Special Delivery

Programming with Mailbox Types

SIMON FOWLER
University of Glasgow, UK
(email: Simon.Fowler@glasgow.ac.uk*)*

DUNCAN PAUL ATTARD
University of Malta, Malta
(email: Duncan.Attard@um.edu.mt*)*

DANIELLE MARSHALL
Royal Holloway, University of London, UK
(email: Danielle.Marshall@rhul.ac.uk*)*

SIMON J. GAY
University of Glasgow, UK
(email: Simon.Gay@glasgow.ac.uk*)*

PHIL TRINDER
University of Glasgow, UK
(email: Phil.Trinder@glasgow.ac.uk*)*

Abstract

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. Although actors eliminate many of the issues stemming from shared memory concurrency, they remain vulnerable to communication errors such as protocol violations and deadlocks. Behavioural types make it possible to detect communication errors early in the development process, but most work has addressed channel-based languages rather than actor languages.

Mailbox types are a recent behavioural type system for actors first introduced for a process calculus by de'Liguoro and Padovani in 2018, and 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 its algorithmic type system. Pat is a higher-order functional language with sums, products, and lists, along with interfaces that allow fine-grained reasoning about mailbox contents. Typechecking in Pat detects four classes of behavioural error: protocol violation, unexpected message, forgotten reply, and self-deadlock, as well as the usual data type errors.

Pat's type system makes essential use of quasi-linear typing to tame the challenges that aliasing poses for static typing. We introduce a *co-contextual* algorithmic type system, based around a novel use of backwards bidirectional typing, and prove the algorithmic type system sound and complete with respect to the declarative type system. We implement a mailbox type checker, and use it to demonstrate the expressiveness of Pat on a factory automation case study and a series of 11 examples from the Savina actor benchmark suite. These examples show that Pat matches the expressiveness of previous work on dynamically checked session types for actors, and our experimental results show that typechecking times are below 100ms. Our work establishes a foundation for applying mailbox typing to practical actor languages such as Erlang, Elixir and Scala/Akka.



© 2026 Copyright held by the owner/author(s).
This work is licensed under a Creative Commons Attribution 4.0 International License.

Journal of Functional Programming, Volume 36, Article None DOI: 10.46298/jfp.18225
Submitted July 2025; accepted September 2026

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 communication-centric language. In *channel-based* languages like Go, 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. A mailbox is an incoming message queue; in certain actor languages, the recipient can select which message from the mailbox to handle next.

Although communication-centric languages eliminate many concurrency 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.

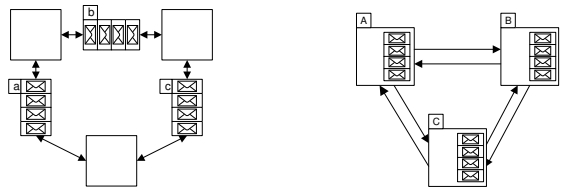
Behavioural type systems (Hüttel et al. 2016) encode correct communication behaviour to support *correct-by-construction* concurrency. Behavioural type systems, in particular *session types* (Gay and Vasconcelos 2025; Honda 1993; Honda et al. 1998; Takeuchi et al. 1994), have been extensively applied 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 (Fowler and Hu 2026; Harvey et al. 2021; Mostrous and Vasconcelos 2011; Tabone and Francalanza 2021, 2022) or relies on dynamic typing (Neykova and Yoshida 2017b). In general, session types are built around the idea of ordered communication channels. Channels are a significantly different communication model to the many-to-one unordered communication model supported by mailboxes, and therefore using session types in an actor language typically requires users to rewrite their applications. We discuss these systems further in §7.

Our approach is based on *mailbox types*, a behavioural type system for actor 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 co-contextual (Erdweg et al. 2015; Kuci et al. 2017), achieved through a novel use of backwards bidirectional typing (Zeilberger 2015).

1.1 Actor languages: easy to distribute, difficult to type

Channel-based languages support anonymous processes that communicate over named channels, whereas actor languages provide 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



(a) Channels (b) Actors

Fig. 1. Channel- and actor-based communication (Fowler et al. 2017)

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. In channel-based languages, since it is possible to send channel names over other channels, such languages 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 a channel-based language, adding a type system to an actor language is more difficult, 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 in channel-based languages 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 provided by mailboxes. 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 (Fowler and Hu 2026; Harvey et al. 2021; Neykova and Yoshida 2017b), requiring a developer to rewrite their application to structure its communication using session types.

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 senders.

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

The `empty_future` function awaits a `put` message to set the value of the future (lines 3–5), and transitions to the `full_future` state. A `full_future` receives `get` messages (lines 7–13) containing a process ID (PID) used to reply with the future's value. The `client` function spawns a future (line 15), sends a `put` message followed by a `get` message (lines 16–17), and awaits the result (lines 18–21). The program prints the number 5.

Several communication errors can arise in this example:

- **Protocol violation.** Receiving two `put` messages 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:

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

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 a future in the mailbox calculus as follows, where the mailbox is denoted *self*.

$$
\begin{array}{rcl}
\mathsf{emptyFuture}(\mathit{self}) & \triangleq & \mathit{self}\,?\,\mathtt{Put}(x)\,.\,\mathsf{fullFuture}(\mathit{self}, x) \\
\mathsf{fullFuture}(\mathit{self}, x) & \triangleq & \textbf{free}\ \mathit{self}\,.\,\textbf{done} \\
& + & \mathit{self}\,?\,\mathtt{Get}(\mathit{sender})\,.\,(\mathit{sender}\,!\,\mathtt{Reply}(x) \parallel \mathsf{fullFuture}(\mathit{self}, x)) \\
& + & \mathit{self}\,?\,\mathtt{Put}(x)\,.\,\textbf{fail}\ \mathit{self}
\end{array}
$$

$$
\begin{array}{l}
(\nu \mathit{future})(\mathsf{emptyFuture}(\mathit{future}) \parallel \mathit{future}\,!\,\mathtt{Put}(5) \parallel \\
\qquad (\nu \mathit{self})(\mathit{future}\,!\,\mathtt{Get}(\mathit{self}) \parallel (\mathit{self}\,?\,\mathtt{Reply}(x)\,.\,\textbf{free}\ \mathit{self}\,.\,\mathsf{print}(\mathsf{intToString}(x)))))
\end{array}
$$

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 issue is 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**) that 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 functional programming language that supports mailbox types, in which we express the *future* example as follows (*self* is again the future's 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))
  }
}
```

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 main difference 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** expression acts on a mailbox and may contain two kinds of clause: **free** $\mapsto M$ frees the mailbox if it contains no messages and 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

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

mailbox variable (with an updated type) to y in continuation M. In the above code, **free** *self* is syntactic sugar (see §3).

Pat has all of the key 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 is an expression-based language that supports **let**-bindings, which enable sequential composition and nested evaluation contexts, along with data types and higher-order functions.

Scope of guarantees. Crucially *all of the classes of concurrency error described earlier (protocol violations, unexpected messages, forgotten replies, and self-deadlocks) result in type errors*. As with most behavioural type systems for higher-order functional languages, ruling out inter-process deadlocks requires additional machinery such as priorities (Kobayashi 2006) or restricting communication topologies to make cycles impossible to construct (Lindley and Morris 2015; Wadler 2014). The mailbox calculus (de'Liguoro and Padovani 2018) achieves deadlock-freedom through dependency graphs, which cannot be applied in a language with aliasing and nested evaluation (§2.2), so we leave a full study of inter-process deadlock detection for Pat to future work.

1.3 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 functional programming language supporting mailbox types, making essential and novel use of quasi-linear types. We show type preservation, mailbox conformance, and a progress result. Mailbox conformance ensures that a process will not receive a message from a mailbox other than those described by the mailbox's type, and rules out protocol violations. Our progress result shows the absence of self-deadlocks, and also implies the absence of unexpected messages and of forgotten replies.
(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, product, and list types; higher-order functions; and mailbox interfaces (§5).
(4) We outline our implementation (§6), and demonstrate the expressiveness of Pat by encoding all of the examples from de'Liguoro and Padovani (2018), and all 11 of the Savina benchmarks (Imam and Sarkar 2014) used by Neykova and Yoshida (2017b) in their evaluation of multiparty session types for actor languages (§6.2). We also detail a larger factory case study.

This paper is a significantly extended and revised version of a paper of the same name that was published at ICFP'23 (Fowler et al. 2023a). This version includes thoroughly revised formal core languages that remove unnecessary annotations in the declarative type system and enable a simplified metatheory for the algorithmic type system. We also incorporate all

definitions omitted from the original conference paper, add many more examples throughout, and give more proof details for the main results. Other highlights include:

- Full technical details of the sum and product type extensions (§5).
- A new extension to allow Pat to express and type lists (§5.1.3), and a revised evaluation showing how to use lists for the relevant Savina examples (§6.2).
- Full declarative typing rules for extensions of Pat with first-class functions (§5.3) and mailbox interfaces (§5.4), and full details of how to use contextual type information to typecheck these features (§5.5).
- A more complete description of the implementation of the Pat typechecker (§6.1).
- A new extensive worked example showing how to express Dijkstra's classic *Sleeping Barber* concurrency problem in Pat, along with a more complete discussion of the implementation of the factory case study (§6.2).

A yet more comprehensive version (Fowler et al. 2026) contains full proofs. The Pat typechecker is available on GitHub (https://www.github.com/mailbox-types/pat-lang).

2 Mailbox types in a programming language: what are the issues?

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. 2023b; Gay and Vasconcelos 2010; Wadler 2014) introduced session types for languages based on the linear λ-calculus. In contrast, the more relaxed view of linearity in the mailbox calculus (de'Liguoro and Padovani 2018) 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, *receives* determine the later behaviour of the actor. To illustrate, consider the following code that shows part of the future example from §1:

```
1.  def client(): 1 {
2.    let future = new in
3.    spawn emptyFuture(future);
4.    let self = new in
5.    future ! Put(5);
6.    future ! Get(self);
7.    guard self : Reply {
8.      receive Reply(result) from self ↦
9.        free self;
10.       print(intToString(result))
11.   }
12. }
```

The client definition uses the *future* mailbox twice to send a message (lines 5 and 6), and similarly uses the *self* mailbox twice: once as a message payload (line 6), and once to receive a message (line 7). 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, in the fullFuture function described earlier, there is a strong expectation of how resources are used: the *self* mailbox name is consumed by the **guard** expression and only re-bound as *self* in the **receive** clause; notably the *self* variable is *not* available in the **free** clause as we would otherwise be able to send to a mailbox after it was deallocated.

We could try to write a function that attempts to use a mailbox after it has been freed:

```
def unsafeUse1(x : ?Msg()⋆): 1 {
  guard x : Msg⋆ {
    receive Msg() from z ↦
      x ! Msg();
      unsafeUse1(z)
    free ↦ x ! Msg()
  }
}
```

Such errors cannot arise in a fully linear type system, since it is not possible to 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 the previous example. Alas, the check can easily be circumvented by introducing an alias a for x:

```
def unsafeUse2(x : ?Msg()⋆): 1 {
  let a = x in
  guard a : Msg⋆ {
    receive Msg() from z ↦
      x ! Msg();
      unsafeUse2(z)
    free ↦ x ! Msg()
  }
}
```

Here, the new name prevents the typechecker from realising that it has been used in the body of the guard. Worse, we must also handle nested evaluation contexts, meaning that the next use of a mailbox variable is not necessarily contained within a subexpression of the guard:

```
def unsafeUse3(x : ?Msg()⋆): 1 {
  let _ =
    guard x : Msg⋆ {
      receive Msg() from z ↦
        x ! Msg();
        unsafeUse3(z)
      free ↦ x ! Msg()
  } in x ! Msg()
}
```

Much of the intricacy arises from using a mailbox name many times as an output capability. In each process, we can avoid the name hygiene 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. We further need to consider the possibility that aliasing is introduced as a consequence of communication. Consider the following example, where mailbox a receives the message $\mathtt{m}(b)$, and b is already free in the continuation of the **receive** clause:

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

In the above example, $a \leftarrow \mathtt{m}(b)$ refers to a message $\mathtt{m}(b)$ that has been sent to mailbox a. 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 (de'Liguoro and Padovani 2018) uses a *dependency graph* (DG) both to avoid issues with aliasing and to eliminate cyclic dependencies and hence deadlocks. In a dependency graph, the vertices are mailbox names, and an edge arises between two vertices if one name depends on another. Specifically, a dependency arises between two names a and b if b appears as a payload in a message addressed to a (meaning that a process must receive from a before being able to use b) or if b appears in the continuation of a process that must first receive from a. 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.

Alas, a language implementation *cannot* use this approach as it relies on knowing runtime names directly. To see why, consider the following Pat program, which evaluates to an analogous configuration:

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

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 could be 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 a static identity (*e.g.*, !‘`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 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 contexts of both 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-sender, single-receiver communication model used by the mailbox calculus. We augment mailbox types with a *usage*: either •, a *returnable* reference that allows a type to appear in the return type of an expression; or ∘, 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; and
(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 unsafeUse1, x is consumed by the **guard** expression and cannot be used thereafter. In unsafeUse2, 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 unsafeUse3, since x is used as the subject of a **guard** expression, that use must be returnable 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.

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 all of the 11 Savina benchmarks (Imam and Sarkar 2014) used by Neykova and Yoshida (2017b) to demonstrate the expressiveness of behavioural type systems for actor languages (§6.2).

Syntax of types

Mailbox types	J, K	::=	$!E \mid ?E$
Mailbox patterns	E, F	::=	$\mathbb{0} \mid \mathbb{1} \mid \mathtt{m} \mid E \oplus F \mid E \odot F \mid E^\star$
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	::=	$C \mid J^\eta$
Type environments	Γ, Π	::=	$\cdot \mid \Gamma, x : A$

Syntax of terms

Variables	x, y, z		
Definition names	f		
Definitions	D	::=	$\mathbf{def}\ f(\overrightarrow{x : A}) \colon B\ \{M\}$
Values	V, W	::=	$x \mid c$
Computations	L, M, N	::=	$V \mid \mathbf{let}\ x = M\ \mathbf{in}\ N \mid f(\overrightarrow{V})$
		$\mid$	$\mathbf{spawn}\ M \mid \mathbf{new} \mid V\,!\,\mathtt{m}(\overrightarrow{W}) \mid \mathbf{guard}\ V\ \{\overrightarrow{G}\} \mid \mathbf{fail}\ V$
Guards	G	::=	$\mathbf{free} \mapsto M \mid \mathbf{receive}\ \mathtt{m}(\overrightarrow{x})\ \mathbf{from}\ y \mapsto M$

Fig. 2. The syntax of Pat, a core language with mailbox types

3 Pat: a core language with mailbox types

This section introduces Pat, a core functional programming language with mailbox types, along with a declarative type system and an operational semantics. We begin by defining Pat as a first-order language, and later show how it extends to a higher-order language in §5.3.

3.1 Syntax

Figure 2 shows the syntax for Pat. We defer discussion of types to §3.2.

Programs and definitions. A program $\mathcal{P} = (\mathcal{S}, \overrightarrow{D}, M)$ consists of a *signature* $\mathcal{S}$ that maps message tags (*e.g.*, `Put` and `Get` from the future example) to payload types; a set of *definitions* $\overrightarrow{D}$; and an *initial term* M. Each definition $\mathbf{def}\ f(\overrightarrow{x : A}) \colon B\ \{M\}$ 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 to introduce a syntactic distinction between values and computations, inspired by *fine-grain call-by-value* (Levy et al. 2003), in order to simplify our typing rules. 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 trivial computation. The only evaluation context is $\mathbf{let}\ x = M\ \mathbf{in}\ N$, which evaluates term M and binds its result to x in N. Function application $f(\overrightarrow{V})$ applies function f to arguments $\overrightarrow{V}$. As usual, we use $M; N$ as sugar for $\mathbf{let}\ x = M\ \mathbf{in}\ N$, where x does not occur free in N.

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

The **guard** V $\{\overrightarrow{G}\}$ construct inspects mailbox V and potentially invokes a guard in $\overrightarrow{G}$. Note that although our examples have included mailbox patterns in **guard** expressions for clarity, and some annotations are required by our algorithmic type system (§4), annotations are not required in the core language used by the declarative type system.

In an untyped system, the **fail** V construct represents an irrecoverable failure and would be invoked if an unexpected message arrives (similar to raising an error in the Erlang future example in §1). In the typed setting, the **fail** V construct eliminates a mailbox V of type $?\mathbb{0}$; we will see in §3.4 that the type system ensures that **fail** V can never be evaluated.

As an example, consider the following modification of the fullFuture example from §1 that includes an additional **receive** guard for a `Put` message. This additional guard is not supported by the pattern $\texttt{Get}^\star$ or the FullFuture type.

```
def fullFuture(self : FullFuture, value : Int): 1 {
  guard self : Get* {
    free ↦ ()
    receive Get(user) from self ↦
      user ! Reply(value);
      fullFuture(self, value)
    receive Put(x) from self ↦
      fail self
  }
}
```

As we will see shortly, given the $\texttt{Get}^\star$ annotation on the guard and the fact that the *self* parameter has annotation FullFuture, in order to type this program, the type system would require the occurrence of *self* in the subject of the **guard** to have type $?(\mathbb{1} \oplus (\texttt{Get} \odot \texttt{Get}^\star) \oplus (\texttt{Put} \odot \mathbb{0}))$, which is equivalent to $?\texttt{Get}^\star$. Therefore, the binding occurrence of *self* in the `Put` case would have type $?\mathbb{0}$ and so can be eliminated using **fail**. Pat's *mailbox conformance* property ensures that a **fail** construct will never be evaluated.

Guards. The **free** $\mapsto M$ guard is triggered when a mailbox is empty and there are no more references to it in the system; and **receive** $\texttt{m}(\overrightarrow{x})$ **from** $y \mapsto M$ is triggered when the mailbox contains a message with tag `m`, binding its payloads to $\overrightarrow{x}$ and the continuation mailbox name with updated mailbox type to y in continuation term M.

We write **free** V as syntactic sugar for **guard** V $\{$**free** $\mapsto ()\}$. It only makes sense to free the *input* reference to a mailbox; output references are consumed upon sending a message.

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 $\mathtt{m}$. Note that unlike in §1, mailbox types in our formalism do 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. Pattern choice $E \oplus F$ denotes that the mailbox contains either messages conforming to pattern E or messages conforming to pattern F. Pattern composition $E \odot F$ denotes that the mailbox contains messages conforming to E and messages conforming to F (in either order). Finally, $E^\star$ denotes replication of E, so for example $\mathtt{m}^\star$ 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. We treat both $\oplus$ and $\odot$ as right-associative, so a pattern $\mathtt{m} \oplus \mathtt{n} \oplus \mathtt{p}$ should be read as $\mathtt{m} \oplus (\mathtt{n} \oplus \mathtt{p})$.

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 and formally define pattern semantics as follows:

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

$$[\![\mathtt{m}]\!] = \{\langle \mathtt{m} \rangle\} \qquad [\![E^\star]\!] = [\![\mathbb{1}]\!] \cup [\![E]\!] \cup [\![E \odot E]\!] \cup \cdots$$

The pattern $\mathbb{0}$ is interpreted as an empty set; $\mathbb{1}$ as a singleton set containing an empty multiset; $\oplus$ as set union; $\odot$ as pointwise multiset union; $\mathtt{m}$ as a singleton set containing the singleton multiset; and $E^\star$ as the possibly infinite set containing any number of compositions 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, so all base types are treated as returnable. Only values with a returnable type can be returned from an evaluation frame.

3.2.2 *Operations and predicates on types*

We say that a type is *returnable*, written $\mathsf{returnable}(A)$, if A is a base type C or a returnable mailbox type $J^\bullet$. The $\lfloor - \rfloor$ operator produces a returnable type from a non-annotated type, or changes an arbitrary mailbox type to be returnable. Similarly, the $\lceil - \rceil$ operator produces a type with a second-class usage annotation:

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

We also extend the operators to type environments in the usual way. The $\mathsf{base}(A)$ predicate holds if A is some base type C. We extend the predicate to environments: $\mathsf{base}(\Gamma)$ holds if for all $x : A \in \Gamma$, it follows that $\mathsf{base}(A)$.

Subtyping. Subtyping is crucial for mailbox typing. Rather than being an additional feature to increase the expressiveness of the system, subtyping is the core technical mechanism that allows us to determine whether an inferred pattern is contained within a specification, for example checking that $\texttt{Get} \odot \texttt{Get}$ is contained within a specification $\texttt{Get}^\star$. Subtyping relies on *pattern inclusion*. 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 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 relation defined by axioms* $\eta \leq \eta$ *and* $\bullet \leq \circ$*. We write* $A \simeq B$*, read "A is equivalent to 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: an input reference to a mailbox can safely be replaced with another that can receive more messages. Likewise subtyping is *contravariant* for mailboxes with a send capability: an output reference to 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:

Definition 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* $!\mathbb{0} \not\leq J$ *and* unusable *otherwise. A* type environment Γ *is reliable if all input mailbox types* $?E$ *in* Γ *are reliable.*

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. In the remainder of the paper we assume that all output mailbox types are usable: we therefore forbid types such as $!\mathbb{0}$ or $!(\texttt{m} \odot \mathbb{0})$.

Definition 3 (Unrestricted and linear types)**.** *We say that a type* A *is* unrestricted*, written* $\mathsf{un}(A)$*, if* A *is a base type or* $A = !\mathbb{1}^\circ$*. Otherwise, we say that* A *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.

Definition 4 (Environment subtyping)**.** *Environment subtyping* $\Gamma_1 \leq \Gamma_2$ *is the preorder relation on environments defined as follows:*

$$\frac{}{\Gamma \leq \Gamma} \qquad \frac{\Gamma_1 \leq \Gamma_2 \qquad \Gamma_2 \leq \Gamma_3}{\Gamma_1 \leq \Gamma_3} \qquad \frac{\mathsf{un}(A)}{\Gamma, x : A \leq \Gamma} \qquad \frac{A \leq B}{\Gamma, x : A \leq \Gamma, x : B}$$

The subtyping relation includes a notion of weakening, allowing an environment Γ to 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 $?(\mathtt{Put} \odot \mathtt{Get}^\star)$ results in a mailbox type $?(\mathtt{Get}^\star)$. 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 that 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, which would be unsafe.

Definition 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), for convenience we identify types up to commutativity and associativity, *e.g.*, we do not distinguish between $?(\mathtt{A} \odot \mathtt{B})^\bullet$ and $?(\mathtt{B} \odot \mathtt{A})^\bullet$, though since these patterns are semantically equivalent we can always rewrite one into another using subtyping. 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 $!\mathtt{A}$ and $?\mathtt{A}^\star$, we would need to use subtyping to rewrite the latter type to $?(\mathtt{A} \odot \mathtt{A}^\star)$).

The following *usage combination* operator is used to enforce quasi-linearity. Importantly, it is not commutative because 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 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, which is used extensively throughout the declarative type system in order to enforce quasi-linearity and mailbox typing within a single process. Since usage combination is not commutative, typing multiple concurrent processes at runtime uses an alternative environment combination operator, described in §3.4.

Definition 7 (Usage-annotated type combination)**.** *The* usage-annotated type combination operator $A \triangleright B$ *is the partial 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 8 (Environment combination (Γ))**.** *The* usage-annotated environment combination operator $\Gamma_1 \triangleright \Gamma_2$ *is the smallest partial operator on type environments closed under the following rules:*

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

$$\frac{\Gamma_1 \triangleright \Gamma_2 = \Gamma}{(\Gamma_1, x : A) \triangleright (\Gamma_2, x : B) = \Gamma, x : (A \triangleright B)}$$

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

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

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

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

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

Fig. 3. Pattern residual and pattern normal form

We use usage-annotated type combination when combining the types of a variable used in two successive 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 9 (Disjoint environment combination)**.** *Disjoint environment combination* $\Gamma_1 + \Gamma_2$ *is the smallest partial operator on type environments closed under the following rules:*

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

$$\frac{\Gamma_1 + \Gamma_2 = \Gamma}{(\Gamma_1, x : C) + (\Gamma_2, x : C) = \Gamma, x : C}$$

3.2.3 *Pattern residual and pattern normal form*

Before looking at the typing rules, it is useful to discuss the concepts of a *pattern residual* and *pattern normal form*, whose formal descriptions are given in Figure 3.

Pattern residual. Whenever we receive a message from a mailbox, the type of the receive reference to that mailbox should be updated to reflect that the message is no longer present. For instance, in our future example, the EmptyFuture type is $?(\mathtt{Put} \odot \mathtt{Get}^\star)$, but the FullFuture type (used after receiving the $\mathtt{Put}$ message) is $?(\mathtt{Get}^\star)$ (that is, the EmptyFuture type *without* the $\mathtt{Put}$ message).

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 pattern $\mathbb{0}$. The residual of $\mathtt{m}$ with respect to $\mathtt{m}$ is $\mathbb{1}$. The residual operator distributes over $\oplus$, and the residual of composition is the disjunction of the residual of each subpattern. In our example, $(\mathtt{Put} \odot \mathtt{Get}^\star) \,/\, \mathtt{Put} \simeq \mathtt{Get}^\star$.

Pattern normal form. When typing **guard** expressions, we require mailbox types to be in *pattern normal form* (PNF). A pattern E is in PNF if it is in the form $E_1 \oplus \cdots \oplus E_n$ where each E_i is $\mathbb{1}$ (used for typing a **free** guard) or of the form $\mathtt{m} \odot F$ (used for typing a **receive** guard, where F is equivalent to $E \,/\, \mathtt{m}$).

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

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

Typing rules for values and computations $\boxed{\Gamma \vdash_{\mathcal{P}} M : A}$

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

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

$$\text{T-Send}\ \frac{\mathcal{P}(\mathtt{m}) = \overrightarrow{T} \qquad \Gamma \vdash V : !\mathtt{m}^\circ \qquad (\Gamma'_i \vdash W_i : \lceil T_i \rceil)_{i\in 1..n}}{\Gamma + \Gamma'_1 + \cdots + \Gamma'_n \vdash V\,!\mathtt{m}(\overrightarrow{W}) : \mathbf{1}} \qquad \text{T-Guard}\ \frac{\Gamma_1 \vdash V : ?E^\bullet \qquad \{E\}\ \Gamma_2 \vdash \overrightarrow{G} : A \qquad \vDash E}{\Gamma_1 + \Gamma_2 \vdash \mathbf{guard}\ V\ \{\overrightarrow{G}\} : A}$$

$$\text{T-Fail}\ \frac{\Gamma \vdash V : ?\mathbb{0}^\bullet}{\Gamma \vdash \mathbf{fail}\ V : A} \qquad \text{T-Sub}\ \frac{\Gamma \leq \Gamma' \qquad A \leq B \qquad \Gamma' \vdash M : A}{\Gamma \vdash M : B}$$

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

$$\text{TG-GuardSeq}\ \frac{(\{E_i\}\ \Gamma \vdash G_i : A)_{i\in 1..n}}{\{E_1 \oplus \cdots \oplus E_n\}\ \Gamma \vdash \overrightarrow{G} : A}$$

$$\text{TG-Free}\ \frac{\Gamma \vdash M : A}{\{\mathbb{1}\}\ \Gamma \vdash \mathbf{free} \mapsto M : A} \qquad \text{TG-Recv}\ \frac{\mathcal{P}(\mathtt{m}) = \overrightarrow{T} \qquad \mathsf{base}(\overrightarrow{T}) \vee \mathsf{base}(\Gamma) \qquad \Gamma, y : ?E^\bullet, \overrightarrow{x} : \overrightarrow{\lceil T \rceil} \vdash M : B}{\{\mathtt{m} \odot E\}\ \Gamma \vdash \mathbf{receive}\ \mathtt{m}(\overrightarrow{x})\ \mathbf{from}\ y \mapsto M : B}$$

Fig. 4. Declarative typing rules for Pat

We express PNF using three judgements: Judgement $E \vDash F$ can be read "pattern F is in pattern normal form with respect to pattern E". Judgement $E \vDash_{\mathsf{lit}} F$ is similar but requires F to be free of pattern choice constructors ($\oplus$). Judgement $\vDash E$ can be read "pattern E is in pattern normal form", and holds if $E \vDash E$.

As an example, pattern $\mathbb{1} \oplus (\mathtt{m} \odot \mathtt{n})$ is in PNF: we could guard on a mailbox with this pattern by including a **free** guard and a **receive** guard that receives a message with tag $\mathtt{m}$. However, the equivalent pattern $(\mathbb{1} \odot \mathbb{1}) \oplus \mathbb{0} \oplus (\mathtt{m} \odot \mathtt{n})$ is *not* in PNF.

The definition of PNF relies on the definition of the pattern residual, and PNF is later used in the typing rule for **guard** expressions.

3.2.4 *Typing rules*

Figure 4 shows 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 the unit type $\mathbf{1}$. A definition $\textbf{def}\ f(\overrightarrow{x : A}) \colon B\ \{M\}$ 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 using environment subtyping in T-Sub. Rule T-Const types a constant under an empty environment; we assume an implicit schema mapping constants to types, and assume the existence of at least the unit value () of type $\mathbf{1}$. Rule T-App types application of a function f according to the definition of f 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 unsafe usage errors.

Rule T-Spawn types the **spawn** M construct, which spawns a unit-typed term M as a new process. The type environment used to type M can contain any number of returnable types, 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 in concurrent processes. Rule T-New creates a fresh mailbox with type $?\mathbb{1}^{\bullet}$, capturing the requirement that 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 returnable name. All payloads $\overrightarrow{W}$ 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 $\mathbf{1}$; behavioural typing is instead enforced using environment combination.

Rule T-Guard types the expression $\textbf{guard}\ V\ \{\overrightarrow{G}\}$, which retrieves from mailbox V using guards $\overrightarrow{G}$. The first premise ensures that under a type environment Γ_1, mailbox V has type $?E^{\bullet}$: the mailbox should have a receive capability with pattern E, and must be returnable. Demanding that the mailbox is returnable rules out unsafe usage 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 E. The final premise, $\vDash E$, ensures that E is in pattern normal form.

Rule T-Fail types the **fail** V construct, which is the elimination form for an unreliable mailbox with type $?\mathbb{0}$. The type system will ensure that this construct is never evaluated, and so it may have an arbitrary return type.

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 expression types.

Guards. Given a pattern of the form $E_1 \oplus \cdots \oplus E_n$, rule TG-GuardSeq types a sequence of guards, ensuring that each guard G_i is typable using pattern literal E_i under the same type environment and with the same return type. 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 $\textbf{receive}\ \mathtt{m}(\overrightarrow{x})\ \textbf{from}\ y \mapsto M$, which 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 (with an updated type) to y in continuation M. The payloads are made *second-class* rather than returnable, as otherwise the payloads could interfere with the names in the enclosing context and potentially violate the constraints required by quasi-linearity. The rule also introduces a conservative check to avoid aliasing by communication (§2): either all received payloads must have base types, or all free variables in the environment must have base types. We discuss a more liberal version of this condition in §5.4.

Example. We end this section by showing the derivation for the client definition from the example in §1, which creates a future mailbox and a 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 $\mathcal{S} = [\mathtt{Put} \mapsto \mathsf{Int}, \mathtt{Get} \mapsto\ !\mathtt{Reply}, \mathtt{Reply} \mapsto \mathsf{Int}]$, the emptyFuture and fullFuture definitions from §1, as well as the primitives print : String → **1** and intToString : Int → String. 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 $\mathbf{D}_1$:

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

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 single $\mathtt{Reply}$ message. The **receive** guard binds s at type $?\mathbb{1}^\bullet$ and r at Int, freeing s and using r in the print expression. The above derivation is used within derivation $\mathbf{D}_2$:

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

$$f : !(\mathtt{Put} \odot \mathtt{Get})^\bullet, s : ?\mathbb{1}^\bullet \vdash \begin{array}{l} f!\mathtt{Put}(5); f!\mathtt{Get}(s); \\ \textbf{guard}\ s\ \{\cdots\}\end{array} : \mathbf{1}$$

Here f is used to send a $\mathtt{Put}$ and then a $\mathtt{Get}$ with s of type $!\mathtt{Reply}^\circ$ as a payload. As the two sends to the f mailbox are sequentially composed, the type of f at the root of the

subderivation is $!(\texttt{Put} \odot \texttt{Get})^\bullet$. Since s is used at type $?(\texttt{Reply} \odot \mathbb{1})^\bullet$ 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:

$$\dfrac{\dfrac{\dfrac{f : ?(\texttt{Put} \odot \texttt{Get}^\star)^\bullet \vdash \mathsf{emptyFuture}(f) : \mathbf{1}}{f : ?(\texttt{Put} \odot \texttt{Get}^\star)^\circ \vdash \textbf{spawn}\ \mathsf{emptyFuture}(f) : \mathbf{1}}}{f : ?((\texttt{Put} \odot \texttt{Get}) \odot \mathbb{1})^\circ \vdash \textbf{spawn}\ \mathsf{emptyFuture}(f) : \mathbf{1}} \qquad \dfrac{\dfrac{}{\cdot \vdash \textbf{new} : ?\mathbb{1}^\bullet}\ \mathbf{D}_2}{f : !(\texttt{Put} \odot \texttt{Get})^\bullet \vdash \begin{array}{l}\textbf{let}\ s = \textbf{new}\ \textbf{in} \\ f!\texttt{Put}(5); \cdots\end{array} : \mathbf{1}}}{\dfrac{}{\cdot \vdash \textbf{new} : ?\mathbb{1}^\bullet} \qquad f : ?\mathbb{1}^\bullet \vdash \begin{array}{l}\textbf{spawn}\ \mathsf{emptyFuture}(f); \\ \textbf{let}\ s = \textbf{new}\ \textbf{in} \\ f!\texttt{Put}(5); \cdots\end{array} : \mathbf{1}}$$

$$\cdot \vdash \begin{array}{l}\textbf{let}\ f = \textbf{new}\ \textbf{in} \\ \textbf{spawn}\ \mathsf{emptyFuture}(f); \\ \textbf{let}\ s = \textbf{new}\ \textbf{in} \\ f!\texttt{Put}(5);\ f!\texttt{Get}(s); \\ \textbf{guard}\ s\ \{ \\ \quad \textbf{receive}\ \texttt{Reply}(r)\ \textbf{from}\ s \mapsto \\ \quad\quad \textbf{free}\ s; \\ \quad\quad \mathsf{print}(\mathsf{intToString}(r)) \\ \}\end{array} : \mathbf{1}$$

Since we let-bind f to **new**, f must have type $?\mathbb{1}^\bullet$. Definition emptyFuture requires an argument of type $?(\texttt{Put} \odot \texttt{Get}^\star)^\bullet$; 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 $?((\texttt{Put} \odot \texttt{Get}) \odot \mathbb{1})^\circ$. This then balances out with the use of f in $\mathbf{D}_2$, completing the derivation.

3.3 *Operational semantics*

Figure 5 shows the runtime syntax and reduction rules for Pat. To allow runtime names to occur in values and computations during reduction, we extend values V with runtime names a. We also extend type environments with mappings $a : J^\eta$ and lift the operations on type environments to these extended environments.

The concurrent semantics of the language is described as a nondeterministic reduction relation on a language of *configurations*, which resemble processes 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 $\mathtt{m}(\overrightarrow{V})$ sent to mailbox a; name restriction $(\nu a)C$ binds name a in C; and $C \parallel \mathcal{D}$ denotes the parallel composition of C and $\mathcal{D}$.

Structural congruence $C \equiv \mathcal{D}$ is defined as the smallest congruence relation satisfying the rules in Figure 5 and is standard, capturing the commutativity and associativity of parallel compositions; scope extrusion; and the unordered nature of name restrictions. We let $\mathsf{fn}(C)$ denote the set of free runtime names in configuration C.

Running a program. To run a program $(\mathcal{S}, \overrightarrow{D}, M)$, we place its body M into a thread with 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: specifically, whereas evaluation contexts are defined directly on the structure of terms, the explicit definition of frame stacks makes it more convenient to define runtime typing rules that allow us to more easily reason about quasi-linearity during reduction (see §3.4). A frame $\langle x, M \rangle$ is a pair of a variable x and a continuation M, where x binds the result of the current computation in M. A frame stack is an ordered sequence of frames, where ϵ denotes the empty stack.

Runtime syntax

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

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

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

$$\textsc{E-Recv}\quad (\!\mid \mathbf{guard}\ a\ \{\mathcal{G}[\mathbf{receive}\ \mathtt{m}(\overrightarrow{x})\ \mathbf{from}\ y \mapsto M]\}, \Sigma \mid\!) \parallel a \leftarrow \mathtt{m}(\overrightarrow{V}) \longrightarrow (\!\mid M\{\overrightarrow{V}/\overrightarrow{x}, a/y\}, \Sigma \mid\!)$$

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

Structural congruence $\boxed{\mathcal{C} \equiv \mathcal{D}}$

$$\mathcal{C} \parallel \mathcal{D} \equiv \mathcal{D} \parallel \mathcal{C} \qquad \mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{D}') \equiv (\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{D}'$$

$$\mathcal{C} \parallel (\nu a)\mathcal{D} \equiv (\nu a)(\mathcal{C} \parallel \mathcal{D}) \quad \text{if } a \notin \mathsf{fn}(\mathcal{C}) \qquad (\nu a)(\nu b)\mathcal{C} \equiv (\nu b)(\nu a)\mathcal{C}$$

Fig. 5. Pat operational semantics

Reduction rules. Frame stacks are best demonstrated by the E-Let and E-Return rules: intuitively, the **let** $x = M$ **in** N construct evaluates M, binding its result to x in N. To capture this behaviour, the E-Let 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 top frame on the stack: 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 $\overrightarrow{x}$.

Rule E-New creates a fresh mailbox name restriction and returns the fresh name. Rule E-Send sends a message with tag $\mathtt{m}$ and payloads $\overrightarrow{V}$ to a mailbox a: it evaluates to () and creates 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 the updated mailbox name to y in continuation M. The remaining rules are administrative.

3.4 Metatheory

In this section we describe Pat's metatheory: specifically type preservation and a progress result. Consequently we show that a well-typed Pat process will never process any messages that are not supported by its mailbox types (*mailbox conformance*), and that processes are free of self-deadlocks, unexpected messages, and forgotten replies. We achieve this by introducing a runtime type system to maintain inductive invariants during evaluation, and proving type preservation and a progress result.

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 $\Delta_1 \bowtie \Delta_2$ 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 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:*

$$\frac{}{\cdot \bowtie \cdot = \cdot} \qquad \frac{u \notin \mathit{dom}(\Delta_2) \qquad \Delta_1 \bowtie \Delta_2 = \Delta}{(\Delta_1, u{:}T) \bowtie \Delta_2 = \Delta, u{:}T} \qquad \frac{u \notin \mathit{dom}(\Delta_1) \qquad \Delta_1 \bowtie \Delta_2 = \Delta}{\Delta_1 \bowtie (\Delta_2, u{:}T) = \Delta, u{:}T}$$

$$\frac{\Delta_1 \bowtie \Delta_2 = \Delta}{(\Delta_1, u{:}T) \bowtie (\Delta_2, u{:}U) = \Delta, u{:}(T \boxplus U)}$$

We can derive a runtime environment from a type environment by erasing all usage annotations.

Definition 11 (Usage erasure)**.** *The* usage erasure *operator is defined as follows:*

$$|C| = C \qquad\qquad |J^{\eta}| = J$$

We extend the operator to type environments by applying erasure pointwise on types, i.e., $|u_1 : A_1, \ldots, u_n : A_n| = u_1 : |A_1|, \ldots, u_n : |A_n|$.

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

Runtime typing rules. Figure 6 shows the runtime typing rules. Rule TP-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 TP-Par allows configurations $\mathcal{C}$ and $\mathcal{D}$ to be composed in parallel if they are typable under combinable runtime type environments. Rule

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

$$\textsc{TP-Nu}\ \frac{\Delta, a : ?\mathbb{1} \vdash C}{\Delta \vdash (\nu a)C} \qquad \textsc{TP-Par}\ \frac{\Delta_1 \vdash C \qquad \Delta_2 \vdash \mathcal{D}}{\Delta_1 \bowtie \Delta_2 \vdash C \parallel \mathcal{D}}$$

$$\textsc{TP-Message}\ \frac{(\lceil \Delta_i \rceil \vdash V_i : A_i)_{i \in 1..n} \qquad \overrightarrow{A} \leq \lceil \mathcal{P}(\mathtt{m}) \rceil}{\Delta_1 + \cdots + \Delta_n, a : !\mathtt{m} \vdash a \leftarrow \mathtt{m}(\overrightarrow{V})}$$

$$\textsc{TP-Thread}\ \frac{\Delta = |\Gamma_1 \triangleright \Gamma_2| \qquad \Gamma_1 \vdash M : A \qquad \Gamma_2 \vdash A \blacktriangleright \Sigma}{\Delta \vdash (\!| M, \Sigma |\!)} \qquad \textsc{TP-Sub}\ \frac{\Delta \leq \Delta' \qquad \Delta' \vdash C}{\Delta \vdash C}$$

Frame stack typing $\boxed{\Gamma \vdash A \blacktriangleright \Sigma}$

$$\textsc{TF-Empty}\ \frac{}{\cdot \vdash A \blacktriangleright \epsilon} \qquad \textsc{TF-Frame}\ \frac{\Gamma_1, x : A \vdash M : B \qquad \mathsf{returnable}(A) \qquad \Gamma_2 \vdash B \blacktriangleright \Sigma}{\Gamma_1 \triangleright \Gamma_2 \vdash A \blacktriangleright \langle x, M \rangle \cdot \Sigma} \qquad \textsc{TF-Sub}\ \frac{\Gamma_1 \leq \Gamma_2 \qquad \Gamma_2 \vdash A \blacktriangleright \Sigma}{\Gamma_1 \vdash A \blacktriangleright \Sigma}$$

Fig. 6. Pat runtime typing

TP-Message types a message configuration $a \leftarrow \mathtt{m}(\overrightarrow{V})$. Since contexts can only contain a single occurrence of a variable, name a of type $!\mathtt{m}$ cannot appear in $\Delta_1, \ldots, \Delta_n$ and thus cannot occur in any of the values sent as a payload. This property is ensured in the corresponding static rule for message sends (T-Send) because the environments used to type the target of the sends and the message payloads are combined using the disjoint environment combination operator $\Gamma_1 + \Gamma_2$, and therefore cannot share mailbox-typed variables. 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 TP-Sub allows subtyping on runtime type environments; the subtyping relation $\Delta \leq \Delta'$ is analogous to subtyping on Γ.

Thread and frame stack typing. Rule TP-Thread types a thread, consisting of a currently-evaluating term (typable under Γ_1) and a frame stack (typable under Γ_2). Since the term and the stack frames execute sequentially, $\Gamma_1 \triangleright \Gamma_2$ must be defined. Because usage annotations are thread-local, the runtime environment needed to type the thread is $\Gamma_1 \triangleright \Gamma_2$ with annotations erased. TP-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 (TF-Empty). Rule TF-Frame details the typing rule for a non-empty frame stack $\langle x, M \rangle \cdot \Sigma$, which is well-typed if continuation M has type B, given a variable x of returnable 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 $!(\mathtt{m} \odot \mathtt{n})^\circ$. Finally, rule TF-Sub allows the use of subtyping when typing frame stacks.

3.4.2 Preservation

We can now state some metatheoretical results. We give outlines of the salient proofs here but full proofs can be found in the extended version (Fowler et al. 2026).

Typability is preserved by reduction; the proof is nontrivial since we must do extensive reasoning about environment combination.

To begin with, we need to tame some of the complexity introduced by environment subtyping, since environment subtyping allows a notion of weakening. Read top-down, T-Sub allows us to use environment subtyping to add a variable with mailbox type $!\mathbb{1}^\circ$ or base type C, or replace a type with its subtype. It is therefore useful to introduce a definition referring to environments that can be populated by repeated uses of T-Sub and which might not be used by a term. We call these environments *cruft*.

Definition 12 (Cruft)**.** *A type environment* Γ *is* cruft *if* $\Gamma \leq \cdot$.

It also helps to define a stricter version of environment subtyping that does not permit weakening:

Definition 13 (Strict environment subtyping)**.** *An environment* Γ *is a* strict subtype environment *of an environment* Γ'*, written* $\Gamma \preccurlyeq \Gamma'$*, if* $\Gamma \leq \Gamma'$ *and* $\mathsf{dom}(\Gamma) = \mathsf{dom}(\Gamma')$.

Let $\mathsf{fv}(-)$ denote the set of free variables in a term M or frame stack Σ. We can now characterise when an environment is *cruftless* for a given term.

Definition 14 (Cruftless)**.** *We say that an environment* Γ *is* cruftless for a term M *if* $\Gamma \vdash M : A$ *and* $\mathsf{dom}(\Gamma) = \mathsf{fv}(M)$.

The following lemma allows us to separate the type environment required for typing the term from the cruft introduced by environment subtyping, and is used extensively within the preservation proof.

Lemma 1. *If* $\Gamma \vdash M : A$*, then there exist* Π_1, Π_2, Π_3 *such that:*

- $\Gamma = \Pi_1, \Pi_2$
- $\Pi_3 \vdash M : A'$
- Π_1 *is cruftless for* M*, and* $\Pi_1 \preccurlyeq \Pi_3$
- $A' \leq A$
- $\mathsf{cruft}(\Pi_2)$

Proof. Follows from the definition of environment subtyping: read top-down, each application of environment subtyping will either add a variable with an unrestricted type, or alter the type of an existing variable. □

One of the most important lemmas used in the preservation proof uses quasi-linear typing to show that if two environments Π_1, Π_2 are combined with a third environment Γ using the $\triangleright$ operator (*i.e.*, Π_1, Π_2 are used to type an evaluation frame), and all types in environment Π_1 are returnable, then none of the mailbox variables in Π_1 are present in $\Pi_2 \triangleright \Gamma$. This lemma is crucial for reasoning about nested evaluation contexts and follows from the definition of usage combination.

Lemma 2. *If* $(\Pi_1, \Pi_2) \triangleright \Gamma$ *is defined and* $\mathsf{returnable}(\Pi_1)$*, then* $(\Pi_1, \Pi_2) \triangleright \Gamma = \Pi_1 + (\Pi_2 \triangleright \Gamma)$.

Proof. Follows from the definition of usage combination: the $\triangleright$ operation is not commutative for returnable mailbox types, so the returnable mailbox type must be the last occurrence of that name in the combination. For base types, the definitions of combination for $\triangleright$ and $+$ coincide. □

The preservation theorem is interesting in that the type environment remains the same before and after reduction, reflecting the observation that sends and receives on a mailbox must eventually "balance out". This reasoning is exemplified by the key *balancing lemma* used in the proof of preservation, showing the symmetry inherent in pattern inclusions:

Lemma 3 (Balancing (de'Liguoro and Padovani 2018))**.** *If* $\mathtt{m} \odot F \sqsubseteq E$ *and* $F \not\sqsubseteq \mathbb{0}$*, then* $F \sqsubseteq E \,/\, \mathtt{m}$.

We need several other lemmas to allow us to do equational reasoning on environments. For example, since we identify mailbox patterns up to associativity and commutativity, we can straightforwardly show that environment combination $\Gamma_1 \triangleright \Gamma_2$ is associative, and that runtime environment combination $\Delta_1 \bowtie \Delta_2$ is both associative and commutative.

Recall that a type environment Γ is reliable if all input mailbox types $?E^{\eta}$ in Γ are reliable, specifically that each $?E^{\eta_1} \not\leq ?\mathbb{0}^{\eta_2}$. We extend this definition to runtime environments Δ.

Theorem 1 (Preservation)**.** *If* $\vdash \mathcal{P}$*, and* $\Delta \vdash_{\mathcal{P}} C$ *with* Δ *reliable, and* $C \longrightarrow_{\mathcal{P}} \mathcal{D}$*, then* $\Delta \vdash_{\mathcal{P}} \mathcal{D}$.

Proof. By induction on the derivation of $C \longrightarrow \mathcal{D}$. □

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

Corollary 1 (Mailbox conformance)**.** *If* $\vdash \mathcal{P}$ *and* $\Delta \vdash_{\mathcal{P}} C$ *with* Δ *reliable, then* $C \not\longrightarrow^* \mathcal{H}[\,\mathbf{fail}\ V\,]$.

This corollary follows from preservation because, in order to evaluate **fail** V, we would need a mailbox with type $?\mathbb{0}^{\bullet}$ in the environment. This would contradict the premise that Δ is reliable.

3.4.3 Progress

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

Definition 15 (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* $\mathcal{M} \equiv a \leftarrow \mathtt{m}(\overrightarrow{V}) \parallel \mathcal{M}'$ *for some* $\mathcal{M}'$.

Next, we define *canonical forms*, which give us a global view of a configuration.

Definition 16 (Canonical form)**.** *A configuration* C *is in* canonical form *if it is of the form:*

$$(\nu a_1) \cdots (\nu a_m)((\!|\, M_1, \Sigma_1 \,|\!) \parallel \cdots \parallel (\!|\, M_n, \Sigma_n \,|\!) \parallel \mathcal{M})$$

Every configuration can be written in canonical form; the result follows from repeated application of the structural congruence rules. As a result we can reason about any arbitrary configuration by rewriting it in canonical form.

Proposition 1 (Canonical forms)**.** *For every configuration* C*, there exists some* $\mathcal{D}$ *such that* $C \equiv \mathcal{D}$ *and* $\mathcal{D}$ *is in canonical form.*

We need two further definitions to characterise a thread that is blocked while waiting for a message to arrive.

Definition 17 (Waiting)**.** *We say that a term M is* waiting on mailbox *a if M can be written* ***guard*** *$a\ \{\overrightarrow{G}\}$ for some guards $\overrightarrow{G}$.*

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* ***guard*** *$a\ \{\mathcal{G}[\textbf{receive}\ \mathtt{m}(\overrightarrow{x})\ \textbf{from}\ y \mapsto N]\}$ for some guard context $\mathcal{G}$. We say that M is a* waiting term *if there exists some a such that M is waiting on mailbox a.*

Definition 18 (Guard clauses)**.** *The* guard clauses *of a waiting term $M = \textbf{guard}\ V\ \{\overrightarrow{G}\}$ are the guards $\overrightarrow{G}$.*

With these definitions in hand, we can state the notion of progress enjoyed by Pat.

We begin by showing functional reduction, *i.e.*, that threads can always reduce up to communication and concurrency constructs.

Lemma 4 (Progress (functional reduction))**.** *If $\Delta \vdash_{\mathcal{P}} (\!| M, \Sigma |\!)$ with Δ reliable, then either:*

- *M is a value and $\Sigma = \epsilon$; or*
- *there exist some M', Σ' such that $(\!| M, \Sigma |\!) \longrightarrow (\!| M', \Sigma' |\!)$; or*
- *M is a communication and concurrency construct, i.e.,* ***new****, or* ***spawn*** *M, or $V\,!\,\mathtt{m}(\overrightarrow{W})$, or* ***guard*** *$V\ \{\overrightarrow{G}\}$.*

Proof. By induction on the derivation of $\Delta \vdash (\!| M, \Sigma |\!)$ and inspection of the reduction rules. □

We can then use canonical forms to characterise a progress result: either a configuration can reduce, or each constituent thread has either reduced to a value, or is waiting for a message that has not yet been sent by a different thread.

Theorem 2 (Partial progress)**.** *Suppose $\vdash \mathcal{P}$ and $\cdot \vdash_{\mathcal{P}} C$ where C is in canonical form:*

$$C = (\nu a_1) \cdots (\nu a_m)((\!| M_1, \Sigma_1 |\!) \parallel \cdots \parallel (\!| M_n, \Sigma_n |\!) \parallel \mathcal{M})$$

Then either there exists some C' such that $C \longrightarrow C'$, or for each M_i, either:

- *M_i is a value and $\Sigma_i = \epsilon$; or*
- *M_i is waiting on some mailbox a with guard clauses $\overrightarrow{G_i}$, and for all $\mathtt{m}_j$ such that $\mathsf{waiting}(M_i, a, \mathtt{m}_j)$, message set $\mathcal{M}$ does not contain a message $a \leftarrow \mathtt{m}_j(\overrightarrow{V})$, and $a \notin \mathit{fn}(\overrightarrow{G_i}) \cup \mathit{fn}(\Sigma_i)$.*

Proof. Functional reduction enjoys progress (Lemma 4), and the constructs **new**, **spawn** M, and $a\,!\,\mathtt{m}(\overrightarrow{V})$ can all always reduce by E-New, E-Spawn, or E-Send. Therefore, the body of an irreducible thread $(\!| M_k, \Sigma_k |\!)$ must be waiting: *i.e.*, it must be of the form $(\!| \textbf{guard}\ a\ \{\overrightarrow{G}\}, \Sigma_k |\!)$ for some name a and guards $\overrightarrow{G}$.

There are four cases to consider:

(1) That there exists some $\mathtt{m}$ such that $\mathsf{waiting}(M_k, a, \mathtt{m})$ and there exists some sent message $a \leftarrow \mathtt{m}(V)$ in the message set $\mathcal{M}$. In this case, we can reduce by E-Recv.

(2) That there exist no messages for a in $\mathcal{M}$, and a does not occur free in $\overrightarrow{G}$, Σ_k, or any other thread. In this case by TP-Nu, a must have some type J^{η} such that $?\mathbb{1}^{\bullet} \leq J^{\eta}$ and as such $\overrightarrow{G}$ must include a **free** guard. We can therefore reduce by E-Free.

(3) That a occurs free in $\overrightarrow{G}$ or Σ_k. This is impossible because the subject of a guard must be returnable, and therefore by Lemma 2 cannot occur in the guards or frame stack.

(4) That there exist no messages for a in $\mathcal{M}$ but a occurs free in some other waiting thread, indicating a cyclic inter-process dependency. This satisfies the second clause of the theorem statement. □

Consequences of progress. The progress theorem shows how Pat can rule out self-deadlocks, forgotten replies, and unexpected messages.

Specifically, a self-deadlock occurs when a process is waiting on a name a where the message is sent in a later part of the same thread: in our case, either the guard body or the frame stack. This is ruled out by the second case of the progress theorem.

A forgotten reply occurs when a process is waiting for a message that will never be sent. In this case, we would have a process that is waiting on a name a but no other thread has a reference to send the message. This is evidenced by the second case of the theorem statement and more specifically case (2) of the proof: the only way a thread can be waiting on a while no other threads have a reference to a is if the guard expression has a **free** guard, in which case the configuration would be able to free the mailbox.

An unexpected message occurs when a process receives a message that is not accounted for by any of its **receive** clauses: in a language like Erlang this would manifest itself as a process terminating while still containing messages in its mailbox. Again, this cannot occur: consider the case where all processes are terminated. By TP-Nu it would be impossible for there to be a message in the message set without a corresponding input reference.

Deadlocks. Although Pat rules out self-deadlocks, it does not rule out inter-process deadlocks. For example, the following (correct) processes encode a request-response pattern:

```
def requester(self : ?Response, other : !Request): 1 {
  other!Request();
  guard self:Response {
    receive Response() from self ↦ free self
  }
}

def responder(self : ?Request, other : !Response): 1 {
  guard self:Request {
    receive Request() from self ↦
      other!Response();
      free self
  }
}

def main(): 1 {
  let mb1 = new in
  let mb2 = new in
  spawn requester(mb1, mb2);
  spawn responder(mb2, mb1)
}
```

However, if we were to modify the requester process to send the request only after receiving the response, which would result in a deadlock, the program would still be accepted by our type system because we do not have any static way of ruling out inter-process deadlocks:

```
def badRequester(self : ?Response, other : !Request): 1 {
  guard self:Response {
    receive Response() from self ↦
      other!Request(); free self
  }
}
```

Pattern variables	α, β		
Mailbox patterns	γ, δ	::=	$\mathbb{0} \mid \mathbb{1} \mid \mathtt{m} \mid \gamma \oplus \delta \mid \gamma \odot \delta \mid \gamma^\star \mid \alpha$
Mailbox types	ς	::=	$!\gamma \mid ?\gamma$
Types	π, ρ	::=	$C \mid \varsigma$
Usage-annotated types	τ, σ	::=	$C \mid \varsigma^\eta$
Type environments	Θ	::=	$\cdot \mid \Theta, x : \tau$
Constraints	ϕ	::=	$\gamma <: \delta$
Constraint sets	Φ		

Open signatures	$\widehat{\mathcal{S}}$	::=	$\overrightarrow{\mathtt{m} \mapsto \overrightarrow{\pi}}$
Annotated definitions	$\widehat{D}$	::=	$\mathbf{def}\ f(\overrightarrow{x : \tau}) \colon \sigma\ \{\widehat{M}\}$
Annotated programs	$\widehat{\mathcal{P}}$	::=	$(\widehat{\mathcal{S}}, \overrightarrow{D}, \widehat{M})$
Annotated computations	$\widehat{M}, \widehat{N}$	::=	$V \mid \mathbf{let}\ x \colon A = \widehat{M}\ \mathbf{in}\ \widehat{N} \mid f(\overrightarrow{V})$
		$\mid$	$\mathbf{spawn}\ \widehat{M} \mid \mathbf{new} \mid V\,!\,\mathtt{m}(\overrightarrow{W}) \mid \mathbf{guard}\ V : E\ \{\overrightarrow{\widehat{G}}\} \mid \mathbf{fail}\ V$
Annotated guards	$\widehat{G}$	::=	$\mathbf{free} \mapsto \widehat{M} \mid \mathbf{receive}\ \mathtt{m}(\overrightarrow{x})\ \mathbf{from}\ y \mapsto \widehat{M}$

Fig. 7. Pat syntax extended for algorithmic typing

Although the mailbox calculus can rule out deadlocks using a dependency graph (de'Liguoro and Padovani 2018), dependency graphs are difficult to integrate with the richer structure of a programming language (see §2.2). We look forward to investigating inter-process deadlock detection in future work.

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, and does not scale naturally to a programming language. 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* approach (Erdweg et al. 2015): rather than taking a type environment as an *input* to the typechecking algorithm, we produce a type environment as an *output*. The intuition is that, read bottom-up, *splitting* an environment into two sub-environments is harder than *merging* two environments inferred from subexpressions. We also generate *pattern inclusion constraints* 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 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, which act as placeholders for parts of patterns and are generated during typechecking. We can then generate and solve *inclusion constraints* ϕ on patterns. Figure 7 shows the extended syntax used in the algorithmic system.

Our algorithmic type syntax uses a type annotation on **let**-expressions, used when establishing completeness with respect to the declarative system. As we will see, in the vast majority of cases this annotation is unnecessary, and such type annotations are not required by our implementation. The algorithmic type system also requires annotations on **guard** expressions; these *are* required by our implementation and we will discuss the necessity of these annotations when describing the algorithmic typing rules. Annotated computations $\widehat{M}, \widehat{N}$ replace the **let** $x = M$ **in** N construct from the declarative system with an annotated **let** $x\colon A = \widehat{M}$ **in** $\widehat{N}$ construct, and replace the **guard** $V\ \{\overrightarrow{G}\}$ construct from the declarative system with a **guard** $V : E\ \{\overrightarrow{\widehat{G}}\}$ expression that involves guarding on mailbox V and asserting that it has pattern E. Annotated guards $\widehat{G}$ are modified to include annotated expressions in their bodies.

We also introduce *open signatures* $\widehat{S}$ that allow message payload types to contain pattern variables; *annotated definitions* $\widehat{D}$ that allow function arguments and return types to contain pattern variables and where the function body is an annotated computation; and extend a program to include an open signature, annotated definitions, and an annotated body.

Constraints. An important challenge for the algorithmic type system is determining whether one pattern is *included* within another: for example, $\mathtt{m} \sqsubseteq \mathtt{m}^\star$. 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 $\gamma <: \delta$ to require that pattern γ is included in pattern δ.

Figure 8 shows the algorithmic type 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 corresponding definitions in the declarative system. Algorithmic subtyping is similar: a base type is a subtype of itself, and we check that one mailbox type is a subtype of another by checking that the capabilities match, and generating a contravariant constraint for a send type, and a covariant constraint for a receive type.

Algorithmic sequential merge. 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 sequential merge* 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 composed with the fresh pattern variable is included in the pattern δ.

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

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

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

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

Sequential merge $\boxed{\varsigma_1 \fatsemi \varsigma_2 \blacktriangleright \varsigma; \Phi}$ $\boxed{\tau_1 \fatsemi \tau_2 \blacktriangleright \sigma; \Phi}$

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

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

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

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

$$\frac{}{C \sqcap C \blacktriangleright C; \emptyset}$$

Fig. 8. Pat algorithmic type operations

As an example, applying the sequential merge to types $!\mathtt{m}$ and $?(\mathtt{n} \odot \mathtt{m})$ produces an input 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 gives rise to the expected combined type $?\mathtt{n}$.

Algorithmic branching merge. In the declarative type system, branching control flow requires that each branch is typable under the same type environment (using the T-Sub rule). The algorithmic type system instead generates constraints that ensure that each type is used consistently across branches using the *algorithmic branching 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 minimum usage annotation, *i.e.*, $\mathsf{min}(\bullet, \circ) = \bullet$. 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. Figure 9 shows how the algorithmic type combination operators can be extended to type environments.

The environment sequential merge operator $\Theta_1 \fatsemi \Theta_2 \blacktriangleright \Theta; \Phi$ concatenates Θ_1 and Θ_2, computing the sequential merge of any types for overlapping variables, and produces constraints Φ. The environment branching merge operator $\Theta_1 \sqcap \Theta_2 \blacktriangleright \Theta; \Phi$ computes the algorithmic branching merge of Θ_1 and Θ_2. If a variable is contained in both environments, then the merged type is used in the output environment. If the variable is only contained in one of the environments, then the result depends on the type. If the variable is only contained in one of the environments and has type $!\gamma^{\eta}$, then its type is changed to $!(\gamma \oplus \mathbb{1})^{\eta}$ to denote

Environment sequential merge $\boxed{\Theta_1 \fatsemi \Theta_2 \blacktriangleright \Theta; \Phi}$

$$\frac{}{\cdot \fatsemi \cdot \blacktriangleright \cdot; \emptyset} \qquad \frac{x \notin \mathsf{dom}(\Theta_2) \qquad \Theta_1 \fatsemi \Theta_2 \blacktriangleright \Theta; \Phi}{\Theta_1, x{:}\tau \fatsemi \Theta_2 \blacktriangleright \Theta, x{:}\tau; \Phi} \qquad \frac{x \notin \mathsf{dom}(\Theta_1) \qquad \Theta_1 \fatsemi \Theta_2 \blacktriangleright \Theta; \Phi}{\Theta_1 \fatsemi \Theta_2, x{:}\tau \blacktriangleright \Theta, x{:}\tau; \Phi}$$

$$\frac{\tau_1 \fatsemi \tau_2 \blacktriangleright \sigma; \Phi_1 \qquad \Theta_1 \fatsemi \Theta_2 \blacktriangleright \Theta; \Phi_2}{\Theta_1, x{:}\tau_1 \fatsemi \Theta_2, x{:}\tau_2 \blacktriangleright \Theta, x{:}\sigma; \Phi_1 \cup \Phi_2}$$

Environment branching merge $\boxed{\Theta_1 \sqcap \Theta_2 \blacktriangleright \Theta; \Phi}$

$$\frac{}{\cdot \sqcap \cdot \blacktriangleright \cdot; \emptyset} \qquad \frac{x \notin \mathsf{dom}(\Theta_2) \qquad \Theta_1 \sqcap \Theta_2 \blacktriangleright \Theta; \Phi}{\Theta_1, x{:}C \sqcap \Theta_2 \blacktriangleright \Theta, x{:}C; \Phi} \qquad \frac{x \notin \mathsf{dom}(\Theta_2) \qquad \Theta_1 \sqcap \Theta_2 \blacktriangleright \Theta; \Phi}{\Theta_1, x{:}!\gamma^{\eta} \sqcap \Theta_2 \blacktriangleright \Theta, x{:}!(\gamma \oplus \mathbb{1})^{\eta}; \Phi}$$

$$\frac{x \notin \mathsf{dom}(\Theta_1) \qquad \Theta_1 \sqcap \Theta_2 \blacktriangleright \Theta; \Phi}{\Theta_1 \sqcap \Theta_2, x{:}C \blacktriangleright \Theta, x{:}C; \Phi} \qquad \frac{x \notin \mathsf{dom}(\Theta_1) \qquad \Theta_1 \sqcap \Theta_2 \blacktriangleright \Theta; \Phi}{\Theta_1 \sqcap \Theta_2, x{:}!\gamma^{\eta} \blacktriangleright \Theta, x{:}!(\gamma \oplus \mathbb{1})^{\eta}; \Phi}$$

$$\frac{\tau_1 \sqcap \tau_2 \blacktriangleright \sigma; \Phi_1 \qquad \Theta_1 \sqcap \Theta_2 \blacktriangleright \Theta; \Phi_2}{\Theta_1, x{:}\tau_1 \sqcap \Theta_2, x{:}\tau_2 \blacktriangleright \Theta, x{:}\sigma; \Phi_1 \cup \Phi_2}$$

Disjoint combination $\boxed{\Theta_1 + \Theta_2 \blacktriangleright \Theta; \Phi}$

$$\frac{}{\cdot + \cdot \blacktriangleright \cdot; \emptyset} \qquad \frac{x \notin \mathsf{dom}(\Theta_2)}{\Theta_1 + \Theta_2 \blacktriangleright \Theta; \Phi \quad \Theta_1, x{:}\tau + \Theta_2 \blacktriangleright \Theta, x{:}\tau; \Phi} \qquad \frac{x \notin \mathsf{dom}(\Theta_1)}{\Theta_1 + \Theta_2 \blacktriangleright \Theta; \Phi \quad \Theta_1 + \Theta_2, x{:}\tau \blacktriangleright \Theta, x{:}\tau; \Phi} \qquad \frac{\Theta_1 + \Theta_2 \blacktriangleright \Theta; \Phi}{\Theta_1, x{:}C + \Theta_2, x{:}C \blacktriangleright \Theta, x{:}C; \Phi}$$

Fig. 9. Algorithmic environment combination

the fact that it may not be used. Note that this only applies to mailbox types with an *output* capability since mailbox types with an *input* capability must be treated linearly.

Disjoint environment combination combines two environments; if a variable is used in both environments then it must have an identical base type.

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 $\widehat{M} \Rightarrow_{\widehat{\mathcal{P}}} \tau \blacktriangleright \Theta; \Phi$, which can be read "synthesise type τ for term $\widehat{M}$ under program $\widehat{\mathcal{P}}$, producing type environment Θ and constraints Φ". Here, $\widehat{M}$ and $\widehat{\mathcal{P}}$ are inputs of the judgement, whereas τ, Θ, and Φ are outputs. The checking judgement $\widehat{M} \Leftarrow_{\widehat{\mathcal{P}}} \tau \blacktriangleright \Theta; \Phi$ can be read "check that term $\widehat{M}$ has type τ under program $\widehat{\mathcal{P}}$, producing type environment Θ and constraints Φ". Here, $\widehat{M}$, $\widehat{\mathcal{P}}$, and τ are inputs of the judgement, whereas Θ and Φ are outputs. As in the declarative system we omit the $\widehat{\mathcal{P}}$ 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 $\widehat{M}$ 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 usage annotations in the produced environment are masked as second-class since usage restrictions are process-local.

Constraint generation for programs and definitions $\boxed{\vdash \widehat{\mathcal{P}} \blacktriangleright \Phi}$ $\boxed{\vdash_{\widehat{\mathcal{P}}} \widehat{D} \blacktriangleright \Phi}$

$$\frac{\widehat{\mathcal{P}} = (\widehat{\mathcal{S}}, \overrightarrow{\widehat{D}}, \widehat{M}) \qquad (\vdash_{\widehat{\mathcal{P}}} \widehat{D}_i \blacktriangleright \Phi_i)_{i \in 1..n} \qquad \widehat{M} \Leftarrow \mathbf{1} \blacktriangleright \cdot; \Phi}{\vdash \widehat{\mathcal{P}} \blacktriangleright \Phi \cup \Phi_1 \cup \cdots \cup \Phi_n}$$

$$\frac{\widehat{M} \Leftarrow \sigma \blacktriangleright \Theta; \Phi_1 \qquad \mathsf{check}(\Theta, \overrightarrow{x}, \overrightarrow{\tau}) = \Phi_2 \qquad \Theta - \overrightarrow{x} = \cdot}{\vdash \mathbf{def}\ f(\overrightarrow{x : \tau}) : \sigma\ \{\widehat{M}\} \blacktriangleright \Phi_1 \cup \Phi_2}$$

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

TS-Const
$$\frac{c \text{ has base type } C}{c \Rightarrow C \blacktriangleright \cdot; \emptyset}$$

TS-New
$$\frac{}{\mathbf{new} \Rightarrow ?\mathbb{1}^{\bullet} \blacktriangleright \cdot; \emptyset}$$

TS-Spawn
$$\frac{\widehat{M} \Leftarrow \mathbf{1} \blacktriangleright \Theta; \Phi}{\mathbf{spawn}\ \widehat{M} \Rightarrow \mathbf{1} \blacktriangleright \lceil \Theta \rceil; \Phi}$$

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

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

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

TC-Var
$$\frac{}{x \Leftarrow \tau \blacktriangleright x : \tau; \emptyset}$$

TC-Let
$$\frac{\widehat{M} \Leftarrow A \blacktriangleright \Theta_1; \Phi_1 \qquad \widehat{N} \Leftarrow \tau \blacktriangleright \Theta_2; \Phi_2 \qquad \mathsf{returnable}(A) \qquad \mathsf{check}(\Theta_2, x, A) = \Phi_3 \qquad \Theta_1 \fatsemi (\Theta_2 - x) \blacktriangleright \Theta; \Phi_4}{\mathbf{let}\ x\colon A = \widehat{M}\ \mathbf{in}\ \widehat{N} \Leftarrow \tau \blacktriangleright \Theta; \Phi_1 \cup \cdots \cup \Phi_4}$$

TC-Guard
$$\frac{\{E\}\ \overrightarrow{\widehat{G}} \Leftarrow \tau \blacktriangleright \Theta_1; \Phi_1; F \qquad V \Leftarrow ?F^{\bullet} \blacktriangleright \Theta_2; \Phi_2 \qquad \Theta_1 + \Theta_2 \blacktriangleright \Theta; \Phi_3}{\mathbf{guard}\ V : E\ \{\overrightarrow{\widehat{G}}\} \Leftarrow \tau \blacktriangleright \Theta; \Phi_1 \cup \Phi_2 \cup \Phi_3 \cup \{E <: F\}}$$

TC-Fail
$$\frac{V \Leftarrow ?\mathbb{0}^{\bullet} \blacktriangleright \Theta; \Phi}{\mathbf{fail}\ V \Leftarrow \tau \blacktriangleright \Theta; \Phi}$$

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

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

$$\frac{x \notin \mathsf{dom}(\Theta) \qquad \mathsf{unr}(\tau) \blacktriangleright \Phi}{\mathsf{check}(\Theta, x, \tau) = \Phi} \qquad \frac{\sigma \leq \tau \blacktriangleright \Phi}{\mathsf{check}((\Theta, x : \tau), x, \sigma) = \Phi} \qquad \frac{(\mathsf{check}(\Theta, x_i, \tau_i) = \Phi_i)_{i \in 1..n}}{\mathsf{check}(\Theta, \overrightarrow{x}, \overrightarrow{\tau}) = \Phi_1 \cup \cdots \cup \Phi_n}$$

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

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 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 $\textbf{let}\, x\colon A = \widehat{M}\, \textbf{in}\, \widehat{N}$ has type τ: first, we check that $\widehat{M}$ has returnable type A, producing environment Θ_1 and constraints Φ_1. Next we check that the body $\widehat{N}$ has type τ, producing environment Θ_2 and constraints Φ_2. The next step is to check whether the type 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 $\textbf{guard}\, V : E\, \{\overrightarrow{\widehat{G}}\}$ has return type τ. First, the rule checks that the guard sequence $\overrightarrow{\widehat{G}}$ has type τ, producing environment Θ_1, constraint set Φ_1, and pattern F in pattern normal form (recall that a pattern E is in pattern normal form if it is a sum of $\mathbb{1}$ and $\mathtt{m} \odot F$ patterns, where F is equivalent to the pattern residual of E with respect to $\mathtt{m}$). Next, the rule checks that the mailbox name V has type $?F^{\bullet}$, producing environment Θ_2 and constraint set Φ_2. Finally, the rule calculates the disjoint combination of Θ_1 and Θ_2, producing final environment Θ and constraints Φ_3. The resulting constraint set is the union of the constraint sets produced by the premises, along with a constraint that the annotation E is a subpattern of the pattern F produced by the guards.

Rule TC-Fail allows $\textbf{fail}\, V$ to be checked at any type, provided that V has type $?\mathbb{0}^{\bullet}$.

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

Unannotated let expressions. Although our algorithmic core calculus uses an annotation on **let** expressions, this is unnecessary if the let-bound variable is used in the continuation $\widehat{N}$, or $\widehat{M}$ 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 $\widehat{M}$. Similarly, TC-LetNoAnn2 allows us to type a **let**-binding where x is *not* used in the continuation, as long as the type of $\widehat{M}$ is synthesisable and unrestricted.

$$\text{TC-LetNoAnn1}\quad \frac{\widehat{N} \Leftarrow \sigma \blacktriangleright \Theta_1, x : \tau;\, \Phi_1 \qquad \widehat{M} \Leftarrow \lfloor \tau \rfloor \blacktriangleright \Theta_2;\, \Phi_2 \qquad \Theta_2 \fatsemi \Theta_1 \blacktriangleright \Theta;\, \Phi_3}{\textbf{let}\, x = \widehat{M}\, \textbf{in}\, \widehat{N} \Leftarrow \sigma \blacktriangleright \Theta;\, \Phi_1 \cup \Phi_2 \cup \Phi_3}$$

$$\text{TC-LetNoAnn2}\quad \frac{\begin{array}{c}\widehat{N} \Leftarrow \sigma \blacktriangleright \Theta_1;\, \Phi_1 \qquad x \notin \mathsf{dom}(\Theta_1) \qquad \widehat{M} \Rightarrow \tau \blacktriangleright \Theta_2;\, \Phi_2 \\ \mathsf{returnable}(\tau) \qquad \mathsf{unr}(\tau) \blacktriangleright \Phi_3 \qquad \Theta_2 \fatsemi \Theta_1 \blacktriangleright \Theta;\, \Phi_4\end{array}}{\textbf{let}\, x = \widehat{M}\, \textbf{in}\, \widehat{N} \Leftarrow \sigma \blacktriangleright \Theta;\, \Phi_1 \cup \cdots \cup \Phi_4}$$

Constraint generation for guards $\boxed{\{E\}\ \overrightarrow{\widehat{G}} \Leftarrow_{\widehat{\mathcal{P}}} \tau \blacktriangleright \Theta; \Phi; F}$ $\boxed{\{E\}\ \widehat{G} \Leftarrow_{\widehat{\mathcal{P}}} \tau \blacktriangleright \Theta; \Phi; F}$

$$\textsc{TCG-Guards}\ \frac{(\{E\}\ \widehat{G}_i \Leftarrow \tau \blacktriangleright \Theta_i; \Phi_i; F_i)_{i \in 1..n} \qquad F = F_1 \oplus \cdots \oplus F_n \qquad \Theta_1 \sqcap \cdots \sqcap \Theta_n \blacktriangleright \Theta; \Phi}{\{E\}\ \overrightarrow{\widehat{G}} \Leftarrow \tau \blacktriangleright \Theta; \Phi \cup \Phi_1 \cup \cdots \cup \Phi_n; F}$$

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

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

Fig. 11. Pat algorithmic typing (guards)

We use the explicitly typed representation in the core language 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 $\{E\}\ \widehat{G} \Leftarrow \tau \blacktriangleright \Theta; \Phi; F$ can be read "Check that guard $\widehat{G}$ has type τ, producing environment Θ, 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 branching merge of all environments and the sum of all produced patterns. Rule TCG-Free checks that guard $\mathbf{free} \mapsto \widehat{M}$ has type τ by checking that $\widehat{M}$ has type τ; the guard produces pattern $\mathbb{1}$.

Finally, rule TCG-Recv checks that a receive guard $\mathbf{receive}\ \mathtt{m}(\overrightarrow{x})\ \mathbf{from}\ y \mapsto \widehat{M}$ has type τ. First, the rule checks that $\widehat{M}$ 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 the 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 that ensures $E \,/\, \mathtt{m}$ is included in γ, allowing us to produce the closed PNF literal $\mathtt{m} \odot (E \,/\, \mathtt{m})$. The annotation E on guard expressions is necessary because the pattern residual, used in the final constraint, is only defined on *closed* patterns.

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 pattern substitutions and solutions.

A *pattern substitution* Ξ is a mapping from pattern variables α to (fully defined) patterns E: applying Ξ to a pattern γ replaces each occurrence of a pattern variable α with $\Xi(\alpha)$. We extend application of pattern substitutions to types and environments. We write $\mathsf{pv}(\gamma)$ for the set of pattern variables in a pattern and extend it to types and environments.

Definition 19 (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 $\gamma <: \delta \in \Phi$, 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 $\mathbb{0}$.*

It is useful to define the notion of a *covering solution* to characterise a solution that resolves all pattern variables present in an algorithmic typing derivation.

Definition 20 (Covering solution)**.** *We say that a pattern substitution* Ξ *is a* covering solution *for a derivation* $\widehat{M} \Rightarrow_{\widehat{\mathcal{P}}} \tau \blacktriangleright \Theta; \Phi$ *or* $\widehat{M} \Leftarrow_{\widehat{\mathcal{P}}} \tau \blacktriangleright \Theta; \Phi$ *if given* $\vdash \widehat{\mathcal{P}} \blacktriangleright \Phi'$*, it is the case that* Ξ *is a usable solution for* $\Phi \cup \Phi'$ *such that* $\mathsf{pv}(\tau) \cup \mathsf{pv}(\widehat{\mathcal{P}}) \subseteq \mathsf{dom}(\Xi)$.

4.2.1 Algorithmic soundness

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. Proving this result involves establishing several auxiliary results on the soundness of the various type operators and type combination operators. Full proof details can be found in the extended version.

Properties of pattern variables and solutions. The first auxiliary result states that a solution for a set of constraints is also a solution for a subset of those constraints.

Lemma 5. *If* Ξ *is a solution for a constraint set* $\Phi_1 \cup \Phi_2$*, then* Ξ *is a solution for* Φ_1.

Proof. Since Ξ is a solution for $\Phi_1 \cup \Phi_2$, it follows that $\mathsf{pv}(\Phi_1 \cup \Phi_2) \subseteq \mathsf{dom}(\Xi)$. The result follows from the fact that $\mathsf{pv}(\Phi_1) \subseteq \mathsf{pv}(\Phi_1 \cup \Phi_2) \subseteq \mathsf{dom}(\Xi)$. □

We also need to reason about the provenance of pattern variables that appear in an inferred environment. Specifically, any pattern variable that appears in an inferred environment must either occur in the type of an expression, in the program, or in the constraint set.

Lemma 6. *If* $\widehat{M} \Rightarrow_{\widehat{\mathcal{P}}} \tau \blacktriangleright \Theta; \Phi$ *or* $\widehat{M} \Leftarrow_{\widehat{\mathcal{P}}} \tau \blacktriangleright \Theta; \Phi$*, then* $\mathsf{pv}(\Theta) \subseteq \mathsf{pv}(\tau) \cup \mathsf{pv}(\widehat{\mathcal{P}}) \cup \mathsf{pv}(\Phi)$.

Proof. By mutual induction on the two derivations, noting that whenever a pattern variable is introduced fresh, it is always added to the constraint set. □

Properties of type operations. Next, we need to show the relation between algorithmic and declarative versions of the various type operations. Given a usable solution of a constraint set, we can show the soundness of algorithmic subtyping.

Lemma 7. *If* $\tau \leq \sigma \blacktriangleright \Phi$ *and* Ξ *is a usable solution of* Φ *with* $\mathsf{pv}(\tau) \cup \mathsf{pv}(\sigma) \subseteq \mathsf{dom}(\Xi)$*, then* $\Xi(\tau) \leq \Xi(\sigma)$.

Proof. By case analysis on the derivation of $\tau \leq \sigma \blacktriangleright \Phi$. Base types are trivial, and the property follows for mailbox types from the definition of a usable solution. □

We can also show the soundness of the algorithmic unrestrictedness operation.

Lemma 8. *If* $\mathsf{unr}(\tau) \blacktriangleright \Phi$ *and* Ξ *is a usable solution of* Φ *with* $\mathsf{pv}(\tau) \subseteq \mathsf{dom}(\Xi)$*, then there exists some* A *such that* $\mathsf{un}(A)$ *and* $\Xi(\tau) \leq A$.

Proof. By case analysis on the derivation of $\mathsf{unr}(\tau) \blacktriangleright \Phi$, noting that cases are undefined for linear types, and that the result follows immediately for base types. The only interesting case is $\mathsf{unr}(!\,\gamma^{\eta}) \blacktriangleright \{\mathbb{1} <: \gamma\}$; since Ξ is a usable solution, we have that $\mathbb{1} \sqsubseteq \Xi(\gamma)$. Since $\bullet \leq \circ$ we can therefore show that $!\,(\Xi(\gamma))^{\eta} \leq !\,\mathbb{1}^{\circ}$ where $\mathsf{un}(!\,\mathbb{1}^{\circ})$ as required. □

Properties of type combination operations. Finally we need to show the soundness of the merge operators, which in both cases follows by case analysis on the respective derivations.

Lemma 9 (Soundness of algorithmic sequential merge)**.** *If* $\tau_1 \fatsemi \tau_2 \blacktriangleright \sigma; \Phi$ *and* Ξ *is a usable solution of* Φ *such that* $\mathsf{pv}(\tau_1) \cup \mathsf{pv}(\tau_2) \subseteq \mathsf{dom}(\Xi)$*, then there exist* $A \leq \Xi(\tau_1)$ *and* $B \leq \Xi(\tau_2)$ *such that* $A \triangleright B = \Xi(\sigma)$.

Lemma 10 (Soundness of algorithmic branching merge)**.** *If* $\tau_1 \sqcap \tau_2 \blacktriangleright \sigma; \Phi$ *and* Ξ *is a usable solution of* Φ *such that* $\mathsf{pv}(\tau_1) \cup \mathsf{pv}(\tau_2) \subseteq \mathsf{dom}(\Xi)$*, then* $\Xi(\sigma) \leq \Xi(\tau_1)$ *and* $\Xi(\sigma) \leq \Xi(\tau_2)$.

To relate annotated terms with unannotated terms in the declarative system, we define an *erasure* operator $\mathsf{erase}(\widehat{M}) = M$ on annotated computations that removes annotations on let binders and guards. The erasure operator is defined by the homomorphic extension of the following rules over computations and guards:

$$\begin{aligned} \mathsf{erase}(\mathbf{let}\ x\colon A = \widehat{M}\ \mathbf{in}\ \widehat{N}) &= \mathbf{let}\ x = \mathsf{erase}(\widehat{M})\ \mathbf{in}\ \mathsf{erase}(\widehat{N}) \\ \mathsf{erase}(\mathbf{guard}\, V : E\ \{\overrightarrow{\widehat{G}}\}) &= \mathbf{guard}\ V\ \{\overrightarrow{\mathsf{erase}(\widehat{G})}\} \end{aligned}$$

We also extend the erasure operator to definitions and programs:

$$\begin{aligned} \mathsf{erase}(\mathbf{def}\ f(\overrightarrow{x:\tau})\colon \sigma\ \{\widehat{M}\}) &= \mathbf{def}\ f(\overrightarrow{x:\tau})\colon \sigma\ \{\mathsf{erase}(\widehat{M})\} \\ \mathsf{erase}((\widehat{\mathcal{S}}, \overrightarrow{\widehat{D}}, \widehat{M})) &= (\widehat{\mathcal{S}}, \mathsf{erase}(\overrightarrow{\widehat{D}}), \mathsf{erase}(\widehat{M})) \end{aligned}$$

With these auxiliary results defined, we can show that the algorithmic type system is sound with respect to the declarative type system, meaning that our algorithmic type system will never accept ill-typed terms.

Theorem 3 (Algorithmic soundness)**.**

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

The result follows from a generalised version of algorithmic soundness on all of the typing judgements (for values, computations, guards, and guard sequences), which is established by mutual induction.

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. Recall that by the definition in §3, program $\mathcal{P}$ is *closed*, *i.e.*, no definitions or message payloads contain pattern variables.

Again, the proof of algorithmic completeness requires several auxiliary lemmas. Full proofs can be found in the extended version.

Closed and satisfiable constraint sets. We first define *closed* and *satisfiable* constraint sets.

Definition 21 (Closed and satisfiable constraint sets)**.** *A constraint set* Φ *is* closed *if* $\mathsf{pv}(\Phi) = \emptyset$*. A closed constraint set* Φ *is* satisfiable *if the empty solution is a solution for* Φ *(i.e.,* $\Phi = (E_i <: F_i)_{i \in 1..n}$ *and* $(E_i \sqsubseteq F_i)_{i \in 1..n}$*).*

Checkability of values and synthesisable terms. Values are checkable, without creating any additional constraints.

Lemma 11. *If* $\Gamma \vdash V : A$*, then there exists some* Γ' *such that* $\Gamma \leq \Gamma'$ *and* $V \Leftarrow A \blacktriangleright \Gamma'; \emptyset$.

Proof. By induction on the derivation of $\Gamma \vdash V : A$. T-Var and T-Const follow immediately, and T-Sub follows from the IH and the transitivity of subtyping. □

Next, any synthesisable term is checkable with the same type, without needing to introduce any additional constraints.

Lemma 12. *If* $\widehat{M} \Rightarrow \tau \blacktriangleright \Theta; \Phi$*, then* $\widehat{M} \Leftarrow \tau \blacktriangleright \Theta; \Phi$.

Proof. Follows from the definition of TC-Sub, noting that the subtyping constraint is instantiated as $\tau \leq \tau \blacktriangleright \emptyset$. □

Properties of type operations. As with algorithmic soundness, we need various lemmas that show the relationship between the algorithmic and declarative versions of the operations on types and environments. We begin by showing three useful lemmas relating to subtyping.

First, algorithmic subtyping on closed types is transitive.

Lemma 13 (Transitivity of algorithmic subtyping)**.** *If* $A \leq A' \blacktriangleright \Phi$ *where* Φ *is satisfiable, and* $A' \leq B$*, then* $A \leq B \blacktriangleright \Phi'$ *and* Φ' *is satisfiable.*

Proof. By case analysis on the derivation of $A \leq A' \blacktriangleright \Phi$ and the transitivity of pattern inclusion. □

Second, it is useful to show that if a term is checkable at some type A, then it is also checkable at some supertype B.

Lemma 14 (Checkability at a supertype)**.** *If* $\widehat{M} \Leftarrow_{\mathcal{P}} A \blacktriangleright \Theta; \Phi$ *where* Ξ *is a usable solution of* Φ*, and* $A \leq B$*, then there exist* Θ', Φ' *such that* $\widehat{M} \Leftarrow_{\mathcal{P}} B \blacktriangleright \Theta'; \Phi'$ *where* Ξ *is a usable solution of* Φ' *and* $\Xi(\Theta) \preccurlyeq \Xi(\Theta')$.

Proof. Follows from a more general result established by mutual induction, making essential use of Lemma 13. □

The completeness of subtyping follows by case analysis on the derivation of $\Xi(\tau) \leq \Xi(\sigma)$ and the definition of pattern inclusion.

Lemma 15 (Completeness of subtyping)**.** *Given a pattern substitution* Ξ *such that* $\Xi(\tau) \leq \Xi(\sigma)$*, it follows that* $\tau \leq \sigma \blacktriangleright \Phi$ *and* Ξ *is a usable solution of* Φ.

The completeness of the check meta-function follows from case analysis on whether the given variable is contained within an environment, with appeal to Lemma 15.

Lemma 16 (Completeness of check)**.** *Given a pattern substitution* Ξ *such that* $\Xi(\Theta, x : \tau) \leq \Xi(\Theta')$*, it follows that* $\mathsf{check}(\Theta', x, \tau) = \Phi$ *and* Ξ *is a usable solution of* Φ.

Properties of type combination operations. The final set of lemmas concentrates on the completeness of the type combination operators. Both proofs follow by case analysis on the respective declarative derivations and appeal to the underlying pattern semantics.

Lemma 17 (Completeness of algorithmic sequential merge)**.** *If* $A_1 \triangleright A_2 = B$ *where* $A_1 \leq \Xi_1(\tau_1)$ *and* $A_2 \leq \Xi_2(\tau_2)$ *for pattern substitutions* Ξ_1, Ξ_2 *such that* $\mathsf{dom}(\Xi_1) \cap \mathsf{dom}(\Xi_2) = \emptyset$*, then there exist* σ, Φ *such that* $\tau_1 \fatsemi \tau_2 \blacktriangleright \sigma; \Phi$*, and there exists a usable solution* $\Xi \supseteq \Xi_1 \cup \Xi_2$ *of* Φ *such that* $B \leq \Xi(\sigma)$*.*

Lemma 18 (Completeness of algorithmic branching merge)**.** *If* $A \leq \Xi_1(\tau_1)$ *and* $A \leq \Xi_2(\tau_2)$ *for pattern substitutions* Ξ_1, Ξ_2 *such that* $\mathsf{dom}(\Xi_1) \cap \mathsf{dom}(\Xi_2) = \emptyset$*, then there exist* σ, Φ *such that* $\tau_1 \sqcap \tau_2 \blacktriangleright \sigma; \Phi$ *and there exists a usable solution* $\Xi \supseteq \Xi_1 \cup \Xi_2$ *of* Φ *such that* $A \leq \Xi(\sigma)$*.*

Algorithmic completeness. With these intermediate results in hand, we can state the completeness of the algorithmic type system with respect to the declarative type system.

To relate unannotated terms that are typable in the declarative system to annotated terms required for the algorithmic type system, we introduce type-directed annotation rules. The interesting rules are those for **let**-bindings and guard expressions, which use information from the declarative type system to add appropriate annotations. The remaining rules (detailed in the extended version) are defined similarly.

Type-directed annotation $\boxed{\Gamma \vdash M : A \overset{\mathsf{ann}}{\hookrightarrow} \widehat{M}}$

$$\frac{\Gamma_1 \vdash M : A \overset{\mathsf{ann}}{\hookrightarrow} \widehat{M} \qquad \mathsf{returnable}(A) \qquad \Gamma_2, x : A \vdash N : B \overset{\mathsf{ann}}{\hookrightarrow} \widehat{N}}{\Gamma_1 \triangleright \Gamma_2 \vdash \mathbf{let}\, x = M\, \mathbf{in}\, N : B \overset{\mathsf{ann}}{\hookrightarrow} \mathbf{let}\, x \colon A = \widehat{M}\, \mathbf{in}\, \widehat{N}}$$

$$\frac{\Gamma_1 \vdash V : ?E^{\bullet} \qquad \{E\}\ \Gamma_2 \vdash \overrightarrow{G} : A \overset{\mathsf{ann}}{\hookrightarrow} \overrightarrow{\widehat{G}} \qquad \vDash E}{\Gamma_1 + \Gamma_2 \vdash \mathbf{guard}\, V\, \{\overrightarrow{G}\} : A \overset{\mathsf{ann}}{\hookrightarrow} \mathbf{guard}\, V : E\, \{\overrightarrow{\widehat{G}}\}}$$

Finally, we can state our algorithmic completeness result.

Theorem 4 (Algorithmic completeness)**.** *If* $\vdash \mathcal{P}$ *where* $\Gamma \vdash_{\mathcal{P}} M : A \overset{\mathsf{ann}}{\hookrightarrow} \widehat{M}$*, then there exist some* Θ, Φ *and usable solution* Ξ *of* Φ *such that* $\widehat{M} \Leftarrow_{\mathcal{P}} A \blacktriangleright \Theta; \Phi$ *where* $\Gamma \leq \Xi(\Theta)$*.*

The proof in the extended version is again by mutual induction using a generalised statement showing completeness of typing values, computations, guards, and guard sequences.

Although the completeness result is aided by the explicit annotations on let bindings, an unannotated **let** binding $\mathbf{let}\, x = \widehat{M}\, \mathbf{in}\, \widehat{N}$ is also typable by the algorithmic type system if either x occurs free in $\widehat{N}$, or the type of $\widehat{M}$ is synthesisable. In practice this encompasses both base types and linear usages of mailbox types, covering the vast majority of use cases.

4.3 Constraint solving

Constraint solving is covered in depth by Padovani (2018c), and is not a contribution of this work. However, as an informal overview, constraint solving proceeds as follows:

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 first group all pattern bounds using pattern disjunction, *e.g.*, a constraint set $\{\gamma <: \alpha, \delta <: \alpha\}$ 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 $(\gamma_i <: \alpha_i)_{i \in 1..n}$: there exists a closed-form 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 are precisely those defined by Presburger formulae (Ginsburg and Spanier 1966). To determine whether each constraint holds, we use the quantifier elimination pass and quantifier-free linear integer arithmetic solver supported by the Z3 tool (de Moura and Bjørner 2008).

5 Extensions

In this section we show how we can extend the base language to include additional language features, and describe how this impacts typechecking.

It is straightforward to extend Pat with product and sum types, and building on these allows us to introduce list types, which are a stepping stone towards extending Pat with general recursive data types in the future.

We then show how to extend Pat with first-class functions, and *interfaces* that describe the set of messages a mailbox is allowed to receive, which in turn increase the precision of typechecking and allow finer-grained alias control. The latter two extensions require contextual typing information prior to constraint generation.

5.1 Data types

5.1.1 Product types

An advantage of adding product types is that we can avoid nested **guard** expressions, by allowing a **guard** expression to return a pair of a received value and an updated mailbox name. Consider the following two expressions: the term on the left receives two integers and returns their sum using nested **guard** expressions, whereas the term on the right avoids nesting by returning a pair of the received value and the updated mailbox.

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

```
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
}
```

This pattern can avoid deeply nested guard blocks when a process wants to make several receives in a row. Additionally, recall from §3 that **receive** clauses use a conservative check to rule out communication-based aliasing, where either all payloads of a message must be base types, or all variables free in the body of the **receive** clause must be base types. Returning received values along with the continuation of the mailbox is a useful tool to increase expressiveness in the presence of this restriction.

Formalism. Figure 12 shows how to extend Pat with product types. The rules for constructing and deconstructing pairs (T-Pair and T-LetPair) are standard aside from the condition that

Additional syntax

Usage-annotated types	A, B	::=	$\cdots \mid A \times B$
Values	V, W	::=	$\cdots \mid (V, W)$
Computations	M, N	::=	$\cdots \mid \textbf{let}\ (x, y) = V\ \textbf{in}\ M$
Annotated computations	$\widehat{M}, \widehat{N}$	::=	$\cdots \mid \textbf{let}\ (x, y) \colon (A \times B) = V\ \textbf{in}\ \widehat{M}$

Additional declarative typing rules $\boxed{\Gamma \vdash V : A}$ $\boxed{\Gamma \vdash M : A}$

T-Pair

$$\frac{\Gamma_1 \vdash V : A \qquad \Gamma_2 \vdash W : B \qquad \mathsf{returnable}(A) \qquad \mathsf{returnable}(B)}{\Gamma_1 + \Gamma_2 \vdash (V, W) : A \times B}$$

T-LetPair

$$\frac{\Gamma_1 \vdash V : A_1 \times A_2 \qquad \Gamma_2, x : A_1, y : A_2 \vdash M : B}{\Gamma_1 + \Gamma_2 \vdash \textbf{let}\ (x, y) = V\ \textbf{in}\ M : B}$$

Additional declarative operations on types $\boxed{\mathsf{un}(A)}$ $\boxed{\mathsf{returnable}(A)}$ $\boxed{A \leq B}$ $\boxed{A \triangleright B}$

$$\frac{\mathsf{un}(A) \qquad \mathsf{un}(B)}{\mathsf{un}(A \times B)} \qquad \mathsf{returnable}(A \times B) \qquad \frac{A_1 \leq B_1 \qquad A_2 \leq B_2}{(A_1 \times A_2) \leq (B_1 \times B_2)}$$

$$(A_1 \times A_2) \triangleright (B_1 \times B_2) = (A_1 \triangleright B_1) \times (A_2 \triangleright B_2)$$

Additional algorithmic typing rules $\boxed{\widehat{M} \Rightarrow_{\widehat{\mathcal{P}}} \tau \blacktriangleright \Theta; \Phi}$ $\boxed{\widehat{M} \Leftarrow_{\widehat{\mathcal{P}}} \tau \blacktriangleright \Theta; \Phi}$

TC-Pair

$$\frac{V \Leftarrow \tau \blacktriangleright \Theta_1; \Phi_1 \qquad W \Leftarrow \sigma \blacktriangleright \Theta_2; \Phi_2 \qquad \mathsf{returnable}(\tau) \qquad \mathsf{returnable}(\sigma) \qquad \Theta_1 + \Theta_2 \blacktriangleright \Theta; \Phi_3}{(V, W) \Leftarrow \tau \times \sigma \blacktriangleright \Theta; \Phi_1 \cup \Phi_2 \cup \Phi_3}$$

TC-LetPair

$$\frac{V \Leftarrow A \times B \blacktriangleright \Theta_1; \Phi_1 \qquad \widehat{M} \Leftarrow \tau \blacktriangleright \Theta_2; \Phi_2 \qquad \mathsf{check}(\Theta_2, x, A) = \Phi_3 \qquad \mathsf{check}(\Theta_2, y, B) = \Phi_4 \qquad \Theta_1 + ((\Theta_2 - x) - y) \blacktriangleright \Theta; \Phi_5}{\textbf{let}\ (x, y) \colon (A \times B) = V\ \textbf{in}\ \widehat{M} \Leftarrow \tau \blacktriangleright \Theta; \Phi_1 \cup \cdots \cup \Phi_5}$$

TC-LetPairNoAnn

$$\frac{\widehat{M} \Leftarrow \sigma \blacktriangleright \Theta_1, x : \tau_1, y : \tau_2; \Phi_1 \qquad V \Leftarrow \tau_1 \times \tau_2 \blacktriangleright \Theta_2; \Phi_2 \qquad \Theta_1 + \Theta_2 \blacktriangleright \Theta; \Phi_3}{\textbf{let}\ (x, y) = V\ \textbf{in}\ \widehat{M} \Leftarrow \sigma \blacktriangleright \Theta; \Phi_1 \cup \Phi_2 \cup \Phi_3}$$

Fig. 12. Extension of Pat with product types

the types of both values used to construct the pair are returnable. If we were to lift this restriction then we would be able to violate the quasi-linearity conditions, for example by using pair construction and deconstruction to subvert the condition that only the last lexical occurrence of a variable can be returnable. We prefer T-LetPair to individual projection functions due to the possibility that one of the pair components may be linear, though we can use the usual syntactic sugar (*e.g.*, $\textbf{fst}\ V \triangleq \textbf{let}\ (x, y) = V\ \textbf{in}\ x$ for fresh x, y). Since product types can only consist of returnable types, they cannot be used to replace n-ary argument sequences in function definitions and **receive** clauses in full generality.

As usual, subtyping for product types is covariant in the component types, and a product type is unrestricted if both of its component types are unrestricted. We also extend type combination to handle products through pointwise combination of the product's component types. To see why this is necessary, consider the following example:

```
def sendToBoth(p : (!M1 × !M2)): 1 {
  let (mb1, mb2) = p in
  mb1 !M1();
  mb2 !M2()
}
def receiveAndFree(p : (?(M1 ⊙ 𝟙) × ?(M2 ⊙ 𝟙))): 1 {
  let (mb1, mb2) = p in
  guard mb1 {
    receive M1() from mb1 ↦ free mb1
  };
  guard mb2 {
    receive M2() from mb2 ↦ free mb2
  }
}
```

```
let mb1 = new in
let mb2 = new in
let p = (mb1, mb2) in
spawn (sendToBoth(p));
receiveAndFree(p)
```

Here we create two mailboxes mb_1 and mb_2 and package them into a pair p. We spawn a process that calls sendToBoth: this sends **M1** to the first component of the pair and **M2** to the second component of the pair, and therefore uses p at type $!\mathbf{M1} \times !\mathbf{M2}$. The main process subsequently calls receiveAndFree, which uses p at type $(?(\mathbf{M1} \odot \mathbb{1}) \times ?(\mathbf{M2} \odot \mathbb{1}))$. Component-wise type combination gives p the combined type of $(?\mathbb{1} \times ?\mathbb{1})$ as needed.

As for the algorithmic rules, pair construction (TC-Pair) checks that both components have the given types, and that both given types are returnable. Environment combination and constraints are handled as usual. Deconstructing the pair in general requires an annotation (TC-LetPair); as with the let rule, we check that the pair has the given annotation and that the types inferred in the environment of the continuation are consistent with the annotation. If both components are used within the continuation then we can omit the annotation (TC-LetPairNoAnn): the rule first checks that the continuation has the given type, and inspects the resulting environment to construct the product type used for checking V. We omit the algorithmic type operations, which are the usual adaptations of the declarative operations.

5.1.2 *Sum types*

It is also useful to include sum types in order to express multiple ways of constructing data. The main principles are the same as for product types.

Formalism. Figure 13 shows how to extend Pat with sum types, which largely follows the development for product types. Again, the declarative rules are unremarkable except for the requirement that sum components must be returnable in the introduction rules. Subtyping and type combination are defined component-wise, and a sum type is unrestricted if both of its components are unrestricted. As for the algorithmic rules, sum injections are checking cases; similar to the product rules we must ensure that the constituent types are both returnable. We also have two separate rules for case expressions that allow annotations to be elided if both x and y are used within continuations M and N respectively.

5.1.3 *List types*

We can apply a similar approach to extend Pat to support inductively defined lists.

Lists are useful as they allow us to encode patterns such as broadcasting a message to a number of clients. For example, the following code (adapted from the K-Fork benchmark described in §6.2) broadcasts a request to a list of actor references:

Additional syntax

Usage-annotated types	A, B	::=	$\cdots \mid A + B$
Values	V, W	::=	$\cdots \mid \mathbf{inl}\ V \mid \mathbf{inr}\ V$
Computations	M, N	::=	$\cdots \mid \mathbf{case}\ V\ \mathbf{of}\ \{\mathbf{inl}\ x \mapsto M; \mathbf{inr}\ y \mapsto N\}$
Annotated computations	$\widehat{M}, \widehat{N}$	::=	$\cdots \mid \mathbf{case}\ V\ \mathbf{of}\ \{\mathbf{inl}\ x : A_1 \mapsto \widehat{M}; \mathbf{inr}\ y : A_2 \mapsto \widehat{N}\}$

Additional declarative typing rules $\boxed{\Gamma \vdash V : A}$ $\boxed{\Gamma \vdash M : A}$

$$\textsc{T-Inl}\ \frac{\Gamma \vdash V : A \qquad \mathsf{returnable}(A) \qquad \mathsf{returnable}(B)}{\Gamma \vdash \mathbf{inl}\ V : A + B} \qquad \textsc{T-Inr}\ \frac{\Gamma \vdash V : B \qquad \mathsf{returnable}(A) \qquad \mathsf{returnable}(B)}{\Gamma \vdash \mathbf{inr}\ V : A + B}$$

$$\textsc{T-Case}\ \frac{\Gamma_1 \vdash V : A_1 + A_2 \qquad \Gamma_2, x : A_1 \vdash M : B \qquad \Gamma_2, y : A_2 \vdash N : B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{case}\ V\ \mathbf{of}\ \{\mathbf{inl}\ x \mapsto M; \mathbf{inr}\ y \mapsto N\} : B}$$

Additional declarative operations on types $\boxed{\mathsf{un}(A)}$ $\boxed{\mathsf{returnable}(A)}$ $\boxed{A \leq B}$ $\boxed{A \triangleright B}$

$$\frac{\mathsf{un}(A) \qquad \mathsf{un}(B)}{\mathsf{un}(A + B)} \qquad \mathsf{returnable}(A + B) \qquad \frac{A_1 \leq B_1 \qquad A_2 \leq B_2}{(A_1 + A_2) \leq (B_1 + B_2)}$$

$$(A_1 + A_2) \triangleright (B_1 + B_2) = (A_1 \triangleright B_1) + (A_2 \triangleright B_2)$$

Additional algorithmic typing rules $\boxed{\widehat{M} \Rightarrow_{\widehat{\varphi}} \tau \blacktriangleright \Theta; \Phi}$ $\boxed{\widehat{M} \Leftarrow_{\widehat{\varphi}} \tau \blacktriangleright \Theta; \Phi}$

$$\textsc{TC-Inl}\ \frac{V \Leftarrow \tau \blacktriangleright \Theta; \Phi \qquad \mathsf{returnable}(\tau) \qquad \mathsf{returnable}(\sigma)}{\mathbf{inl}\ V \Leftarrow \tau + \sigma \blacktriangleright \Theta; \Phi} \qquad \textsc{TC-Inr}\ \frac{V \Leftarrow \sigma \blacktriangleright \Theta; \Phi \qquad \mathsf{returnable}(\tau) \qquad \mathsf{returnable}(\sigma)}{\mathbf{inr}\ V \Leftarrow \tau + \sigma \blacktriangleright \Theta; \Phi}$$

$$\textsc{TC-Case}\ \frac{\begin{array}{c} V \Leftarrow A_1 + A_2 \blacktriangleright \Theta_1; \Phi_1 \qquad \widehat{M} \Leftarrow \tau \blacktriangleright \Theta_2; \Phi_2 \qquad \widehat{N} \Leftarrow \tau \blacktriangleright \Theta_3; \Phi_3 \qquad \mathsf{check}(\Theta_2, x, A_1) = \Phi_4 \\ \mathsf{check}(\Theta_3, y, A_2) = \Phi_5 \qquad (\Theta_2 - x) \sqcap (\Theta_3 - y) \blacktriangleright \Theta_4; \Phi_6 \qquad \Theta_1 + \Theta_4 \blacktriangleright \Theta; \Phi_7 \end{array}}{\mathbf{case}\ V\ \mathbf{of}\ \{\mathbf{inl}\ x : A_1 \mapsto \widehat{M}; \mathbf{inr}\ y : A_2 \mapsto \widehat{N}\} \Leftarrow \tau \blacktriangleright \Theta; \Phi_1 \cup \cdots \cup \Phi_7}$$

$$\textsc{TC-CaseNoAnn}\ \frac{\begin{array}{c} \widehat{M} \Leftarrow \sigma \blacktriangleright \Theta_1, x : \tau_1; \Phi_1 \qquad \widehat{N} \Leftarrow \sigma \blacktriangleright \Theta_2, y : \tau_2; \Phi_2 \\ V \Leftarrow \tau_1 + \tau_2 \blacktriangleright \Theta_3; \Phi_3 \qquad \Theta_1 \sqcap \Theta_2 \blacktriangleright \Theta_4; \Phi_4 \qquad \Theta_3 + \Theta_4 \blacktriangleright \Theta; \Phi_5 \end{array}}{\mathbf{case}\ V\ \mathbf{of}\ \{\mathbf{inl}\ x \mapsto \widehat{M}; \mathbf{inr}\ y \mapsto \widehat{N}\} \Leftarrow \sigma \blacktriangleright \Theta; \Phi_1 \cup \cdots \cup \Phi_5}$$

Fig. 13. Extension of Pat with sum types

```
def broadcast(actorMbs : List(!Request)): 1 {
  caseL (actorMbs : List(!Request)) of {
    nil ↦ ()
    mb :: mbs ↦ mb!Request(); broadcast(mbs)
  }
}
```

Formalism. Figure 14 shows how to extend Pat with list types, which follows the development for product and sum types presented above. As one might expect, constructing a list is similar

Additional syntax

Usage-annotated types	A, B	::=	$\cdots \mid \mathsf{List}(A)$
Values	V, W	::=	$\cdots \mid \mathbf{nil} \mid V :: W$
Computations	M, N	::=	$\cdots \mid \mathbf{caseL}\ V\ \mathbf{of}\ \{\mathbf{nil} \mapsto M; y :: ys \mapsto N\}$
Annotated computations	$\widehat{M}, \widehat{N}$	::=	$\cdots \mid \mathbf{caseL}\ (V : \mathsf{List}(A))\ \mathbf{of}\ \{\mathbf{nil} \mapsto \widehat{M}; y :: ys \mapsto \widehat{N}\}$

Additional declarative typing rules $\boxed{\Gamma \vdash V : A}$ $\boxed{\Gamma \vdash M : A}$

$$\text{T-Nil}\ \frac{\mathsf{returnable}(A)}{\cdot \vdash \mathbf{nil} : \mathsf{List}(A)} \qquad \text{T-Cons}\ \frac{\Gamma_1 \vdash V : A \qquad \Gamma_2 \vdash W : \mathsf{List}(A) \qquad \mathsf{returnable}(A)}{\Gamma_1 + \Gamma_2 \vdash V :: W : \mathsf{List}(A)}$$

$$\text{T-CaseL}\ \frac{\Gamma_1 \vdash V : \mathsf{List}(A) \qquad \Gamma_2 \vdash M : B \qquad \Gamma_2, y : A, ys : \mathsf{List}(A) \vdash N : B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{caseL}\ V\ \mathbf{of}\ \{\mathbf{nil} \mapsto M; y :: ys \mapsto N\} : B}$$

Additional declarative operations on types $\boxed{\mathsf{un}(A)}$ $\boxed{\mathsf{returnable}(A)}$ $\boxed{A \leq B}$ $\boxed{A \triangleright B}$

$$\frac{\mathsf{un}(A)}{\mathsf{un}(\mathsf{List}(A))} \qquad \mathsf{returnable}(\mathsf{List}(A)) \qquad \frac{A \leq B}{\mathsf{List}(A) \leq \mathsf{List}(B)} \qquad (\mathsf{List}(A) \triangleright \mathsf{List}(B)) = \mathsf{List}(A \triangleright B)$$

Additional algorithmic typing rules $\boxed{\widehat{M} \Rightarrow_{\widehat{\mathcal{P}}} \tau \blacktriangleright \Theta; \Phi}$ $\boxed{\widehat{M} \Leftarrow_{\widehat{\mathcal{P}}} \tau \blacktriangleright \Theta; \Phi}$

$$\text{TC-Nil}\ \frac{\mathsf{returnable}(\tau)}{\mathbf{nil} \Leftarrow \mathsf{List}(\tau) \blacktriangleright \cdot; \emptyset} \qquad \text{TC-Cons}\ \frac{V \Leftarrow \tau \blacktriangleright \Theta_1; \Phi_1 \qquad W \Leftarrow \mathsf{List}(\tau) \blacktriangleright \Theta_2; \Phi_2 \qquad \mathsf{returnable}(\tau) \qquad \Theta_1 + \Theta_2 \blacktriangleright \Theta; \Phi_3}{V :: W \Leftarrow \mathsf{List}(\tau) \blacktriangleright \Theta; \Phi_1 \cup \Phi_2 \cup \Phi_3}$$

$$\text{TC-CaseL}\ \frac{\begin{array}{l} V \Leftarrow \mathsf{List}(A) \blacktriangleright \Theta_1; \Phi_1 \qquad \widehat{M} \Leftarrow \tau \blacktriangleright \Theta_2; \Phi_2 \qquad \widehat{N} \Leftarrow \tau \blacktriangleright \Theta_3; \Phi_3 \qquad \mathsf{check}(\Theta_3, y, A) = \Phi_4 \\ \mathsf{check}(\Theta_3, ys, \mathsf{List}(A)) = \Phi_5 \qquad \Theta_2 \sqcap ((\Theta_3 - y) - ys) \blacktriangleright \Theta_4; \Phi_6 \qquad \Theta_1 + \Theta_4 \blacktriangleright \Theta; \Phi_7 \end{array}}{\mathbf{caseL}\ (V : \mathsf{List}(A))\ \mathbf{of}\ \{\mathbf{nil} \mapsto \widehat{M}; y :: ys \mapsto \widehat{N}\} \Leftarrow \tau \blacktriangleright \Theta; \Phi_1 \cup \cdots \cup \Phi_7}$$

$$\text{TC-CaseLNoAnn}\ \frac{\begin{array}{l} \widehat{M} \Leftarrow \sigma \blacktriangleright \Theta_1; \Phi_1 \qquad \widehat{N} \Leftarrow \sigma \blacktriangleright \Theta_2, y : \tau, ys : \mathsf{List}(\tau); \Phi_2 \\ V \Leftarrow \mathsf{List}(\tau) \blacktriangleright \Theta_3; \Phi_3 \qquad \Theta_1 \sqcap \Theta_2 \blacktriangleright \Theta_4; \Phi_4 \qquad \Theta_3 + \Theta_4 \blacktriangleright \Theta; \Phi_5 \end{array}}{\mathbf{caseL}\ V\ \mathbf{of}\ \{\mathbf{nil} \mapsto \widehat{M}; y :: ys \mapsto \widehat{N}\} \Leftarrow \sigma \blacktriangleright \Theta; \Phi_1 \cup \cdots \cup \Phi_5}$$

Fig. 14. Extension of Pat with list types

to constructing a product after injecting into a sum, while pattern matching on a list is similar to case matching on a sum followed by deconstructing a product.

The declarative rules are unremarkable aside from the type of values in the list needing to be returnable in the introduction rules. Since sums and lists are currently the only types in Pat that can be pattern matched against, we use different syntax for each: **case** and **caseL** to avoid overloading. A future extension introducing general recursive types would aim to unify these and allow for general pattern matching against the constructors of any data type, but this extension is outside the scope of the current work. Note that we need the $\mathsf{returnable}(A)$ condition on *both* T-Cons *and* T-Nil, rather than just T-Nil, to avoid issues with subtyping.

In the algorithmic rules, rule TC-Nil requires that the type of list elements is returnable. In TC-Cons, environment combination and constraints are handled as in the analogous product rule. Case expressions, like sums, have two separate rules allowing the annotation to be elided if both y and ys are used in the continuation N.

5.2 Towards more liberal data types

We have required that the data contained within each of the described data types must be returnable. This ensures that we cannot deconstruct a data type containing a second-class name and unpackage it later, thus breaking the lexical scoping requirements of quasi-linearity. However, this approach can be restrictive.

A potential solution is to allow the construction of data types that may contain names with second-class types, but only to allow them to be deconstructed in a situation where unsafe aliasing cannot occur: namely where the continuation M of a **let** $(x, y) = V$ **in** M construct or the relevant continuations of a **case** construct do not close over any mailbox types (or, as we will see in §5.4, do not close over any mailbox types that might lead to aliasing). We can formalise the declarative rules for product types as follows; the corresponding rules for sums and lists are similar.

$$\frac{\Gamma_1 \vdash V : A \qquad \Gamma_2 \vdash W : B}{\Gamma_1 + \Gamma_2 \vdash (V, W) : A \times B} \qquad \frac{\Gamma_1 \vdash V : A_1 \times A_2 \qquad \Gamma_2, x : A_1, y : A_2 \vdash M : B \qquad (\mathsf{returnable}(A_1) \wedge \mathsf{returnable}(A_2)) \vee \mathsf{base}(\Gamma_2)}{\Gamma_1 + \Gamma_2 \vdash \textbf{let}\ (x, y) = V\ \textbf{in}\ M : B}$$

This approach has the advantage that it rules out problematic cases that introduce unsafe aliasing, where we have two static names for the same underlying runtime name. Consider the following example (assuming x is bound elsewhere):

```
let mb = new in
let pair = (mb, 1) in
let (a, b) = pair in
x ! m(a);
guard mb {
  receive m(y) from mb ↦ free mb
}
```

This code snippet packages *mb* (with type $!\mathbb{1}^\circ$) into a pair, then uses pair deconstruction to alias the first component of the pair to a, and would be ruled out as mailbox names x and *mb* are free in the continuation of the pair deconstruction construct.

However, the approach is not sound according to our declarative rules due to quasi-linearity, and does not rule out self-deadlocks. For example, the following code would be well-typed even though it introduces a self-deadlock:

```
let mb = new in
let pair = (mb, 1) in
guard mb {
  receive m() from mb ↦ free mb
};
let (x, y) = pair in
x ! m()
```

Additional syntax

Linearity annotations	$\diamond$	::=	$\square \mid \blacksquare$
Modified usage-annotated types	A, B	::=	$\cdots \mid \overrightarrow{A} \xrightarrow{\diamond} B$
Modified values	V, W	::=	$\cdots \mid \lambda^{\circ}(\overrightarrow{x : A}) : B \,.\, M$
Modified computations	M, N	::=	$\cdots \mid V(\overrightarrow{W})$

Additional declarative typing rules $\boxed{\Gamma \vdash M : A}$

$$\textsc{T-LinLambda}\ \frac{\Gamma, \overrightarrow{x : A} \vdash M : B \qquad \mathsf{returnable}(\Gamma)}{\Gamma \vdash \lambda^{\square}(\overrightarrow{x : A}) : B \,.\, M : \overrightarrow{A} \xrightarrow{\square} B} \qquad \textsc{T-UnLambda}\ \frac{\Gamma, \overrightarrow{x : A} \vdash M : B \qquad \mathsf{returnable}(\Gamma) \qquad \mathsf{un}(\Gamma)}{\Gamma \vdash \lambda^{\blacksquare}(\overrightarrow{x : A}) : B \,.\, M : \overrightarrow{A} \xrightarrow{\blacksquare} B}$$

$$\textsc{T-AppLambda}\ \frac{\Gamma \vdash V : \overrightarrow{A} \xrightarrow{\diamond} B \qquad (\Gamma_i \vdash W_i : A_i)_{i \in 1..n}}{\Gamma + \Gamma_1 + \cdots + \Gamma_n \vdash V(\overrightarrow{W}) : B}$$

Additional declarative operations on types $\boxed{\mathsf{un}(A)}$ $\boxed{\mathsf{returnable}(A)}$ $\boxed{A \leq B}$ $\boxed{A \triangleright B}$

$$\mathsf{un}(\overrightarrow{A} \xrightarrow{\blacksquare} B) \qquad \mathsf{returnable}(\overrightarrow{A} \xrightarrow{\diamond} B) \qquad \frac{\overrightarrow{A_2} \leq \overrightarrow{A_1} \qquad B_1 \leq B_2}{(\overrightarrow{A_1} \xrightarrow{\diamond} B_1) \leq (\overrightarrow{A_2} \xrightarrow{\diamond} B_2)}$$

$$(\overrightarrow{A} \xrightarrow{\blacksquare} B) \triangleright (\overrightarrow{A} \xrightarrow{\blacksquare} B) = \overrightarrow{A} \xrightarrow{\blacksquare} B$$

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

$$(\!| (\lambda^{\circ}(\overrightarrow{x : A}) : B \,.\, M)(\overrightarrow{V}), \Sigma |\!) \longrightarrow (\!| M\{\overrightarrow{V}/\overrightarrow{x}\}, \Sigma |\!)$$

Fig. 15. Additional syntax, typing rules, and reduction rules for first-class functions

We have implemented this approach in our typechecker as an opt-in (unsound) feature, but we expect finer-grained alias analysis techniques to be an important area of future work. None of the examples presented in the paper or our evaluation rely on this more liberal treatment of data types.

5.3 First-class functions

We can extend Pat with first-class functions, but this requires care with quasi-linearity and requires additional typing information during typechecking. We will discuss quasi-linearity now but defer discussion of typechecking to §5.5.

Quasi-linearity. Consider the following expression (where the $\square$ annotation on the function abstraction denotes a *linear* function abstraction that must be applied precisely once):

```
let mb = new in
let f = (λ□(): 1 . mb ! m()) in
guard mb : m {
  receive m() from mb ↦ free mb
};
f()
```

Here we bind f to a function that sends message `m` to mailbox mb; note that it is used lexically before the **guard**, which aligns with type combination. However, after reducing the expression (assuming that a is chosen as a runtime name), we obtain the following term:

$$
\begin{array}{l}
\textbf{guard}\, a : \texttt{m}\, \{ \\
\quad \textbf{receive}\, \texttt{m}() \,\textbf{from}\, mb \mapsto \textbf{free}\, mb \\
\}; \\
(\lambda^{\square}() : \mathbf{1} \,.\, a\,!\,\texttt{m}())()
\end{array}
$$

After substituting the function body for f we now have a second-class use *after* the first-class use, violating the ordering of returnable and second-class usages.

Formalism. Figure 15 shows the additional syntax, declarative typing rules, and reduction rule for extending Pat with first-class functions. We defer discussion of algorithmic typing rules to §5.5. We extend values with fully annotated, n-ary anonymous functions $\lambda^{\diamond}(\overrightarrow{x : A}) : B \,.\, M$, where $\diamond$ is a linearity annotation that specifies whether a function is linear or unrestricted. We include n-ary functions rather than using currying because anonymous functions may only close over returnable values, to ensure they do not violate the quasi-linearity conditions on lexical scoping once applied, and therefore unary functions would be less expressive. We also extend computations with n-ary function application $V(\overrightarrow{W})$.

Rule T-LinLambda types a linear function, *i.e.*, a function that must be applied precisely once. A linear function may, for example, close over returnable mailbox output capabilities. The rule is similar to the regular function abstraction rule, but binds multiple parameters and requires that the function body closes over only variables with returnable types. Rule T-UnLambda types an unrestricted function and is similar, but requires that the function closes over only variables with unrestricted types.

The $\mathsf{un}(-)$ predicate holds only for unrestricted function types, and since functions must close over returnable environments, all function types are returnable. We use the usual subtyping rule for functions where function types are contravariant in their argument types and covariant in their result types; we write $\overrightarrow{A} \leq \overrightarrow{B}$ to denote that $A_i \leq B_i$ for all $i \in 1..n$. The sequential combination of two unrestricted function types does not affect the argument or result types.

The reduction rule for function application is the standard β-reduction rule adapted for frame stacks.

Metatheory. The addition of first-class functions does not violate the metatheoretical properties of the system: preservation and self-deadlock-freedom are maintained because we restrict λ-abstractions to close over only returnable variables and thus applying a function cannot break the invariants on combining usage annotations.

5.4 Mailbox interfaces

In the core Pat language, a global signature maps message tags to payload types. While technically convenient, this can be inflexible. First, distinct entities may wish to use the same message 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 introduced by communication, as outlined in §2:

either the received payloads or free variables in the guard body must be base types. Consider the following expression:

```
def waitAndSend(mb : ?(M1 ⊙ M2)): 1 {
  guard mb : M1 ⊙ M2 {
    receive M1(x) from mb′ ↦
      guard mb′ : M2 {
        receive M2(y) from mb″ ↦
          x ! Go();
          y ! Go();
          free mb″
      }
  }
}
```

The waitAndSend definition waits for messages M1 and M2 from mailbox *mb*. The messages carry mailbox names x and y respectively. After receiving both messages, the function sends a Go message to both x and y before freeing *mb*. This safe code is *not* typable in the calculus without interfaces, because the mailbox variable x occurs free in the second **receive** guard.

To address this issue we can associate each mailbox with an *interface* I that maps tags to payload types, and allows us to syntactically distinguish different kinds of mailbox (*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 could therefore type the waitAndSend example above, as long as we statically know x and y have different interfaces, and therefore must be different mailboxes.

Formalism. Figure 16 shows the extensions to Pat to support mailbox interfaces. We modify the definition of programs $(\mathcal{S}, \overrightarrow{D}, M)$ (where $\mathcal{S}$ is a mapping from message tags to sequences of payload types) to $(\mathcal{I}, \overrightarrow{D}, M)$, where $\mathcal{I}$ is a mapping from *interface names* I to interfaces ι. Like signatures in the unextended core calculus, interfaces ι map message tags to sequences of payload types. Given some program $\mathcal{P} = (\mathcal{I}, \overrightarrow{D}, M)$ and interface name I, we write $I(\mathtt{m})$ as syntactic sugar for $\mathcal{I}(I)(\mathtt{m})$. Note that interfaces can be mutually recursive.

We extend mailbox types $!^I E$ and $?^I E$ to include their interface name. The main alteration to the term syntax is to require an annotation on the **new** construct to specify the interface of the newly created mailbox.

We update the judgement form for guards to include the interface name of the current mailbox. Rule TI-New includes the specified interface in the mailbox type, and rule TI-Guard uses the mailbox type's interface name when typing the guards. Instead of using the program signature to determine message types, rule TGI-Recv looks up the message tag in the given interface, and also includes a more liberal check that only requires that the sets of interfaces of mailbox types contained in the payload type and the interfaces of any mailbox types used in typing M are disjoint.

We also require that combination of two mailbox types, and mailbox subtyping, are only defined if the two types have the same interface.

Modified syntax

Interface names	I		
Interfaces	ι	$::=$	$\overrightarrow{\mathtt{m} \mapsto \overrightarrow{T}}$
Interface mapping	$\mathcal{I}$	$::=$	$\overrightarrow{I \mapsto \iota}$
Modified programs	$\mathcal{P}$	$::=$	$(\mathcal{I}, \overrightarrow{D}, M)$
Modified mailbox types	J, K	$::=$	$!^I E \mid ?^I E$
Modified computations	M, N	$::=$	$\cdots \mid \mathbf{new}[I]$

Modified typing rules for computations $\boxed{\Gamma \vdash M : A}$

$$\text{TI-New}\ \frac{}{\cdot \vdash \mathbf{new}[I] : ?^I \mathbb{1}^\bullet} \qquad \text{TI-Guard}\ \frac{\Gamma_1 \vdash V : ?^I E^\bullet \qquad \{I; E\}\ \Gamma_2 \vdash \overrightarrow{G} : A \qquad \vDash E}{\Gamma_1 + \Gamma_2 \vdash \mathbf{guard}\ V\ \{\overrightarrow{G}\} : A} \qquad \text{TI-Fail}\ \frac{\Gamma \vdash V : ?^I \mathbb{0}^\bullet}{\Gamma \vdash \mathbf{fail}\ V : A}$$

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

$$\text{TGI-GuardSeq}\ \frac{(\{I; E_i\}\ \Gamma \vdash G_i : A)_{i \in 1..n}}{\{I; E_1 \oplus \cdots \oplus E_n\}\ \Gamma \vdash \overrightarrow{G} : A} \qquad \text{TGI-Free}\ \frac{\Gamma \vdash M : A}{\{I; \mathbb{1}\}\ \Gamma \vdash \mathbf{free} \mapsto M : A}$$

$$\text{TGI-Recv}\ \frac{I(\mathtt{m}) = \overrightarrow{T} \qquad \mathsf{interfaces}(\overrightarrow{T}) \cap \mathsf{interfaces}(\Gamma) = \emptyset \qquad \Gamma, y : ?^I E^\bullet, \overrightarrow{x} : \overrightarrow{\lceil T \rceil} \vdash M : B}{\{I; \mathtt{m} \odot E\}\ \Gamma \vdash \mathbf{receive}\ \mathtt{m}\,(\overrightarrow{x})\ \mathbf{from}\ y \mapsto M : B}$$

Fig. 16. Extension of Pat to support mailbox interfaces

5.5 *Using contextual type information: typechecking first-class functions and mailbox interfaces*

As described in §4, Pat's typechecker uses a *co-contextual* approach in order to generate the pattern inclusion constraints required for algorithmic typechecking. While this suffices for Pat without extensions and Pat with the data type extensions, typechecking first-class functions and mailbox interfaces requires *contextual* type information *before* the constraint generation pass.

5.5.1 *Issues with typechecking first-class functions and mailbox interfaces*

Typechecking first-class functions. Consider typechecking the application of a first-class function (here ■ signifies that the function is unrestricted):

$$(\lambda^{\blacksquare}(x : \mathsf{Int}) : \mathsf{Int}\,.\,x)(5)$$

The annotated λ expression allows us to synthesise a type and use a rule similar to TS-App. Unfortunately, the lack of contextual type information means that the approach fails as soon as we stray from applying function literals, for example:

$$\mathbf{let}\ f = (\lambda^{\blacksquare}(x : \mathsf{Int}) : \mathsf{Int}\,.\,x)\ \mathbf{in}\ f(5)$$

This is because we *do not* have information about the type of f when attempting to type $f(5)$. 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.

Modified syntax

Interface names	I		
Interfaces	ι	::=	$\overrightarrow{\mathtt{m} \mapsto \overrightarrow{\pi}}$
Modified programs	$\mathcal{P}$	::=	$(\mathcal{I}, \overrightarrow{D}, \widehat{M})$
Modified mailbox types	ς	::=	$!^I\gamma \mid ?^I\gamma$
Modified types	π, ρ	::=	$C \mid \varsigma \mid \overrightarrow{\tau} \overset{\circ}{\to} \sigma$
Modified usage-annotated types	τ, σ	::=	$C \mid \varsigma^\eta \mid \overrightarrow{\tau} \overset{\circ}{\to} \sigma$
Pre-types	R, S	::=	$C \mid \mathsf{Mailbox}(I) \mid \overrightarrow{\tau} \overset{\circ}{\to} \sigma$
Modified annotated values	$\widehat{V}, \widehat{W}$	::=	$\cdots \mid x^{\boldsymbol{R}} \mid \lambda^\circ(\overrightarrow{x:\tau}) : \sigma \,.\, \widehat{M}$
Modified annotated computations	$\widehat{M}, \widehat{N}$	::=	$\cdots \mid \widehat{V}(\overrightarrow{\widehat{W}}) \mid \mathbf{new}^{\boldsymbol{I}}[I] \mid \widehat{V}\,!^{\boldsymbol{I}}\,\mathtt{m}(\overrightarrow{\widehat{W}})$
		$\mid$	$\mathbf{guard}^{\boldsymbol{I}}\,\widehat{V} : E\,\{\overrightarrow{\widehat{G}}\} \mid \mathbf{fail}^{\boldsymbol{I}}\,\widehat{V}$

Promotion from pre-types to types $\boxed{\uparrow(R) = \pi}$

$$\uparrow(C) = C \qquad \uparrow(\overrightarrow{\tau} \overset{\circ}{\to} \sigma) = \overrightarrow{\tau} \overset{\circ}{\to} \sigma \qquad \uparrow(\mathsf{Mailbox}(I)) \text{ undefined}$$

Demotion from types to pre-types $\boxed{\downarrow(\pi) = R}$

$$\downarrow(C) = C \qquad \downarrow(\overrightarrow{\tau} \overset{\circ}{\to} \sigma) = \overrightarrow{\tau} \overset{\circ}{\to} \sigma \qquad \downarrow(!^I\gamma) = \mathsf{Mailbox}(I) \qquad \downarrow(?^I\gamma) = \mathsf{Mailbox}(I)$$

Fig. 17. Extension of Pat to support typechecking of first-class functions and interfaces (1)

Typechecking mailbox interfaces. The complexity with typing mailbox interfaces comes with recording the interface associated with each mailbox name. In Pat without extensions, the required type information can be gained from context (*i.e.*, through the message tag) and the payload types are globally available through the program's message signature $\mathcal{S}$.

For example, when typing a message send $a\,!\mathtt{m}(5)$, the base Pat type system would look up message tag m in the program's signature and see that the message had payload type Int, before checking that 5 had the corresponding type Int and a had type $!\mathtt{m}$. However, in a system with interfaces, we need knowledge of a's interface in order to look up the payload types. The checking judgement also requires knowledge of a's interface in addition to the expected pattern.

Typechecking strategy. We implement the above extensions via a contextual type-directed elaboration pass. We can annotate variables with an annotation that is useful when typing function applications. As for interfaces, users specify an interface when creating a mailbox ($\mathbf{new}[I]$); our pass then annotates sends, guards, and fail constructs with interface information (*i.e.*, $V\,!^I\,\mathtt{m}(\overrightarrow{W})$ and $\mathbf{guard}^I\,V : E\,\{\overrightarrow{G}\}$ and $\mathbf{fail}^I\,V$) for use in constraint generation.

5.5.2 *Extended syntax*

Figure 17 shows the additional syntax required to add first-class functions and interfaces.

Types and pre-types. *Pre-types* R, S are similar to types τ but type mailbox variables differently. Unlike the types we have seen so far, pre-types can be inferred and checked algorithmically using an entirely standard *contextual* approach. The key difference between pre-types and types is that mailbox names have type $\mathsf{Mailbox}(I)$ and therefore do not contain either a capability or a pattern. The main benefit of using pre-types is that they allow us to propagate mailbox interface information, and to annotate (non-mailbox-typed) variables

Type-directed elaboration $\boxed{\Omega \vdash \widehat{M} \Rightarrow R \overset{\text{elab}}{\rightsquigarrow} \widehat{N}}$ $\boxed{\Omega \vdash \widehat{M} \Leftarrow R \overset{\text{elab}}{\rightsquigarrow} \widehat{N}}$

$$\textsc{Elab-Var}\ \frac{x : R \in \Omega}{\Omega \vdash x \Rightarrow R \overset{\text{elab}}{\rightsquigarrow} x^R} \qquad \textsc{Elab-New}\ \frac{}{\Omega \vdash \mathbf{new}[I] \Rightarrow \mathsf{Mailbox}(I) \overset{\text{elab}}{\rightsquigarrow} \mathbf{new}[I]}$$

$$\textsc{Elab-Send}\ \frac{\Omega \vdash \widehat{V} \Rightarrow \mathsf{Mailbox}(I) \overset{\text{elab}}{\rightsquigarrow} \widehat{V}' \qquad I(\mathtt{m}) = \overrightarrow{\pi} \qquad (\Omega \vdash \widehat{W}_i \Leftarrow \downarrow(\pi_i) \overset{\text{elab}}{\rightsquigarrow} \widehat{W}'_i)_{i \in 1..n}}{\Omega \vdash \widehat{V}\,!\,\mathtt{m}(\overrightarrow{\widehat{W}}) \Rightarrow \mathbf{1} \overset{\text{elab}}{\rightsquigarrow} \widehat{V}'\,!^I\,\mathtt{m}(\overrightarrow{\widehat{W}'})}$$

$$\textsc{Elab-Guard}\ \frac{\Omega \vdash \widehat{V} \Rightarrow \mathsf{Mailbox}(I) \overset{\text{elab}}{\rightsquigarrow} \widehat{V}' \qquad (\{I\}\ \Omega \vdash \widehat{G}_i \Rightarrow R \overset{\text{elab}}{\rightsquigarrow} \widehat{G}'_i)_{i \in 1..n}}{\Omega \vdash \mathbf{guard}\ \widehat{V} : E\ \{\overrightarrow{\widehat{G}}\} \Rightarrow R \overset{\text{elab}}{\rightsquigarrow} \mathbf{guard}^I\ \widehat{V}' : E\ \{\overrightarrow{\widehat{G}'}\}}$$

$$\textsc{Elab-Fail}\ \frac{\Omega \vdash \widehat{V} \Rightarrow \mathsf{Mailbox}(I) \overset{\text{elab}}{\rightsquigarrow} \widehat{V}'}{\Omega \vdash \mathbf{fail}\ \widehat{V} \Leftarrow R \overset{\text{elab}}{\rightsquigarrow} \mathbf{fail}^I\ \widehat{V}'}$$

Fig. 18. Type-directed elaboration (selected rules)

with type information that can be used in constraint generation (*e.g.*, when typing functions). Pre-types that are not mailbox types can be *promoted* to a full type using the promotion operator $\uparrow(R)$, and all types can be *demoted* into a pre-type using the demotion operator $\downarrow(\pi)$.

We modify types to include *interface-annotated* mailbox types, as well as n-ary function types $\overrightarrow{\tau} \xrightarrow{\diamond} \sigma$.

Values and computations. Since first-class functions can refer to computations, we add an additional syntactic class $\widehat{V}$ of annotated values (thus allowing first-class functions to contain function bodies that have annotated **guard** expressions used for typechecking). We also extend variables x with a pre-type annotation.

Mailbox terms. We extend computations so that a user specifies an interface I when creating a mailbox (**new**$[I]$). We also augment send and guard expressions with the interface of the mailbox they operate on. Only the annotation on **new** must be specified by a user: annotations on variables, send expressions, fail expressions, and guard expressions (shaded) are instead added by type-directed elaboration.

5.5.3 *Type-directed elaboration*

We propagate pre-type annotations via a contextual type-directed elaboration phase (Figure 18). Pre-type environments Ω map variables to pre-types. We use the judgements $\Omega \vdash_{\widehat{\mathcal{P}}} \widehat{M} \Rightarrow R \overset{\text{elab}}{\rightsquigarrow} \widehat{M}'$ (read "under pre-type environment Ω and in the context of program $\widehat{\mathcal{P}}$, synthesise pre-type R for computation $\widehat{M}$, and produce elaborated term $\widehat{M}'$"), and $\Omega \vdash_{\widehat{\mathcal{P}}} \widehat{M} \Leftarrow R \overset{\text{elab}}{\rightsquigarrow} \widehat{M}'$ (read "under pre-type environment Ω and in the context of program $\widehat{\mathcal{P}}$, check that computation $\widehat{M}$ has pre-type R, and produce elaborated term $\widehat{M}'$"). We use analogous judgements for values and guards. The rules for each judgement follow the usual bidirectional typing rules for the simply typed λ-calculus, so we concentrate on the pertinent elaboration rules.

Modified constraint generation rules

$\boxed{\widehat{M} \Rightarrow_{\widehat{\mathcal{P}}} \tau \blacktriangleright \Theta; \Phi}$ $\boxed{\widehat{M} \Leftarrow_{\widehat{\mathcal{P}}} \tau \blacktriangleright \Theta; \Phi}$ $\boxed{\{I; E\}\ \widehat{G} \Leftarrow_{\widehat{\mathcal{P}}} \tau \blacktriangleright \Theta; \Phi; F}$

TS-LinLam

$$\frac{\widehat{M} \Leftarrow \sigma \blacktriangleright \Theta'; \Phi_1 \qquad \mathsf{check}(\Theta', \overrightarrow{x}, \overrightarrow{\tau}) = \Phi_2 \qquad \Theta = \lfloor \Theta' - \overrightarrow{x} \rfloor}{\lambda^{\square}(\overrightarrow{x : \tau}) : \sigma \,.\, \widehat{M} \Rightarrow \overrightarrow{\tau} \xrightarrow{\square} \sigma \blacktriangleright \Theta; \Phi_1 \cup \Phi_2}$$

TS-UnLam

$$\frac{\widehat{M} \Leftarrow \sigma \blacktriangleright \Theta'; \Phi_1 \qquad \mathsf{check}(\Theta', \overrightarrow{x}, \overrightarrow{\tau}) = \Phi_2 \qquad \Theta = \lfloor \Theta' - \overrightarrow{x} \rfloor \qquad \mathsf{unr}(\Theta) \blacktriangleright \Phi_3}{\lambda^{\blacksquare}(\overrightarrow{x : \tau}) : \sigma \,.\, \widehat{M} \Rightarrow \overrightarrow{\tau} \xrightarrow{\blacksquare} \sigma \blacktriangleright \Theta; \Phi_1 \cup \Phi_2 \cup \Phi_3}$$

TS-Var

$$\frac{\uparrow(R) = \tau}{x^R \Rightarrow \tau \blacktriangleright x : \tau; \emptyset}$$

TS-FnApp

$$\frac{\widehat{V} \Rightarrow \overrightarrow{\tau} \xrightarrow{\diamond} \sigma \blacktriangleright \Theta_V; \Phi_V \qquad (\widehat{W}_i \Leftarrow \tau_i \blacktriangleright \Theta_i; \Phi_i)_{i \in 1..n} \qquad \Theta_V + \Theta_1 + \cdots + \Theta_n \blacktriangleright \Theta; \Phi}{\widehat{V}(\overrightarrow{\widehat{W}}) \Rightarrow \sigma \blacktriangleright \Theta; \Phi \cup \Phi_V \cup \Phi_1 \cup \cdots \cup \Phi_n}$$

TS-Send

$$\frac{I(\mathtt{m}) = \overrightarrow{\pi} \qquad \widehat{V} \Leftarrow !^I \mathtt{m}^{\circ} \blacktriangleright \Theta_V; \Phi_V \qquad (\widehat{W}_i \Leftarrow \lceil \pi_i \rceil \blacktriangleright \Theta_i; \Phi'_i)_{i \in 1..n} \qquad \Theta_V + \Theta_1 + \cdots + \Theta_n \blacktriangleright \Theta; \Phi}{\widehat{V}\, !^I \mathtt{m}(\overrightarrow{\widehat{W}}) \Rightarrow \mathbf{1} \blacktriangleright \Theta; \Phi \cup \Phi_V \cup \Phi'_1 \cup \cdots \cup \Phi'_n}$$

TS-New

$$\frac{}{\mathbf{new}[I] \Rightarrow ?^I \mathbb{1}^{\bullet} \blacktriangleright \cdot; \emptyset}$$

TC-Guard

$$\frac{\{I; E\}\ \overrightarrow{G} \Leftarrow \tau \blacktriangleright \Theta_1; \Phi_1; F \qquad \widehat{V} \Leftarrow ?^I F^{\bullet} \blacktriangleright \Theta_2; \Phi_2 \qquad \Theta_1 + \Theta_2 \blacktriangleright \Theta; \Phi_3}{\mathbf{guard}^I\, \widehat{V} : E\, \{\overrightarrow{G}\} \Leftarrow \tau \blacktriangleright \Theta; \Phi_1 \cup \Phi_2 \cup \Phi_3 \cup \{E <: F\}}$$

TC-Fail

$$\frac{\widehat{V} \Leftarrow ?^I \mathbb{0}^{\bullet} \blacktriangleright \Theta; \Phi}{\mathbf{fail}^I\, \widehat{V} \Leftarrow \tau \blacktriangleright \Theta; \Phi}$$

TCG-Recv

$$\frac{\widehat{M} \Leftarrow \tau \blacktriangleright \Theta', y : ?^I \gamma^{\bullet}; \Phi_1 \qquad I(\mathtt{m}) = \overrightarrow{\pi} \qquad \Theta = \Theta' - \overrightarrow{x} \qquad \mathsf{interfaces}(\overrightarrow{\pi}) \cap \mathsf{interfaces}(\Theta) = \emptyset \qquad \mathsf{check}(\Theta', \overrightarrow{x}, \overrightarrow{\lceil \pi \rceil}) = \Phi_2}{\{I; E\}\ \mathbf{receive}\ \mathtt{m}(\overrightarrow{x})\ \mathbf{from}\ y \mapsto \widehat{M} \Leftarrow \tau \blacktriangleright \Theta; \Phi_1 \cup \Phi_2 \cup \{E\,/\,\mathtt{m} <: \gamma\}; \mathtt{m} \odot (E\,/\,\mathtt{m})}$$

Fig. 19. Extension of Pat to support typechecking of first-class functions and interfaces (2)

Rule Elab-Var annotates a variable with its pre-type, and rule Elab-New synthesises type Mailbox(I), where I corresponds to the user-specified interface. Rule Elab-Send states that if pre-type Mailbox(I) can be synthesised for the target mailbox, then the rule looks up the payload types for the message in interface I and checks all payloads against the (demoted) payload types. The result is a send expression annotated with the interface I.

Rule Elab-Guard synthesises pre-type Mailbox(I) for the given mailbox, and then checks that every guard clause synthesises the same pre-type R. The interface is passed to the guard synthesis judgement so that the payload types can be retrieved when typing a **receive** guard. Again, the result is the guard annotated with interface I.

Finally, rule Elab-Fail is a checking rule for **fail** V: it synthesises pre-type Mailbox(I) for V and annotates the **fail** construct with interface I.

5.5.4 *Constraint generation rules*

Figure 19 shows the constraint generation rules for the extended calculus. We require three new rules for first-class functions: rule TS-LinLam types a linear anonymous function by checking that the body has the given result type, and the inferred environment uses variables

consistently with the parameter annotations: the rule synthesises a type consistent with the annotation. Further, we require that the inferred environment only closes over variables with returnable types. Rule TS-UnLam is similar, but additionally requires that the inferred environment is unrestricted. Rule TS-FnApp synthesises a type for the function (made possible using either the type annotation on the function abstraction, or the annotation on the function variable); it then checks that the arguments and results have the correct types.

As for the rules that support interfaces, rule TS-Send is similar but looks up the types according to the interface rather than the global signature, and checks that the target mailbox has the given interface. Rule TS-New synthesises a mailbox type with the user-supplied interface. Finally, we modify the shape of the guard typing judgement to record the interface of the mailbox being guarded upon, and use this to look up the desired payload types in TCG-Recv.

Example. To illustrate this approach, consider our earlier troublesome example:

$$\textsf{let}\ f = (\lambda^{\blacksquare}(x : \mathsf{Int}) : \mathsf{Int}\,.\,x)\ \textsf{in}\ f(5)$$

This example cannot be typed purely co-contextually since we do not have the required type information for f when typing the function application.

Let $\Omega = f : \mathsf{Int} \xrightarrow{\blacksquare} \mathsf{Int}$. The type-directed elaboration phase results in the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : \mathsf{Int} \vdash x \Rightarrow \mathsf{Int} \overset{\mathsf{elab}}{\rightsquigarrow} x^{\mathsf{Int}}}}{x : \mathsf{Int} \vdash x \Leftarrow \mathsf{Int} \overset{\mathsf{elab}}{\rightsquigarrow} x^{\mathsf{Int}}}}{\cdot \vdash \lambda^{\blacksquare}(x : \mathsf{Int}) : \mathsf{Int}\,.\,x \Rightarrow \mathsf{Int} \xrightarrow{\blacksquare} \mathsf{Int} \overset{\mathsf{elab}}{\rightsquigarrow} \lambda^{\blacksquare}(x : \mathsf{Int}) : \mathsf{Int}\,.\,x^{\mathsf{Int}}} \qquad \dfrac{\dfrac{}{\Omega \vdash f \Rightarrow \mathsf{Int} \xrightarrow{\blacksquare} \mathsf{Int} \overset{\mathsf{elab}}{\rightsquigarrow} f^{\mathsf{Int} \xrightarrow{\blacksquare} \mathsf{Int}}} \quad \dfrac{}{\Omega \vdash 5 \Leftarrow \mathsf{Int} \overset{\mathsf{elab}}{\rightsquigarrow} 5}}{\Omega \vdash f(5) \Rightarrow \mathsf{Int} \overset{\mathsf{elab}}{\rightsquigarrow} f^{\mathsf{Int} \xrightarrow{\blacksquare} \mathsf{Int}}(5)}}{\cdot \vdash \textsf{let}\ f = (\lambda^{\blacksquare}(x : \mathsf{Int}) : \mathsf{Int}\,.\,x)\ \textsf{in}\ f(5) \Rightarrow \mathsf{Int} \overset{\mathsf{elab}}{\rightsquigarrow} \textsf{let}\ f = (\lambda^{\blacksquare}(x : \mathsf{Int}) : \mathsf{Int}\,.\,x^{\mathsf{Int}})\ \textsf{in}\ f^{\mathsf{Int} \xrightarrow{\blacksquare} \mathsf{Int}}(5)}$$

Finally, we can type the expression with the modified constraint generation rules. We omit the straightforward environment operations in the premises of the rules for simplicity. Let **D** be the following subderivation:

$$\dfrac{\dfrac{\dfrac{}{f^{\mathsf{Int} \xrightarrow{\blacksquare} \mathsf{Int}} \Rightarrow \mathsf{Int} \xrightarrow{\blacksquare} \mathsf{Int} \blacktriangleright f : \mathsf{Int} \xrightarrow{\blacksquare} \mathsf{Int};\ \emptyset} \quad \dfrac{\dfrac{}{5 \Rightarrow \mathsf{Int} \blacktriangleright \cdot;\ \emptyset}}{5 \Leftarrow \mathsf{Int} \blacktriangleright \cdot;\ \emptyset}}{f^{\mathsf{Int} \xrightarrow{\blacksquare} \mathsf{Int}}(5) \Rightarrow \mathsf{Int} \blacktriangleright f : \mathsf{Int} \xrightarrow{\blacksquare} \mathsf{Int};\ \emptyset}}{f^{\mathsf{Int} \xrightarrow{\blacksquare} \mathsf{Int}}(5) \Leftarrow \mathsf{Int} \blacktriangleright f : \mathsf{Int} \xrightarrow{\blacksquare} \mathsf{Int};\ \emptyset}$$

Then, we can construct the whole derivation using TC-LetNoAnn1:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x^{\mathsf{Int}} \Leftarrow \mathsf{Int} \blacktriangleright x : \mathsf{Int};\ \emptyset}}{\lambda^{\blacksquare}(x : \mathsf{Int}) : \mathsf{Int}\,.\,x^{\mathsf{Int}} \Rightarrow \mathsf{Int} \xrightarrow{\blacksquare} \mathsf{Int} \blacktriangleright \cdot;\ \emptyset}}{\lambda^{\blacksquare}(x : \mathsf{Int}) : \mathsf{Int}\,.\,x^{\mathsf{Int}} \Leftarrow \mathsf{Int} \xrightarrow{\blacksquare} \mathsf{Int} \blacktriangleright \cdot;\ \emptyset}}{\textsf{let}\ f = (\lambda^{\blacksquare}(x : \mathsf{Int}) : \mathsf{Int}\,.\,x^{\mathsf{Int}})\ \textsf{in}\ f^{\mathsf{Int} \xrightarrow{\blacksquare} \mathsf{Int}}(5) \Leftarrow \mathsf{Int} \blacktriangleright \cdot;\ \emptyset}\ \mathbf{D}$$

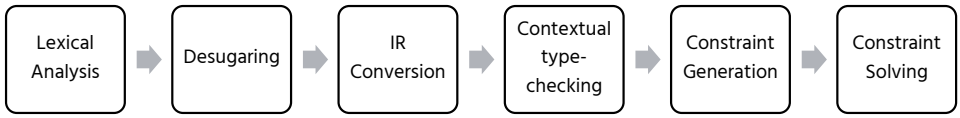


Fig. 20. Pat typechecking pipeline

6 Implementation and evaluation

We outline the implementation of a mailbox type checker written in OCaml, and evidence the expressiveness of Pat via a selection of representative 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 (once extended with sums, products, and lists) can express all of the 11 Savina expressiveness benchmarks used by Neykova and Yoshida (2017b). This selection confirms that our language can express real-world scenarios that arise in concurrent and distributed computing. The full Savina benchmark suite contains 30 programs; we base our choice of Savina programs on the selection implemented by Neykova and Yoshida (2017b) in order to demonstrate that mailbox types are as expressive as multiparty session types for actor systems, at least within the context of this set of examples.

Finally, we describe the *Sleeping Barber* example in detail, and show a case study provided by an industrial partner that develops control software for factories.

6.1 Implementation overview

Pat programs consist of *interface definitions* that detail the messages and payload types supported by a mailbox; a series of *function definitions*; and finally a *program body* that serves as an entry point. Programs are typechecked in the six-stage pipeline outlined in Figure 20:

Lexical analysis Performs standard lexing and parsing using the OCaml Menhir library.

Desugaring Desugars **free** V to **guard** $V : \mathbb{1}\ \{\textbf{free} \mapsto ()\}$ and adds omitted pattern variables.

IR conversion Transforms the surface language (supporting nested expressions) to our explicitly sequenced intermediate representation.

Contextual typechecking Performs a (standard) bidirectional typing pass to propagate contextual type information (§5.5).

Constraint generation Implements the algorithmic type system from §4, and generates a set of pattern inclusion constraints.

Constraint solving Applies the constraint solving approach detailed in §4.3, and invokes Z3 (de Moura and Bjørner 2008) to determine whether the generated constraints are satisfiable.

The Pat typechecker operates in two *modes* that determine how receive guards are typechecked. *Strict* mode uses the lightweight syntactic checks outlined in §3 and §4, whereas *interface* mode uses interface type information (§5.5) to relax these checks. This means that every Pat program accepted in strict mode is also accepted in interface mode.

Table 1. Typechecking concurrent actor examples in Pat

#	Name	Description	Strict	SLOC	Time (ms)
Original mailbox calculus models taken from de'Liguoro and Padovani (2018)					
1	Lock	Concurrent lock modelling mutual exclusion	•	35	29.4
2	Future	Future variable written once and read multiple times	•	32	25.0
3	Account	Concurrent accounts exchanging debits and credits	•	38	20.0
4	AccountF	Concurrent accounts with debits effected via futures	•	65	37.1
5	Master-Worker	Master-worker parallel network	•	70	29.8
6	Session Types	Session-typed communicating actors using one arbiter	○	96	85.3
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	•	47	26.9
8	Thread Ring	Ring network relaying one token with counter k	○	76	40.6
9	Counter	One actor sending messages to a second that sums the count, k	○	56	30.0
10	K-Fork	Fork-join where a central actor delegates k requests	•	41	8.0
11	Fibonacci	Server delegating terms $k-1$ and $k-2$ to parallel actors	•	43	27.1
12	Big	Peer-to-peer actors randomly exchanging k messages	○	108	64.8
13	Philosopher	Dining philosophers problem	○	90	73.8
14	Smokers	Central arbiter allocating k messages to actors	○	82	40.7
15	Log Map	Computes $x_{k+1} = r \cdot x_k (1 - x_k)$ using parallel actors	○	103	63.2
16	Transaction	Request-reply communication initiated by a teller actor	○	96	51.2
17	Barber	Multiple customers who interact with one 'sleeping' barber	○	100	77.5

6.2 Expressiveness and typechecking time

Table 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 7–17 are the suite of Savina benchmarks (Imam and Sarkar 2014, Table 1, No. 1–4, 6, 7, 12–16) used by Neykova and Yoshida (2017b). The table indicates whether a Pat program can be checked in strict mode, and therefore also interface mode (denoted by •), or whether it requires interface mode to typecheck (denoted by ○). As a simple measure of their size, we report the number of Source Lines of Code (SLOC) for each example. We report the mean typechecking time, excluding phases 1–3 of the pipeline. Measurements are made on a MacBook M1 Pro with 8GB of memory, running macOS 15.4 and OCaml 5.2. To minimise variability we report the mean time from 1000 repetitions. The number of repetitions was determined empirically by calculating the coefficient of variation (CV) (Devore and Berk 2012), *i.e.*, the ratio of the standard deviation to the mean, CV $= \sigma / \bar{x}$, for different repetitions until an adequately low value ($< 10\%$) was obtained.

6.2.1 Benchmarks

Table 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 (8, 9, and 12–17) require contextual type information to typecheck.

The new list extension (§5.1.3) allows us to implement many examples more idiomatically: specifically we can express examples that use fixed collections (examples 8, 10, and 12–16)

as lists. The original implementations in Fowler et al. (2023a) emulated the collections using parameters. 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 does not require actor programs to be rewritten to accommodate the type system.

Despite supporting a permissive many-sender communication model, checking mailbox types does not incur high typechecking runtimes (see Table 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.

Nevertheless, typechecking times for all of these benchmarks are under 100ms. The benchmark that takes the longest to typecheck is the *Session Types* benchmark, which has 12 different messages that can be exchanged along with 12 different guard expressions. The benchmark that has the smallest typechecking time is *K-Fork*, which has only one type of message and a single guard expression, and therefore a much simpler communication structure. This would suggest that, as would be expected, constraint solving takes the most time during typechecking.

6.2.2 Worked example: Sleeping Barber

To illustrate how Pat is used in practice, this section describes the development of the *Sleeping Barber* benchmark in greater detail. The Sleeping Barber problem is a classic synchronisation problem, originally specified by Dijkstra (1968), and one of the Savina benchmarks:

- A barber is working in a barber shop with a waiting room.
- When the barber is ready for the next customer, they check the waiting room.
  - If the waiting room has waiting customers, then the barber calls the next customer in for their haircut. Once the barber is finished cutting the customer's hair, they check the waiting room again.
  - If the waiting room is empty, then the barber will sleep.
- When a customer enters the barber shop, they check to see if the waiting room has space. If there is space, then the customer will wait; if there is no space, then the customer will leave. If the barber is asleep, then the customer will need to wake the barber.

Pat syntax. In this section we use Pat syntax as typechecked by the implementation. In contrast to the formal calculi, implemented Pat programs are typeset in `teletype`. The syntax expected by the typechecker closely resembles the core calculus, with some changes:

- An interface-annotated mailbox type is written with the interface preceding the capability, and the pattern after the capability. For example, $?^{I}\texttt{Msg}$ is written in Pat source as `I?Msg`.

- Since Pat's typechecker allows a degree of pattern inference, it is also possible to omit the message pattern in type annotations: for example, `I?` is shorthand for $?^{I}\alpha$, where α is a fresh pattern variable.
- Patterns are written using ASCII regular expression syntax. Specifically, composition $\mathbf{M} \odot \mathbf{N}$ is written as `M.N`; choice $\mathbf{M} \oplus \mathbf{N}$ is written as `M + N`; and replication $\mathbf{M}^{\star}$ is written as `M*`. The unreliable ($\mathbb{0}$) and empty ($\mathbb{1}$) mailbox patterns are written as `0` and `1` respectively.

Interfaces. The Savina benchmarks implement an (untyped) actor-based version of the problem by modelling the barber, customers, and waiting room as individual actors. We take a similar approach in Pat, and begin by defining interfaces for the three types of mailbox. We also need a fourth interface, `WaitingRoomBuffer`, to model customers waiting in the waiting room.

The `WaitingRoom` will receive three types of message:

```
interface WaitingRoom { Enter(Customer!), Next(Barber!), Sleeping(Barber!) }
```

The `Enter` message is received from a customer when they enter the barber shop, and contains an output reference to the customer's mailbox. The `Next` message is received from the barber to signify that they are ready to service the next customer, and the `Sleeping` message is received from the barber to signify that the barber has gone to sleep. Both of the latter messages contain an updated output reference to the barber's mailbox.

The `WaitingRoomBuffer` mailbox models the customers who are currently in the waiting room. A mailbox with interface `WaitingRoomBuffer` can receive a single kind of message, `WaitingCustomer`, that contains a reference to a `Customer` mailbox.

```
interface WaitingRoomBuffer { WaitingCustomer(Customer!) }
```

The `Barber` will also receive three types of message:

```
interface Barber { Wake(WaitingRoom!), CustomerReady(Customer!, WaitingRoom!),
  RoomEmpty(WaitingRoom!) }
```

The `Wake` message is received from an empty waiting room when a customer has entered and the barber is asleep. The `CustomerReady` and `RoomEmpty` messages are received from the waiting room in response to a `Next` message sent by the barber; the former notifies the barber of the next customer, and the latter notifies the barber that the room is empty and that they can go to sleep. The `CustomerReady` message includes a reference that the barber can use to communicate with the customer, and both messages include an updated `WaitingRoom` reference.

Finally, the `Customer` can receive four types of message:

```
interface Customer { Full(), Wait(), Start(), Done() }
```

After entering the waiting room, the customer will either receive a `Full` or `Wait` message from the waiting room to state that the customer should leave or wait in the waiting room respectively. When the barber is ready to cut the customer's hair, the barber will send the customer a `Start` message, and when the barber is finished with the haircut, the barber will send a `Done` message.

Customer. The implementation of the customer is fairly straightforward. A customer must send an `Enter` message to the waiting room, and then wait for a response:

```
def customer(self: Customer?, waitingRoom: WaitingRoom!): Unit {
```

```
    waitingRoom ! Enter(self);
    guard self : Full + (Wait.Start.Done) {
        receive Full() from self ->
            print("Room is full. Oh well, best go somewhere else");
            free(self)
        receive Wait() from self ->
            print("Waiting");
            waitingCustomer(self)
    }
}
```

There are two possible responses from the waiting room: either a `Full` message to say that the waiting room is full (at which point there are no possible interactions and the only thing to do is to free the mailbox), or a `Wait` message. If the latter, the mailbox types *also* guarantee that the customer will need to handle a `Start` message (when the barber begins their haircut), and a `Done` message (when the barber has finished their haircut).

The `waitingCustomer` function processes the `Start` and `Done` messages and frees the customer's mailbox after the haircut is finished:

```
def waitingCustomer(self: Customer?): Unit {
    guard self : Start.Done {
        receive Start() from self ->
            print("Barber is starting my haircut");
            guard self : Done {
                receive Done() from self ->
                    print("Haircut finished!");
                    free(self)
            }
    }
}
```

Waiting room. To obtain precise mailbox types, we model the waiting room as two mutually recursive functions: one for when the barber is asleep, and one for when there are waiting customers.

```
def waitingRoomSleepingBarber(self: WaitingRoom?, barber: Barber!,
        capacity: Int): Unit {
    guard self : Enter* {
        free -> ()
        receive Enter(customer) from self ->
            let buffer = new[WaitingRoomBuffer] in
            buffer ! WaitingCustomer(customer);
            customer ! Wait();
            barber ! Wake(self);
            waitingRoom(self, buffer, 1, capacity)
    }
}
```

The `waitingRoomSleepingBarber` function takes an input mailbox reference to its own mailbox, an output mailbox reference to the barber, and an integer denoting the capacity of the waiting room. In this state, the barber is asleep and cannot send any more messages until they are awake, and so the waiting room can only receive `Enter` messages (or free itself if no more `Enter` messages can be sent). When the waiting room receives an `Enter` message, it creates

a new mailbox `buffer` that is used to model the queue of customers waiting for the barber; stores the request in the queue by sending the buffer a `WaitingCustomer` message containing an output reference to the customer's mailbox; sends the customer a `Wait` message; sends the barber a `Wake` message to wake them up; and then transitions to the non-empty state.

In the non-empty state, the waiting room's mailbox can also receive many `Enter` messages from customers, but must additionally receive a single `Next` message from the barber:

```
def waitingRoom(self: WaitingRoom?, buffer: WaitingRoomBuffer?,
                numCustomers: Int, capacity: Int): Unit {
    guard self : Enter* . Next {
        receive Enter(customer) from self -> ...
        receive Next(barber) from self -> ...
    }
}
```

Processing an `Enter` message is similar to before: if the number of customers exceeds the capacity of the waiting room, then the waiting room will respond with a `Full` message; otherwise, the waiting room will queue the request and send the customer a `Wait` message:

```
receive Enter(customer) from self ->
    if (numCustomers > capacity) {
        customer ! Full();
        waitingRoom(self, buffer, numCustomers, capacity)
    } else {
        buffer ! WaitingCustomer(customer);
        customer ! Wait();
        waitingRoom(self, buffer, numCustomers + 1, capacity)
    }
```

To process a `Next` message, the waiting room will inspect the `buffer`, which may contain zero or more `WaitingCustomer` messages:

```
receive Next(barber) from self ->
    guard buffer : WaitingCustomer* {
        receive WaitingCustomer(customer) from buffer ->
            barber ! CustomerReady(customer, self);
            waitingRoom(self, buffer, numCustomers - 1, capacity)
        free ->
            # In this case there are no more customers.
            # Notify the barber that they can sleep; move to sleeping state
            barber ! RoomEmpty(self);
            guard self : Enter* . Sleeping {
                # Receive notification that barber is asleep
                receive Sleeping(barber) from self ->
                    waitingRoomSleepingBarber(self, barber, capacity)
            }
    }
```

When processing a `WaitingCustomer` message, the waiting room responds to the barber with a `CustomerReady` message containing the output reference with which to communicate with the customer, and recursively invokes the `waitingRoom` function with a decremented customer count. If the **`free`** guard is triggered, we know that there are no pending `WaitingCustomer` messages and therefore that the waiting room is empty. So the function notifies the barber that the room

is empty, awaits a `Sleeping` notification, and transitions to the `waitingRoomSleepingBarber` state.

Remark. We use a mailbox, rather than a list, for storing the contents of the waiting room to avoid the potential for unsafe aliasing. Consider a signature of the `waitingRoom` function that maintains the waiting room as a list:

```
def waitingRoom(self: WaitingRoom?, buffer: List(Customer!),
                numCustomers: Int, capacity: Int): Unit
```

Here we would also need to modify the **`receive`** clause for the `Enter` message to add the customer to the waiting room:

```
receive Enter(customer) from self ->
    if (numCustomers > capacity) {
        customer ! Full();
        waitingRoom(self, buffer, numCustomers, capacity)
    } else {
        customer ! Wait();
        waitingRoom(self, customer :: buffer, numCustomers + 1, capacity)
    }
```

However there is no guarantee that a reference for the customer does not already exist in the buffer, and therefore deconstructing the list could introduce unsafe aliasing. Moreover we can only safely store *returnable* values in a buffer, whereas the `customer` reference must be treated as second-class. Both issues are avoided by using a mailbox to model the waiting room, since Pat's type system allows us to safely reason about one customer at a time.

It is also possible (if slightly less elegant) to model the Sleeping Barber problem without a separate buffer mailbox by using self-messages.

Barber. Finally, the barber process begins in the `sleepingBarber` state, where they have only a reference to a `Barber` mailbox. The `self` mailbox has pattern `Wake + 1` indicating that the barber is able to respond to a `Wake` message (when a customer enters the waiting room), or to free itself if no customers are ever spawned. When the barber receives a `Wake` message, the process sends a `Next` message to the waiting room provided in the message. Once awoken, the barber calls the `barber` function that models an awake barber.

```
def sleepingBarber(self: Barber?): Unit {
    guard self : Wake + 1 {
        free -> ()
        receive Wake(room) from self ->
            room ! Next(self);
            barber(self)
    }
}
```

The `barber` function again takes a reference to the barber's mailbox. However this time we know that the waiting room must respond with either a `RoomEmpty` message if there are no waiting customers, at which point the barber can go back to sleep, or a `CustomerReady` message if another customer was waiting.

```
def barber(self: Barber?): Unit {
    guard self : RoomEmpty + CustomerReady {
        receive RoomEmpty(room) from self ->
```

```
            print("Room empty; going to sleep");
            room ! Sleeping(self);
            sleepingBarber(self)
        receive CustomerReady(customer, room) from self ->
            customer ! Start();
            print("Cutting hair");
            print("Finished cutting hair; notifying customer and waiting room");
            customer ! Done();
            room ! Next(self);
            barber(self)
    }
}
```

If the room is empty, the barber will send a `Sleeping` message to the waiting room before calling the `sleepingBarber` function, which models the barber falling asleep. When the barber receives a `CustomerReady` message, which contains a reference to the customer and the room, the barber will send a `Start` message to the customer at the start of their haircut and a `Done` message at the end of their haircut, before notifying the waiting room by sending a `Next` message.

Discussion. Perhaps surprisingly, many of the messages exchanged between participants contain references to the sender, even though a reference may already be in scope. For example, a mailbox with the `Barber` interface can receive `Wake`, `CustomerReady`, and `RoomEmpty` messages from the `WaitingRoom`, and all contain a `WaitingRoom` reference with which to respond.

As there is only one waiting room it is common practice in some actor languages, like Erlang, to spawn the barber process with a reference to it, simplifying the `Barber` interface:

```
interface Barber { Wake(), CustomerReady(Customer!), RoomEmpty() }
def barber(self: Barber?, room: WaitingRoom!): Unit { ... }
def sleepingBarber(self: Barber?, room: WaitingRoom!): Unit { ... }
```

The drawback of this approach is a loss of typing precision. With the previous approach we know statically that the barber will *only have a reference to the waiting room when processing a request from the room* and this allows us to specify two precise mailbox types for the two states of the waiting room. Specifically, when the barber is awake, the `self` mailbox in the `waitingRoom` function has type `WaitingRoom?(Enter* . Next)`, meaning that its mailbox can contain many `Enter` messages from customers, but will also eventually contain a *single* `Next` message from the barber when they are ready for the next customer. When the barber is asleep, the barber will *not* have a reference to the waiting room, and thus we know statically that the waiting room *cannot* contain a `Next` message; we can therefore give the `self` mailbox the type `WaitingRoom?Enter*` in the `waitingRoomSleepingBarber` function.

If we were instead to keep the `room` variable in scope across both functions, we would need to use the less precise mailbox type `WaitingRoom?(Enter* . Next* . Sleeping*)` in both states of the waiting room, since there are no guarantees about whether the barber is asleep or awake, and the mailbox type no longer specifies that the barber will only ever send a single `Next` or `Sleeping` message at a time.

The idiom of sending a mailbox reference even when it could be in scope may seem contrary to common practice in untyped languages like Erlang. It does, however, closely mirror the style employed by Akka's typed references (Akka Team 2026), where messages often include an actor reference that allows a process to respond with a message of a different

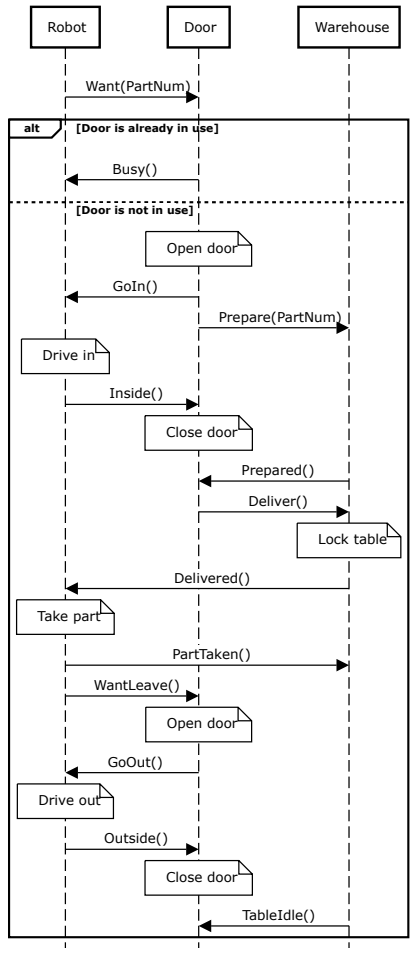


Fig. 21. Factory use case

type. We posit that the practice of re-sending actor references with a different type is therefore more of a typed actor idiom than an idiom specific to mailbox typing.

6.2.3 *Factory case study*

Finally we describe a real-world use case provided by *Actyx AG*[2], 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 the three entities, *Robot*, *Door*, and *Warehouse*, is shown in Figure 21. Typechecking our case study relies on contextual type information (see §5.5), and takes 89.6ms.

Interfaces. The messages accepted by the `Robot`, `Door`, and `Warehouse` are defined by the following interfaces.

```
1 interface Robot {
```

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

```
2   GoIn(Door!), GoOut(Door!), Busy(), Delivered(Warehouse!, Door!)
3 }
4
5 interface Door {
6   Want(Int, Robot!), Inside(Robot!), Outside(),
7   WantLeave(Robot!), Prepared(Warehouse!),
8   TableIdle(Warehouse!)
9 }
10
11 interface Warehouse {
12   Prepare(Int, Door!), Deliver(Robot!, Door!), PartTaken()
13 }
```

Robot. Each `Robot` is initially `idle` and issues a `Want` message to the `Door` to obtain access to the `Warehouse` (line 15). The `Door` replies either with the message `Busy`, in which case the `Robot` terminates (lines 17–18), or `GoIn`, to which the `Robot` replies with an `Inside` message before transitioning to the `working` state (lines 19–21). When in the `working` state, the `Robot` expects one `Delivered` message to inform the `Robot` that the part is delivered by the `Warehouse`, as asserted by the guard on line 25. The recipient `Robot` replies to the `Warehouse` with the `PartTaken` message, and notifies the `Door` that it wants to exit by sending a `WantLeave` message on lines 28–29. It then awaits a `GoOut` message and finalises its negotiation with the `Door` through an `Outside` message.

```
14 def idle(self: Robot?, door: Door!): Unit {
15   door ! Want(0, self);
16   guard self: (Busy + GoIn) {
17     receive Busy() from self →
18       free(self)
19     receive GoIn(door) from self →
20       door ! Inside(self);
21       working(self)
22   }
23 }
```

```
24 def working(self: Robot?): Unit {
25   let self = guard self: Delivered {
26     receive Delivered(wh, door)
27         from self →
28       wh ! PartTaken();
29       door ! WantLeave(self); self
30   } in guard self: GoOut {
31     receive GoOut(door) from self →
32       door ! Outside();
33       free(self)
34   }
35 }
```

Door. The `Door` accepts zero or more `Want` messages, replying to each with `Busy` or `GoIn`. In the latter case, the `Door` informs the `Warehouse` of an inbound `Robot` by sending it a `Prepare` message, and then transitions to the `busy` state (lines 39–42). Both `GoIn` and `Prepare` include an updated self-reference to ensure precise types. The **`free`** guard on line 38 handles the case where no `Robots` are present, *i.e.*, no `Want` messages are received by the `Door`.

```
36 def clear(self: Door?, wh: Warehouse!): Unit {
37   guard self: Want* {
38     free → ()
39     receive Want(part, robot) from self →
40       robot ! GoIn(self);
41       wh ! Prepare(part, self);
42       busy(self)
43   }
44 }
```

When `busy`, the `Door` mailbox potentially contains an `Inside` message from the admitted `Robot`, a `Prepared` message from the `Warehouse`, and `Want` messages sent by other `Robots` requesting access (line 46). These `Want` messages are answered with `Busy`, as lines 47–49 show. Once the `Door` receives the `Inside` message, it awaits a `Prepared` message issued by the `Warehouse`, before notifying the latter that the `Robot` is collecting its part via `Deliver` (lines 50–53).

```
45 def busy(self: Door?): Unit {
46   guard self: Inside.Prepared.Want* {
47     receive Want(partNum, robot) from self →
48       robot ! Busy();
49       busy(self)
50     receive Inside(robot) from self →
51       guard self: Prepared.Want* {
52         receive Prepared(wh) from self →
53           wh ! Deliver(robot, self);
54           guard self: WantLeave.TableIdle.Want* {
55             receive WantLeave(robot) from self →
56               robot ! GoOut(self);
57               ready(self)
58             }
59         }
60   }
61 }
```

Eventually, the `Robot` requests to exit the `Warehouse` by sending `WantLeave` to the `Door`, which handles it on lines 55–56. The `Door` transitions to the `ready` state, whereupon it confirms that the `Robot` has exited and that the `Warehouse` is available; these interactions are captured by the `Outside` and `TableIdle` messages respectively (lines 66–75). Finally, the `Door` transitions back to `clear` on lines 69 and 74, ready to service other `Robots`.

```
62 def ready(self: Door?): Unit {
63   guard self: Outside.TableIdle.Want* {
64     # Handle messages Outside and TableIdle in
65     # any order and clear door.
66     receive Outside() from self →
67       guard self: TableIdle.Want* {
68           receive TableIdle(wh) from self →
69             clear(self, wh)
70       }
71     receive TableIdle(wh) from self →
72       guard self: Outside.Want* {
73         receive Outside() from self →
74           clear(self, wh)
75       }
76   }
77 }
```

Warehouse. The `Warehouse` in its `empty` state expects a `Prepare` message (if there are `Robots` in the system), or none (if no `Robot` requests access), *i.e.*, the guard `Prepare + 1` on line 79. When a part is requested, the `Warehouse` transitions to the `engaged` state and awaits a `Deliver` message from the `Door`, notifying the `Robot` collecting the part via a `Delivered` message (lines 88–95), and then transitions to the `given` state (lines 96–103). The `Robot` acknowledges the

delivery by sending `PartTaken`, as required by the guard on line 98. To conclude its interaction with the `Door`, the `Warehouse` sends `TableIdle` before transitioning back to the `empty` state.

```
78 def empty(self: Warehouse?): Unit {
79   guard self: Prepare + 1 {
80     free → ()
81     receive Prepare(partNum, door)
82         from self →
83       door ! Prepared(self);
84       engaged(self)
85   }
86 }
87
88 def engaged(self: Warehouse?): Unit {
89   guard self: Deliver {
90     receive Deliver(robot, door)
91         from self →
92       robot ! Delivered(self, door);
93       given(self, door)
94   }
95 }
```

```
96 def given(self: Warehouse?,
97           door: Door!): Unit {
98   guard self : PartTaken {
99     receive PartTaken() from self →
100      door ! TableIdle(self);
101      empty(self)
102  }
103 }
104
105 # Launcher function.
106 def main(): Unit {
107   # n Robot mailboxes.
108   let robot_i = new[Robot] in
109   let door = new[Door] in
110   let wh = new[Warehouse] in
111   # Door.
112   spawn { clear(door, wh) };
113   # n Robots.
114   spawn { idle(robot_i, door) };
115   # Warehouse.
116   spawn { empty(wh) }
117 }
```

The `main()` function creates n `Robot` mailboxes, together with a `Door` and `Warehouse` mailbox, spawning the respective processes on lines 112–116.

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), 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. Fowler and Hu (2026) introduce a language design that allows session-typed actor communication by enforcing multiparty session typing using a flow-sensitive effect system, and their language allows actors to be involved in multiple sessions by using ideas from event-driven programming.

Session types are helpful when there are clear, structured communication flows between a fixed set of participants. Session types also provide a convenient top-down development methodology, whereas mailbox types need to be added to individual components in a more bottom-up fashion. However, the big disadvantage of using session types over mailbox types is that session types are specified using point-to-point interactions, and this requires

either designing applications with session types from the beginning, or rewriting existing applications to use a session-typed communication style. In contrast, our mailbox typing approach naturally fits actor-style programming without needing to restructure the code to group participants into sessions.

Bagherzadeh and Rajan (2017) define a type system for active objects (de Boer et al. 2007) which can rule out message 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 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. Neubauer and Thiemann (2004) were the first to design a Haskell embedding of session types, whereas Vasconcelos et al. (2006) were the first to design a core functional language with session types, though their approach had an imperative-style interface and required complex alias checking. Gay and Vasconcelos (2010) integrated session types in a functional language based 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; Wadler 2014)). 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.*, publish-subscribe) that require non-linear variable use.

Co-contextual typing. Co-contextual typing (Erdweg et al. 2015) was originally introduced to support efficient incremental typechecking, and has also been used to support intrinsically typed compilation (Rouvoet et al. 2021). Padovani (2014b) 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 to the best of our knowledge we are the 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 support *correct-by-construction* communication-centric software, but are difficult to apply to actor languages. We have developed the *first* language design incorporating *mailbox types*, which characterise communication in actor languages and can statically detect communication errors that would otherwise be difficult to locate and fix. The many-sender, single-receiver nature of actor communication makes the integration of mailbox types in programming languages 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). Pat can encode common communication patterns (*e.g.*, master-worker), and we have shown that it can also express a real-world case study based on factory automation.

Ongoing and future work. Mailbox typing is a young field and there are many areas that are ripe for exploration.

We are currently investigating using mailbox types to verify communication behaviour in mainstream actor languages such as Erlang. In Pat a process may have multiple mailboxes and *explicitly* creates and destroys each mailbox. In contrast, mainstream actor languages *implicitly* create a single *monolithic* mailbox that holds messages from all protocols. Our approach overlays multiple virtual mailboxes on a monolithic mailbox, and annotates the code to indicate what messages are expected by **receive**s and when a mailbox should be created or reused.

An important area for future work is better inference: both at the level of mailbox patterns in order to allow developers to elide annotations on **guard** expressions, and at the level of types to support more interesting type system features (*e.g.*, polymorphism or set-theoretic typing). For mailbox types to be adopted in practice, it is also important to consider how mailbox types can be adapted to handle failure.

Another avenue for future work is finer-grained deadlock and alias detection. Quasi-linearity provides some guarantees, but since we cannot easily adopt the dependency graph formalism introduced by de'Liguoro and Padovani (2018), we cannot guarantee inter-process deadlock-freedom. Approaches such as *priorities* (Kobayashi 2006; Kokke and Dardha 2023; Padovani 2014a) may prove a useful starting point, but it is not yet clear how to adapt these to the many-sender, single-receiver communication model.

This paper has concentrated on the design and implementation of a *typechecker* for Pat. We are currently working on implementing an interpreter for Pat's semantics, which is nontrivial due to features like the **free** guard. Our approach uses an adaptation of the Perceus reference counting algorithm (Reinking et al. 2021).

Acknowledgements

We are deeply grateful to the JFP reviewers for their extremely thorough reading of the paper and their insightful suggestions, which we believe have significantly improved the final version. We are also grateful to the ICFP'23 reviewers and Artifact Evaluation Committee for their helpful comments on a previous version of this paper. Thanks also to our STARDUST colleagues for many interesting discussions; to Roland Kuhn for discussion of the case study; and to Edgard Schiebelbein for initial work mechanising Pat (Schiebelbein et al. 2026) that

resulted in several significant technical improvements. This work was supported by EPSRC Grant EP/T014628/1 (STARDUST).

Declaration of competing interests

The authors have no competing interests to declare.

References

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

Akka Team. 2026. *Introduction to Actors - Akka Core*. https://doc.akka.io/libraries/akka-core/current/typed/actors.html

Bernardo Almeida, Andreia Mordido, Peter Thiemann, and Vasco T. Vasconcelos. 2022. Polymorphic lambda calculus with context-free session types. *Information and Computation* 289, Part A (2022), 104948. doi:10.1016/j.ic.2022.104948

Roberto M. Amadio, Ilaria Castellani, and Davide Sangiorgi. 1998. On Bisimulations for the Asynchronous pi-Calculus. *Theoretical Computer Science* 195, 2 (1998), 291–324. doi: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. *Foundations and Trends in Programming Languages* 3, 2-3 (2016), 95–230. doi: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. doi: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 (Lecture Notes in Computer Science, Vol. 7907)*. Springer, 118–130. doi:10.1007/978-3-642-38771-5_12

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

Avik Chaudhuri. 2009. A Concurrent ML library in Concurrent Haskell. In *ICFP*. ACM, 269–280. doi: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. doi:10.1007/978-3-642-18378-2_3

Luca Ciccone and Luca Padovani. 2022. Fair termination of binary sessions. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 1–30. doi:10.1145/3498666

Silvia Crafa and Luca Padovani. 2017. The Chemical Approach to Typestate-Oriented Programming. *ACM Transactions on Programming Languages and Systems* 39, 3 (2017), 13:1–13:45. doi: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. doi: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. doi: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. doi:10.4230/LIPIcs.ECOOP.2018.15

Jay L. Devore and Kenneth N. Berk. 2012. *Modern Mathematical Statistics with Applications*. Springer.

Edsger W. Dijkstra. 1968. *Cooperating Sequential Processes*. Springer New York, New York, NY, 65–138. doi:10.1007/978-1-4757-3472-0_2

Jana Dunfield and Neel Krishnaswami. 2022. Bidirectional Typing. *Comput. Surveys* 54, 5 (2022), 98:1–98:38. doi: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. doi: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. doi:10.1145/2814270.2814277

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

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

Simon Fowler, Duncan Paul Attard, Danielle Marshall, Simon J. Gay, and Phil Trinder. 2026. Special Delivery: Programming with Mailbox Types (Extended Version). arXiv:2306.12935 [cs.PL]

Simon Fowler, Duncan Paul Attard, Franciszek Sowul, Simon J. Gay, and Phil Trinder. 2023a. Special Delivery: Programming with Mailbox Types. *Proceedings of the ACM on Programming Languages* 7, ICFP (2023), 78–107. doi:10.1145/3607832

Simon Fowler and Raymond Hu. 2026. Speak Now: Safe Actor Programming with Multiparty Session Types. *Proceedings of the ACM on Programming Languages* 10, OOPSLA1, Article 159 (April 2026), 28 pages. doi:10.1145/3798267

Simon Fowler, Wen Kokke, Ornela Dardha, Sam Lindley, and J. Garrett Morris. 2023b. Separating Sessions Smoothly. *Logical Methods in Computer Science* 19, 3 (2023). doi: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. doi:10.4230/LIPIcs.ECOOP.2017.11

Simon J. Gay and Vasco Thudichum Vasconcelos. 2010. Linear type theory for asynchronous session types. *Journal of Functional Programming* 20, 1 (2010), 19–50. doi:10.1017/S0956796809990268

Simon J. Gay and Vasco T. Vasconcelos. 2025. *Session Types*. Cambridge University Press.

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

Seymour Ginsburg and Edwin Spanier. 1966. Semigroups, Presburger formulas, and languages. *Pacific J. Math.* 16, 2 (1966), 285–296. doi:10.2140/pjm.1966.16.285

Joseph R. Harrison. 2018. Automatic detection of core Erlang message passing errors. In *Erlang Workshop*. ACM, 37–48. doi: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. doi:10.4230/LIPIcs.ECOOP.2021.10

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

Kohei Honda. 1993. Types for Dyadic Interaction. In *CONCUR (Lecture Notes in Computer Science, Vol. 715)*. Springer, 509–523. doi: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. doi:10.1007/BFb0053567

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

Mark W. Hopkins and Dexter Kozen. 1999. Parikh's Theorem in Commutative Kleene Algebra. In *LICS*. IEEE Computer Society, 394–401. doi: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. doi:10.1007/978-3-662-54494-5_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. doi: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. *Comput. Surveys* 49, 1 (2016), 3:1–3:36. doi: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. doi: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)*. Springer, 296–312. doi:10.1007/978-3-319-47846-3_19

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

Naoki Kobayashi. 2006. A New Type System for Deadlock-Free Processes. In *CONCUR (Lecture Notes in Computer Science, Vol. 4137)*. Springer, 233–247. doi:10.1007/11817949_16

Wen Kokke and Ornela Dardha. 2023. Prioritise the Best Variation. *Logical Methods in Computer Science* 19, 4 (2023). doi:10.46298/lmcs-19(4:28)2023

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. doi: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. doi: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. doi: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. doi: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. doi:10.1007/978-3-642-21464-6_7

Matthias Neubauer and Peter Thiemann. 2004. An Implementation of Session Types. In *PADL (Lecture Notes in Computer Science, Vol. 3057)*. Springer, 56–70.

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

Rumyana Neykova and Nobuko Yoshida. 2017b. Multiparty Session Actors. *Logical Methods in Computer Science* 13, 1 (2017). doi: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. doi:10.1145/2983990.2984009

Luca Padovani. 2014a. Deadlock and lock freedom in the linear π-calculus. In *CSL-LICS*. ACM, 72:1–72:10. doi:10.1145/2603088.2603116

Luca Padovani. 2014b. 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. doi:10.1007/978-3-642-54830-7_6

Luca Padovani. 2018a. Deadlock-Free Typestate-Oriented Programming. *The Art, Science, and Engineering of Programming* 2, 3 (2018), 15. doi: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. doi:10.1016/j.jlamp.2018.06.001

Luca Padovani. 2019. Context-Free Session Type Inference. *ACM Transactions on Programming Languages and Systems* 41, 2 (2019), 9:1–9:37. doi:10.1145/3229062

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

Benjamin C. Pierce and David N. Turner. 2000. Local type inference. *ACM Transactions on Programming Languages and Systems* 22, 1 (2000), 1–44. doi: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. doi:10.1007/BFb0055063

Alex Reinking, Ningning Xie, Leonardo de Moura, and Daan Leijen. 2021. Perceus: garbage free reference counting with reuse. In *PLDI*. ACM, 96–111.

Arjen Rouvoet, Robbert Krebbers, and Eelco Visser. 2021. Intrinsically typed compilation with nameless labels. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–28. doi:10.1145/3434303

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

Edgard Schiebelbein, Annette Bieniusa, and Simon Fowler. 2026. Proof of Delivery: Mechanized Mailbox Types. In *COORDINATION (Lecture Notes in Computer Science, Vol. 16590)*. Springer, 195–215.

Gerard Tabone and Adrian Francalanza. 2021. Session types in Elixir. In *AGERE!@SPLASH*. ACM, 12–23. doi: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. doi: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. doi: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. doi: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, Francesco Cesarini, Maurizio Di Stefano, Sverker Eriksson, Viktória Fördős, Amir Ghaffari, Aggelos Giantsios, Rickard Green, Csaba Hoch, David Klaftenegger, Huiqing Li, Kenneth Lundin, Kenneth MacKenzie, Katerina Roukounaki, Yiannis Tsiouris, and Kjell Winblad. 2017. Scaling reliably: Improving the scalability of the Erlang distributed actor platform. *ACM Transactions on Programming Languages and Systems* 39, 4 (2017), 1–46. doi:10.1145/3107937

Vasco T. Vasconcelos. 2012. Fundamentals of session types. *Information and Computation* 217 (2012), 52–70. doi:10.1016/j.ic.2012.05.002

Vasco Thudichum Vasconcelos, Simon J. Gay, and António Ravara. 2006. Type checking a multithreaded functional language with session types. *Theoretical Computer Science* 368, 1-2 (2006), 64–87.

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

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