pdf

Received 2023-03-01; accepted 2023-06-27