Proof of Delivery: Mechanized Mailbox Types

Edgard Schiebelbein[1 [0009−0002−9476−5727]],
Annette Bieniusa[1 [0000−0002−1654−6118]], and Simon Fowler[2 [0000−0001−5143−5475]]

[1] RPTU University Kaiserslautern-Landau, Kaiserslautern, Germany
{edgard.schiebelbein,annette.bieniusa}@cs.rptu.de
[2] University of Glasgow, Glasgow, UK
simon.fowler@glasgow.ac.uk

Abstract. Programming languages based on the actor model such as Erlang and Elixir rely on message passing for communication. A core concept of this communication model are mailboxes: all actors are allowed to send messages to a mailbox, but only its owner may retrieve them. A major challenge in such systems is detecting and eliminating protocol violations and deadlocks. Mailbox types are a novel type system that enables static reasoning about the contents of mailboxes, but the metatheory of mailbox type systems is complex and requires extensive reasoning about subtyping. This paper establishes a machine-checked foundation for mailbox types. As a basis for this, we formalize Pat, the first programming language to use mailbox types, in Rocq. With its help, we identified and corrected several oversights in the original definitions. Furthermore, we provide mechanized proofs of several properties of the semantics of mailbox types and the substitution lemma for Pat.

Keywords: Mailbox types · Behavioural type systems · Formal verification · Actor languages

1 Introduction

Programming concurrent and distributed systems is notoriously difficult. Communication-centric programming languages such as Erlang or Elixir, and frameworks like MPI, are based around *message passing* concurrency, where processes send and receive messages to coordinate computation, rather than relying on shared memory. These actor languages have seen much industrial interest due to their suitability for distribution and compatibility with failure recovery idioms such as supervision hierarchies [2]. However, these languages are also susceptible to issues like communication mismatches and deadlocks which pose significant problems at runtime, and are also difficult to detect and fix.

Mailbox types [6,9] are a novel behavioural type system with a focus on mailboxes, as used in actor languages like Erlang and Elixir. Mailboxes are incoming message queues: multiple processes can *write* to a mailbox, but only the process that owns a mailbox can read from it.

As an example of mailbox typing, consider the following Erlang implementation of a *future variable* [9], which is a write-once placeholder variable.

```
1  empty_future() →
2    receive
3      {put, X} → full_future(X)
4    end.
5  full_future (X) →
6    receive
7      {get, Pid} → Pid ! {reply, X},
8        full_future (X);
9      {put, _} →
10     erlang:error("Multiple writes")
11   end.
```

```
12 client () →
13   Future = spawn(future, empty_future, []),
14   Future ! {put, 5},
15   Future ! {get, self()},
16   receive
17     {reply, Result} →
18       io:fwrite("~w~n", [Result])
19   end.
```

The process empty_future waits to receive a put message containing the value to hold. This is a *selective receive*: other messages may be present in the mailbox but they will not be retrieved. After the put message has been received, the process then calls full_future with the provided value X. In full_future the process waits to receive get messages containing the process ID of the requesting actor, and responds by sending value X to the sender. Since a future variable should only be set once, the process throws an error if more put messages are received. Even in small examples such as this, protocol violations such as unexpected messages, forgotten replies and self-deadlocks may occur. All of these problems can be addressed and statically detected using mailbox typing [6,9].

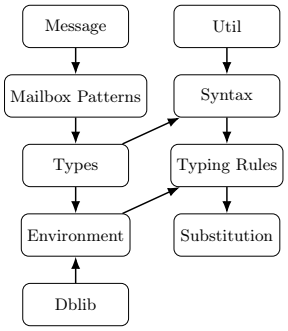
A mailbox type consists of a *capability* and a *pattern*. The capability indicates whether the type describes messages that must be sent (!) or describe messages to be received from a mailbox (?). A pattern is a commutative regular expression that describes the contents of a mailbox. For the above example, we can define the following mailbox types:

$$\mathsf{EmptyFuture} \triangleq ?(\mathsf{Put}[\mathsf{Int}] \odot \mathsf{Get}[!\mathsf{Reply}[\mathsf{Int}]]^\star) \qquad \mathsf{FullFuture} \triangleq ?\mathsf{Get}[!\mathsf{Reply}[\mathsf{Int}]]^\star$$

Type EmptyFuture describes a mailbox that may contain a single Put message with payload of type Int, and a potentially arbitrary amount of Get messages. Each Get message contains a mailbox reference that expects a Reply message to be sent. Type FullFuture is more restrictive in that it describes a mailbox only containing an arbitrary amount of Reply messages. By using these types, the type system can statically require that exactly one Put message is sent, and can ensure that every Get request receives a Reply message.

Of course, a type system can only guarantee correctness if its own specification is correct. It is therefore crucial to rigorously prove the properties a type system gives. Verifying that these proofs are indeed correct can be challenging in itself. Errors or oversights can remain undetected, resulting in false promises about type checked programs. For example, a mechanization of multiparty session typing [13] uncovered errors nearly a decade later [22].

In this paper, we report on our mechanization of the semantics of mailbox types, and of the *Pat* [9] programming language that incorporates mailbox types. Although Pat includes extensive pen-and-paper proofs of correctness, the proofs have not been mechanized. In particular, we have mechanized numerous properties about mailbox types themselves, and have mechanized the static semantics



Module	LOC	Definitions	Lemmas
Message	177	7	9
Mailbox Patterns	828	8	44
Types	695	35	30
Environment	3142	16	131
Util	48	6	2
Syntax	180	16	0
Typing Rules	317	4	12
Substitution	1325	2	28
	6712	94	256

Fig. 1. Dependency graph of the modules in the Rocq project. Transitive dependencies are omitted.

of Pat. During our mechanization, we discovered several oversights in the original definitions and proofs which have since been fixed.

We use the Rocq [20] proof assistant for our mechanization. Figure 1 summarizes the structure of our Rocq project and shows the lines of code, number of definitions and lemmas for each file. Due to the necessity of *type combination* and *environment subtyping*, the `Environment` module contains, by far, the largest number of lemmas.

Throughout this paper, definitions and lemmas that are accompanied with the -symbol link to the corresponding code in the online documentation. The project can be found at https://github.com/edgardSchi/mailbox-types-rocq.

Methodology. Our mechanization includes both the syntax and type system of Pat, and prove several properties of the language such as the substitution lemma. We limit the scope of this paper to Pat's static semantics, which alone requires a significant mechanization effort; a full account of Pat's operational semantics and preservation theorem would require significant additional work and is left as a future direction. We concentrate on properties that are needed for proving preservation, but also do not require a description of Pat's operational semantics.

We use Rocq due to its maturity and active development: several successful mechanizations have been conducted in Rocq, such as *CompCert* [15], the first fully formally verified real-world compiler, and formalizations of session types [22] and process calculi [1], modelling communication between processes.

Contributions. The main contributions of this paper are as follows:

- The first mechanization of the syntax and semantics of mailbox types (§2).
- A formalization of the static semantics of Pat, including definitions that are more amenable to mechanization (§3).
- Proofs of metatheoretical properties of the static semantics, including the nontrivial substitution lemma (§3), requiring over 200 auxiliary lemmas.
- We identify missing cases in the original definitions and places where the original proofs could be improved, that have since been fixed (§2 and §3).

Mailbox configurations (🏴)	m	$::=$	$\langle\rangle \mid \langle \mathsf{m} \rangle \mid m_1 \uplus m_2$	Pattern power (🏴)	
Mailbox patterns (🏴)	E, F	$::=$	$\mathbb{0} \mid \mathbb{1} \mid \langle\!\langle \mathsf{m} \rangle\!\rangle \mid E \odot F$	E^0	$:= \mathbb{1}$
		$\mid$	$E \oplus F \mid E^\star$	E^{n+1}	$:= E \odot E^n$

Fig. 2. The syntax of mailbox configurations and patterns.

2 Mailbox Types

We begin by describing mailbox configurations and mailbox patterns, before formalising mailbox types along with the necessary notion of type combination. Due to the potentially infinite representation of mailbox semantics, we require novel definitions that are more amenable to mechanization.

2.1 Mailboxes and Mailbox Patterns

Like Pat but unlike in the original definition of mailbox types [6], we do not include message payloads as part of a mailbox type, but rely only on *message tags* m like Put and Get and associate types with messages when defining the typing rules for Pat (§3.5).

Mailbox Configurations. A mailbox represents an unordered collection of messages, where a message can be included multiple times. Figure 2 shows the definition of a *mailbox configuration* m. It is either the empty mailbox $\langle\rangle$, a singleton mailbox $\langle \mathsf{m} \rangle$ containing only message m, or the union of two mailboxes $m_1 \uplus m_2$. In Rocq, we model mailbox configurations as lists, where $\uplus$ is implemented as append. To reason about the equality of mailboxes, i.e. mailboxes with the same content, we need to take into account that order does not matter. Mailbox configurations m_1 and m_2 are equal, written $m_1 \approx m_2$, if they are permutations of each other (🏴). We then have $\langle \mathsf{m} \rangle \uplus \langle \mathsf{m} \rangle \uplus \langle \mathsf{n} \rangle \approx \langle \mathsf{m} \rangle \uplus \langle \mathsf{n} \rangle \uplus \langle \mathsf{m} \rangle$, but $\langle \mathsf{n} \rangle \uplus \langle \mathsf{n} \rangle \not\approx \langle \mathsf{n} \rangle$.

Mailbox Patterns. The crucial building block of mailbox types are *mailbox patterns*. They can be thought of as a commutative regular expression that describes the content of a mailbox. Figure 2 shows the definition of mailbox patterns. The pattern $\mathbb{0}$ denotes an *unreliable* mailbox, i.e. one that received an unexpected message and which cannot be used for further communication. Pat's type system ensures that it is not possible to create such a mailbox using well-typed terms. Pattern $\mathbb{1}$ represents an *empty* mailbox, containing zero messages. The pattern $\langle\!\langle \mathsf{m} \rangle\!\rangle$ denotes a mailbox containing a *single message* m. Mailbox *composition* is described by pattern $E \odot F$, combining both patterns E and F. As an example, pattern $\langle\!\langle \mathsf{m} \rangle\!\rangle \odot \langle\!\langle \mathsf{n} \rangle\!\rangle$ would describe a mailbox containing both m and n. On the other hand, $E \oplus F$ denotes *choice* between patterns. In particular, it describes that the mailbox is either described by pattern E or by pattern F. Lastly, pattern $E^\star$ describes repeated composition of pattern E. For example, $\langle\!\langle \mathsf{m} \rangle\!\rangle^\star$ denotes a mailbox containing zero or more m messages.

$$\frac{}{\langle\rangle \in \mathbb{1}} \qquad \frac{}{\langle \mathsf{m} \rangle \in \langle\!\langle \mathsf{m} \rangle\!\rangle} \qquad \frac{m \in E}{m \in E \oplus F} \qquad \frac{m \in F}{m \in E \oplus F} \qquad \frac{\exists n.\ m \in E^n}{m \in E^\star} \qquad \frac{m \approx m_1 \uplus m_2 \quad m_1 \in E \quad m_2 \in F}{m \in E \odot F}$$

Fig. 3. The semantics of mailbox patterns (🏴‍☠️).

Pattern Semantics. Since patterns describe the contents of an unordered mailbox, multiple patterns can describe the same mailbox. For example, pattern $\langle\!\langle \mathsf{m} \rangle\!\rangle \odot \langle\!\langle \mathsf{n} \rangle\!\rangle$ describes the same mailbox as $\langle\!\langle \mathsf{n} \rangle\!\rangle \odot \langle\!\langle \mathsf{m} \rangle\!\rangle$. Thus, checking equality of mailbox patterns requires a semantic definition. The semantics of mailbox types are usually given by a denotational semantics that map patterns to sets of multisets of atoms [6]. Since we are working with Rocq, which is based on constructive logic, we require a method of constructing these sets; a particular challenge is representing the $E^\star$ pattern whose denotation is an infinite union of multisets and not easily representable in Rocq. Instead, we use a relational approach to connect mailbox configurations to patterns.

Instead of building possibly infinite sets of possible configurations, we relate a mailbox configuration to a pattern, written $m \in E$. A configuration is included in the set induced by a pattern if the configuration can be constructed using the pattern. For example, we have $\langle \mathsf{m} \rangle \uplus \langle \mathsf{n} \rangle \in \langle\!\langle \mathsf{m} \rangle\!\rangle \odot \langle\!\langle \mathsf{n} \rangle\!\rangle$ and $\langle \mathsf{n} \rangle \uplus \langle \mathsf{m} \rangle \in \langle\!\langle \mathsf{m} \rangle\!\rangle \odot \langle\!\langle \mathsf{n} \rangle\!\rangle$, but also $\langle \mathsf{m} \rangle \uplus \langle \mathsf{n} \rangle \in \langle\!\langle \mathsf{m} \rangle\!\rangle \odot \langle\!\langle \mathsf{n} \rangle\!\rangle \odot \mathbb{1}$.

The inference rules for this relation are given in Figure 3. The first two rules describe that the empty mailbox configuration $\langle\rangle$ is the pattern $\mathbb{1}$ and a configuration containing a single message m is part of pattern $\langle\!\langle \mathsf{m} \rangle\!\rangle$. For the choice operator $E \oplus F$, there are two rules: one for when m is part of E and one for when m is part of F. The rule for composition is the most complex one. If m is part of a composition of patterns E and F, then m is required to be split into two sub-configurations, where one is part of E and one is part of F. Lastly, the rule for $E^\star$ uses the *power* operation (defined in Figure 2) to state that any mailbox configuration that can be built using a pattern E^n (for some n) is included in $E^\star$.

One important observation to make is that we do not include a rule for pattern $\mathbb{0}$. According to the definition of de'Liguoro and Padovani [6], $\mathbb{0}$ should map to the empty set, containing no configurations. Thus, since we are relating configurations and patterns, there must be no relation between any configuration and $\mathbb{0}$.

We define *pattern inclusion* $E \sqsubseteq F$ as the proposition $\forall m.\ m \in E \Rightarrow m \in F$ (🏴‍☠️). Taking inclusion in both directions yields pattern *equivalence* $E \cong F$ (🏴‍☠️).

To verify this formalization has the required properties described by de'Liguoro and Padovani [6], we prove the following properties:

Lemma 1. *The relation $\sqsubseteq$ is a precongruence, i.e. the following properties hold:*

- *Reflexive (🏴‍☠️): $E \sqsubseteq E$*
- *Transitive (🏴‍☠️): if $E \sqsubseteq F$ and $F \sqsubseteq G$, then $E \sqsubseteq G$*

Pattern residuals (🏴‍☠️) $\boxed{E \setminus \mathsf{m} \triangleq F}$

$$\frac{}{\mathbb{0} \setminus \mathsf{m} \triangleq \mathbb{0}} \qquad \frac{}{\mathbb{1} \setminus \mathsf{m} \triangleq \mathbb{0}} \qquad \frac{}{\langle\!\langle \mathsf{m} \rangle\!\rangle \setminus \mathsf{m} \triangleq \mathbb{1}} \qquad \frac{\mathsf{m} \neq \mathsf{n}}{\langle\!\langle \mathsf{m} \rangle\!\rangle \setminus \mathsf{n} \triangleq \mathbb{0}} \qquad \frac{E \setminus \mathsf{m} \triangleq E'}{E^\star \setminus \mathsf{m} \triangleq E' \odot E^\star}$$

$$\frac{E \setminus \mathsf{m} \triangleq E' \qquad F \setminus \mathsf{m} \triangleq F'}{E \oplus F \setminus \mathsf{m} \triangleq E' \oplus F'} \qquad \frac{E \setminus \mathsf{m} \triangleq E' \qquad F \setminus \mathsf{m} \triangleq F'}{E \odot F \setminus \mathsf{m} \triangleq (E' \odot F) \oplus (E \odot F')}$$

Fig. 4. The definition of pattern residuals.

- *Compatible wrt.* $\odot$ *(🏴‍☠️): if* $E_1 \sqsubseteq F_1$ *and* $E_2 \sqsubseteq F_2$*, then* $(E_1 \odot E_2) \sqsubseteq (F_1 \odot F_2)$
- *Compatible wrt.* $\oplus$ *(🏴‍☠️): if* $E_1 \sqsubseteq F_1$ *and* $E_2 \sqsubseteq F_2$*, then* $(E_1 \oplus E_2) \sqsubseteq (F_1 \oplus F_2)$

Lemma 2. *The relation* $\cong$ *is an equivalence relation, i.e. it is*

- *Reflexive (🏴‍☠️):* $E \cong E$
- *Transitive (🏴‍☠️): if* $E \cong F$ *and* $F \cong G$*, then* $E \cong G$
- *Symmetric (🏴‍☠️): if* $E \cong F$*, then* $F \cong E$

Additionally, we show that we indeed work with a commutative Kleene algebra $(M, \oplus, \odot, ^\star, \mathbb{0}, \mathbb{1})$, where M is the set of messages (🏴‍☠️).

Pattern Residuals. Mailboxes are dynamic objects, not static. Their content changes dynamically while receiving and consuming messages. For example, when consuming message m from a mailbox described by pattern $\langle\!\langle \mathsf{m} \rangle\!\rangle$ the resulting mailbox should be empty, described by pattern $\mathbb{1}$. To model this behavior, de'Liguoro and Padovani introduce the concept of *pattern residuals* [6] (Figure 4). In Pat [10], pattern residuals correspond exactly to the Brzozowski derivative of a commutative regular expression [3]. A pattern residual describes the resulting pattern when a message is consumed, i.e. removed from the mailbox.

An important property of residuals is *balancing*. For a configuration described by pattern $\langle\!\langle \mathsf{m} \rangle\!\rangle \odot E$, which is included in a pattern F, E should be included in the residual $F \setminus \mathsf{m} \triangleq F'$. This property is needed for reasoning about the type of a mailbox after a message has been received.

Lemma 3 (Balancing (🏴‍☠️)). *If* $\langle\!\langle \mathsf{m} \rangle\!\rangle \odot F \sqsubseteq E$ *where* $F \not\sqsubseteq \mathbb{0}$ *and* $E \setminus \mathsf{m} \triangleq E'$*, then* $F \sqsubseteq E'$*.*

Mechanization. Mailbox configurations mostly rely on the definitions of list permutations already provided by Rocq's standard library, but the formalization of mailbox patterns requires more effort. Describing the semantics of mailbox patterns as a relation between a configuration and a pattern, instead of possibly infinite multisets, yields a mechanization that is adequate to work with. The proofs about pattern equivalence and inclusion are typically inductions over

Mailbox types (🏴)	J, K	::=	$?E \mid !E$
Base types (🏴)	C	::=	$\mathbf{1} \mid \mathsf{Bool}$
Types (🏴)	T, U	::=	$C \mid J$

Fig. 5. The syntax of types.

a pattern, but proving balancing (Lemma 3) requires several technical lemmas that rely on proven properties about permutations of mailbox configurations.

When mechanizing the pattern residual relation, we discovered that in the definition of Pat [9] the rule for $\star$ was missing. This issue was consequently fixed in the journal version [8].

2.2 Types

A mailbox type (Figure 5) consists of a capability and a pattern. A capability is either *input* ? or *output* !. Together, capability and pattern describe what messages are expected to be received from (?) or stored in (!) the mailbox. To build an intuition for mailbox types, consider the following examples: A mailbox with mailbox type $?\langle\!\langle \mathsf{m} \rangle\!\rangle$ gives a process the right to receive a message m from that mailbox, while a mailbox with type $!\langle\!\langle \mathsf{m} \rangle\!\rangle$ requires the process to send a message m to that mailbox. For a mailbox with mailbox type $!(\langle\!\langle \mathsf{m} \rangle\!\rangle \oplus \langle\!\langle \mathsf{n} \rangle\!\rangle)$, a process may choose whether to send message m or n, whereas a mailbox with type $?(\langle\!\langle \mathsf{m} \rangle\!\rangle \oplus \langle\!\langle \mathsf{n} \rangle\!\rangle)$ allows a process to receive either message m or n from that mailbox. The type $?(\langle\!\langle \mathsf{m} \rangle\!\rangle \odot \langle\!\langle \mathsf{n} \rangle\!\rangle)$ is associated to a mailbox that allows a process to receive both m and n, while a mailbox with mailbox type $!(\langle\!\langle \mathsf{m} \rangle\!\rangle \odot \langle\!\langle \mathsf{n} \rangle\!\rangle)$ requires a process to send both m and n to the mailbox, in whichever order. Finally, a mailbox with type $!(\langle\!\langle \mathsf{m} \rangle\!\rangle^\star)$ allows a process to send an arbitrary number of m messages to the mailbox, while $?(\langle\!\langle \mathsf{m} \rangle\!\rangle^\star)$ describes a mailbox from which a process is able to receive an arbitrary number of m messages.

In addition to mailbox types, we also include *base types*. While the unit type $\mathbf{1}$ is required by the type system, additional base types can be added. For demonstration purposes we also include a Boolean type Bool. A type T is then either some base type or a mailbox type.

Type Combination. To model the dynamic nature of mailboxes receiving and sending messages, mailbox types should be able to be combined to create a new mailbox type. Mailbox types need to ensure that sends and receives are matched, i.e. a send is required to have a corresponding receive. For example, given mailbox types $!\langle\!\langle \mathsf{m} \rangle\!\rangle$ and $?(\langle\!\langle \mathsf{m} \rangle\!\rangle \odot \langle\!\langle \mathsf{n} \rangle\!\rangle)$, the resulting type from the type combination should be $?\langle\!\langle \mathsf{n} \rangle\!\rangle$.

In the original definitions of type combinations [6,9], types are identified up to commutativity and associativity. In practice, this means that types such as $!(\langle\!\langle \mathsf{m} \rangle\!\rangle \odot \langle\!\langle \mathsf{n} \rangle\!\rangle)$ and $!(\langle\!\langle \mathsf{n} \rangle\!\rangle \odot \langle\!\langle \mathsf{m} \rangle\!\rangle)$ are considered to be equivalent. While this is true from a semantic point of view, syntactically these are different types and assuming equality up to a certain property creates problems when mechanizing

Pattern permutation ($\curvearrowright$) $\boxed{E \sim F}$ **Type permutation** ($\curvearrowright$) $\boxed{T \sim U}$

$$\frac{}{(E \odot F) \sim (F \odot E)} \qquad \frac{F \sim F'}{E \odot F \sim E \odot F'} \qquad \frac{}{C \sim C} \qquad \frac{E \sim F}{!E \sim !F}$$

$$\frac{}{E \sim E} \qquad \frac{F \sim E}{E \sim F} \qquad \frac{E \sim F' \qquad F' \sim F}{E \sim F} \qquad \frac{E \sim F}{?E \sim ?F}$$

$$\frac{}{E \odot (F \odot F') \sim (E \odot F) \odot F'}$$

Type combinations ($\curvearrowright$) $\boxed{T \boxplus U = T'}$

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

$$\frac{}{?(E \odot F) \boxplus !E = ?F} \qquad \frac{T_1' \boxplus T_2' = T_3' \qquad T_1 \sim T_1' \qquad T_2 \sim T_2' \qquad T_3 \sim T_3'}{T_1 \boxplus T_2 = T_3}$$

Fig. 6. The definitions of pattern and type permutations, as well as type combinations.

such definitions. We require a way to express equivalence up to commutativity and associativity in the context of type combinations. To this end, we define a relation between mailbox types, holding if they are permutations of each other.

The relation $E \sim F$ (Figure 6) is expressing that two patterns can be rewritten into each other using commutativity and associativity of the $\odot$ operator. Note how these rules do not include the $\oplus$ operator. This is because type combination only considers type composition, and it is not required to include pattern choice in the definition of $\sim$ to achieve the desired properties. Note the difference between the relations $E \sim F$ and $E \cong F$. The former is expressing that E and F can be rewritten into each other by swapping around patterns that appear between the composition operator. The latter relates two semantically equal patterns. For example, consider pattern $\langle\!\langle \mathsf{m} \rangle\!\rangle \odot \langle\!\langle \mathsf{n} \rangle\!\rangle$. Both $\langle\!\langle \mathsf{m} \rangle\!\rangle \odot \mathbb{1} \sim \mathbb{1} \odot \langle\!\langle \mathsf{m} \rangle\!\rangle$ and $\langle\!\langle \mathsf{m} \rangle\!\rangle \odot \mathbb{1} \cong \mathbb{1} \odot \langle\!\langle \mathsf{m} \rangle\!\rangle$ hold, while $\langle\!\langle \mathsf{m} \rangle\!\rangle \odot \mathbb{1} \cong \langle\!\langle \mathsf{m} \rangle\!\rangle$, but $\langle\!\langle \mathsf{m} \rangle\!\rangle \odot \mathbb{1} \not\sim \langle\!\langle \mathsf{m} \rangle\!\rangle$.

The relation $\sim$ is extended to mailbox types by relating the underlying patterns. Base types are always considered to be permutations of each other.

Type combinations model the concurrent interaction between processes acting on the same mailbox. For example, assume processes P_1 and P_2 that both interact with mailbox m. Process P_1 stores some message m, while P_2 stores some message n. Then, mailbox m can be described by the type combination $!\langle\!\langle \mathsf{m} \rangle\!\rangle \boxplus !\langle\!\langle \mathsf{n} \rangle\!\rangle = !(\langle\!\langle \mathsf{m} \rangle\!\rangle \odot \langle\!\langle \mathsf{n} \rangle\!\rangle)$. If instead, we have type $?(\langle\!\langle \mathsf{m} \rangle\!\rangle \odot \langle\!\langle \mathsf{n} \rangle\!\rangle)$, then we get the type combination $!\langle\!\langle \mathsf{m} \rangle\!\rangle \boxplus ?(\langle\!\langle \mathsf{m} \rangle\!\rangle \odot \langle\!\langle \mathsf{n} \rangle\!\rangle) = ?\langle\!\langle \mathsf{n} \rangle\!\rangle$. Figure 7 shows a example with three mailbox types. There are different ways of arranging the combination, but in the end, the result always balances out to an empty mailbox.

Figure 6 shows the definition of the *type combination relation*. Note how there is no rule for type combination of two types with input capabilities. This

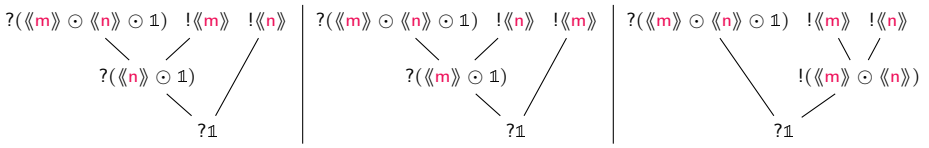


Fig. 7. Example of possible type combinations for three mailbox types. Each top node represents the type of the same mailbox in three different processes that run in parallel. Edges indicate the type combination relation.

would be allowing parallel reads of the same mailbox, which is not safe [6], and thus not permitted. The last rule in the definition of type combinations allows us to substitute types for their permutations. Note how the previous definition of pattern equality $E \cong F$ is specifically *not* used in the definition of type combination. The combination operation is a *syntactic* operation, i.e. we can only combine types that are *syntactically equal* to the forms specified by the type combination operation (up to commutativity and associativity). To combine other mailbox types, for example $!\langle\!\langle \mathsf{m} \rangle\!\rangle$ and $?(\langle\!\langle \mathsf{m} \rangle\!\rangle \oplus \langle\!\langle \mathsf{m} \rangle\!\rangle)$ we must firstly use *subtyping* to rewrite the latter mailbox type to $?(\langle\!\langle \mathsf{m} \rangle\!\rangle \odot \mathbb{1})$ before using the combination operator. We will discuss subtyping further in Section 3.

Mechanization. Our mechanization of mailbox types differs from the original formalizations in that we treat partial operations as relations, and include the notion of pattern and type permutations. This addition allows us to prove desired properties, such as commutativity of type combinations. Naturally, proofs about type combinations become more complex since we have to cover additional cases.

3 The Pat Programming Language

Pat [9] is the first programming language to support mailbox types. In contrast to languages like Erlang or Elixir, where mailboxes are implicit, Pat supports first-class explicit mailboxes. We can express the Future example in Pat as follows:

```
1  def emptyFuture(self : EmptyFuture) : 1 {
2    guard self : Put ⊙ Get* {
3      receive Put[x] from self ↦
4        fullFuture(self, x)
5    }
6  }
7  def fullFuture(self : FullFuture, value : Int) : 1 {
8    guard self : Get* {
9      free ↦ ()
10     receive Get[user] from self ↦
11       user!Reply[value];
12       fullFuture(self, value)
13   }
14 }
15 def client() : 1 {
16   let future = new in
17   spawn emptyFuture(future);
18   let self = new in
19   future!Put[5];
20   future!Get[self];
21   guard self : Reply {
22     receive Reply[result]
23   from self ↦
24     free self;
25     print(intToString(result))
26   }
27 }
```

To support dynamic mailbox creation, new mailboxes are created using the **new** keyword and bound to names using **let**-bindings. Pat programs can then

send to mailboxes (e.g., *future*!Put[5] sends a Put message with payload 5 to mailbox *future*) and receive from mailboxes using the **guard** keyword. A guard is always accompanied by a mailbox pattern, representing the mailbox's contents, and contains *guard clauses* including **receive** which receives a message from the mailbox and binds the payload and new mailbox name; and **free** which is invoked when a mailbox will not receive any more messages and can be deallocated.

A mechanization of the operational semantics of Pat is outside of the scope of this paper, but we give an outline of the semantics here for context. Pat's semantics makes use of a *frame stack* representation that simplifies specifying inductive invariants for the type preservation proof. There are two reduction relations: a deterministic β-reduction relation on functional terms (like **let**-bindings and function calls) that is entirely standard, and a nondeterministic reduction relation on a language of *configurations* that closely resemble processes in the π-calculus. The communication and concurrency constructs reduce as follows:

- Evaluating the **new** construct creates a fresh runtime mailbox name and returns it to the calling thread.
- Evaluating a message send $a!\mathsf{m}[v]$ returns the unit value to the calling thread and creates an *in-flight* message $a \leftarrow \mathsf{m}[v]$.
- Evaluating **spawn** t returns the unit value to the calling thread and evaluates t in a new process.
- The behaviour of a **guard** $a : E\ \{\overrightarrow{g}\}$ expression depends on the messages that have been sent to mailbox a along with the guards contained in $\overrightarrow{g}$:
  - If there are no messages sent to a; no references to a in other threads; and $\overrightarrow{g}$ contains a **free** $\mapsto t$ guard, then mailbox name a will be deleted and the thread will run computation t.
  - If there is a message $a \leftarrow \mathsf{m}[v]$ and $\overrightarrow{g}$ contains a clause **receive** $\mathsf{m} \mapsto t$, then the thread will run computation t with bindings for the received value and updated mailbox name.

A major problem in Pat is *mailbox name aliasing*, where multiple static names could be given to the same underlying mailbox name. This could be used to break type soundness. The Mailbox Calculus addresses this issue using a dependency graph, which can additionally rule out cyclic dependencies, but this approach does not scale to programming languages (see [9] for a detailed discussion). Instead, Pat uses *quasi-linear typing* [14,7], where types are equipped with *usage annotations*. If a type is defined as *returnable*, it is allowed to appear in the return type of an expression and be returned from an evaluation frame. Otherwise, if a type is defined as *second-class*, it can be used several times, but is never allowed to escape its scope. A returnable occurrence of a mailbox name must be its last lexical usage. Quasilinearity annotations, in combination with syntactic restrictions, eliminates aliasing issues and self-deadlocks.

The authors of Pat present both a *declarative* and an *algorithmic* version of their type system. The algorithmic version (which is needed because the declarative system makes use of environment subtyping and non-deterministic context splits) makes use of *bidirectional typing* and *constraint solving*, and is at the core of Pat's implementation. This paper focuses on the declarative type system.

Usage annotations ( ) $\eta ::= \circ \mid \bullet$

Usage-annotated types ( ) $A, B ::= C \mid J^{\eta}$

Type operations (, , ,)

$\lceil C \rceil = C \qquad \lfloor C \rfloor = C$

$\lceil J \rceil = J^{\circ} \qquad \lfloor J \rfloor = J^{\bullet}$

$\lceil J^{\eta} \rceil = J^{\circ} \qquad \lfloor J^{\eta} \rfloor = J^{\bullet}$

Usage-annotated type combinations ( ) $\boxed{A \rhd B = A'}$

$$\frac{}{C \rhd C = C} \qquad \frac{\eta_1 \rhd \eta_2 = \eta \qquad J_1 \boxplus J_2 = K}{J_1^{\eta_1} \rhd J_2^{\eta_2} = K^{\eta}}$$

Usage combinations ( ) $\boxed{\eta_1 \rhd \eta_2 = \eta_3}$

$\circ \rhd \circ = \circ \qquad \circ \rhd \bullet = \bullet$

Usage subtyping ( ) $\boxed{\eta_1 \leq \eta_2}$

$\eta \leq \eta \qquad \bullet \leq \circ$

Subtyping ( ) $\boxed{A \leq B}$

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

Fig. 8. Definitions of usage-annotated types.

3.1 Quasi-linear Mailbox Types

Pat uses quasi-linear typing [14,7] to solve the problem of name aliasing and cyclic dependency between mailboxes. To this end, every mailbox type is equipped with a *usage annotation*: either *second class* ($\circ$) or *returnable* ($\bullet$). Values of second class types can be used several times within the scope they are defined, however, they cannot leave that scope. On the other hand, values of returnable types can be let-bound and used as the subject of a guard, but must appear as the last lexical occurrence of the name. Base types are not equipped with usage, as they do not suffer from the aliasing problems.

Figure 8 shows definitions of usage annotations, usage-annotated types and relations on them. We also define operators $\lceil \cdot \rceil$ and $\lfloor \cdot \rfloor$ to turn types into second class and returnable respectively. Additionally, we define a mailbox type to be *unrestricted* (written $\mathsf{un}(A)$) ( ) if it is a base type, or $A = !\mathbb{1}^{\circ}$.

Usage-annotated Type Combination. We extend type combination to *usage-annotated type combination* by incorporating usage combination. For usage annotations, we have to obey the rule that a $\circ$ variable has to occur before a $\bullet$ variable. Otherwise, we would violate the property that any $\bullet$ variable appears as the final one in a process. Combining two $\bullet$ types is not allowed as there could otherwise be multiple instances of a returnable mailbox name per thread.

Subtyping. Subtyping is needed in mailbox typing to rewrite mailbox types so that they can be combined by the type combination operator. For mailbox types, subtyping is *covariant* for types with receive capability and *contravariant* for types with send capability [6]. Subtyping also allows us to treat returnable types as second-class types.

Definition names (🏴)	f
Function definitions (🏴)	$d ::= \textbf{def}\ \mathsf{f}(A) : B\ \{t\}$
Booleans	$b ::= \mathsf{true} \mid \mathsf{false}$
Values (🏴)	$v ::= b \mid () \mid \mathsf{Var}\ n$
Terms (🏴)	$t ::= v \mid \textbf{let}\ t_1\ \textbf{in}\ t_2 \mid \mathsf{f}(v) \mid \textbf{spawn}\ t$
	$\mid\ \textbf{new} \mid v_1!\mathsf{m}[v_2] \mid \textbf{guard}\ v : E\ \{\overrightarrow{g}\}$
Guards (🏴)	$g ::= \textbf{fail} \mid \textbf{free} \mapsto t \mid \textbf{receive}\ \mathsf{m} \mapsto t$
Environments (🏴)	$\Gamma ::= \emptyset \mid \bot, \Gamma \mid A, \Gamma$

Fig. 9. The syntax of Pat

3.2 Syntax

The Pat syntax we are working with is close to the original definitions by Fowler et al. [9], making use of de Bruijn indices [5] to represent variables. We use de Bruijn indices as they are well-studied for proof assistants, and in particular Rocq [1,19]. We make use of the Rocq library *dblib* [18] that automatically handles de Bruijn indices and operations on them such as shifting.

To ease the burden of proving properties about messages, we restrict messages to only carry a single payload, and consider only functions that take a single argument. We expect that extending our proofs to include arbitrary numbers of payloads and function arguments will not significantly affect the proofs.

Figure 9 shows our syntax of Pat. A value v is either a Boolean value, the unit value $()$, or a variable $\mathsf{Var}\ n$ represented as a de Bruijn index, with natural number n. Including variables into the definition of values is fairly nonstandard. The inspiration for this approach is the *fine-grain call-by-value* evaluation strategy [16], that makes a syntactic difference between values and computations. Values, including variables that can also represent mailbox names, are immutable and cannot be reduced, while other terms produce values and can be reduced further.

A term is either a value, a let-expression or a function application $\mathsf{f}(v)$ of function f on value v. Additionally, there are terms specifically for interacting with mailboxes and processes. The term $\textbf{spawn}\ t$ can be used to create a new process running term t. Dynamic mailbox name creation is done with the **new** keyword, generating a fresh mailbox name. Term $v_1!\mathsf{m}[v_2]$ sends message m with payload value v_2 to the mailbox with name v_1. A guard term $\textbf{guard}\ v : E\ \{\overrightarrow{g}\}$ asserts that the mailbox associated with name V has pattern E, and invokes some guard from the list of guards $\overrightarrow{g}$. Guards perform actions on a mailbox: the guard **fail** is triggered when a mailbox receives an unexpected message, and should be evaluated by a well-typed program. In contrast, guard $\textbf{free} \mapsto t$ is triggered when a mailbox is empty and not referenced elsewhere in the system; the corresponding mailbox is freed and we proceed with continuation t. Finally, the guard $\textbf{receive}\ \mathsf{m} \mapsto t$ is triggered when a mailbox contains the message m. The contents of the message are then available in the continuation t.

A Pat program $\mathcal{P} = (\mathcal{S}, \mathcal{D}, t_0)$ (🏴) is a tuple consisting of a signature $\mathcal{S}$ for mapping messages to their payload type, a mapping $\mathcal{D}$ from definition names to

function definitions, and an initial term t_0. For simplicity, given a program $\mathcal{P}$, we write $\mathcal{S}_\mathcal{P}$ for the signature, and $\mathcal{D}_\mathcal{P}$ for the function definitions.

Mechanization. The mechanization of the syntax of Pat is close to its pen-and-paper counterpart, with the exception of using de Bruijn indices. Defining the syntax poses no problem after settling on the representation of variables. While the choice of using de Bruijn indices is standard [1,22,19], it comes with the downside of extensive reasoning about indices and lifting. While *dblib* alleviates some of this burden, a substantial amount of manual work is still required.

Definition names are treated similar to message tags, being represented as an arbitrary type equipped decidable equality.

3.3 Environments

Pat's type system makes use of *environments* for keeping track of types of variables while type checking terms. The implementation of environments depends on how variables are represented. For traditional strings as variables, it is common to use a map, mapping variables to types [17]. In the case of de Bruijn indices, an environment is typically represented as a list of types where a variable indicates the position of its type in the list [17,23]. For Pat, the representation of an environment as a simple list of types is unfortunately not enough, due to its parallel fragment and use of quasilinearity. To illustrate the problem, consider the type derivation of a mailbox calculus term on the left-hand side:

$$\frac{A_1, \emptyset \vdash \mathsf{Var}\ 0 \qquad A_2, \emptyset \vdash \mathsf{Var}\ 1}{A_2, A_1, \emptyset \vdash \mathsf{Var}\ 0 \parallel \mathsf{Var}\ 1} \qquad \frac{\bot, A_1, \emptyset \vdash \mathsf{Var}\ 0 \qquad A_2, \bot, \emptyset \vdash \mathsf{Var}\ 1}{A_2, A_1, \emptyset \vdash \mathsf{Var}\ 0 \parallel \mathsf{Var}\ 1}$$

We assume variable $\mathsf{Var}\ 0$ has type A_1 and variable $\mathsf{Var}\ 1$ has type A_2. The term expresses that there are two processes running in parallel, each with only a single variable. Additionally, the current environment consists of $A_2, A_1, \emptyset$. By the typing rules of the mailbox calculus (read bottom-up), given by de'Liguoro and Padovani [6], when typing parallel processes, the environment needs to be split into two. Assuming variable $\mathsf{Var}\ 0$ has indeed type A_1, the left subtree works without problems. Looking up the type at position $\mathsf{Var}\ 0$ in environment $A_1, \emptyset$ yields A_1. However, the right subtree poses a problem. There is no type at position $\mathsf{Var}\ 1$ in environment $A_2, \emptyset$, as it only contains a single element.

One way of solving this issue is the inclusion of entries in the environment representing the absence of a type at a certain position. The example on the right-hand side adds an additional $\bot$-entry, representing an empty position. With this approach, whenever an environment is split, the size of the two new environments is equal to the size of the original one. The *dblib* library we use for managing de Bruijn indices also includes an implementation of environments based on this exact approach. It provides the needed definitions and operations, such as lookup and insertion, together with proofs about properties on environments.

Fowler et al. [9] describe several relations and operations on environments, defined in Figure 10. Our definitions are essentially equal to the original ones,

Environment subtyping (◒) $\boxed{\Gamma_1 \leq \Gamma_2}$

$$\frac{}{\Gamma \leq \Gamma} \quad \frac{\Gamma_1 \leq \Gamma_2 \qquad \Gamma_2 \leq \Gamma_3}{\Gamma_1 \leq \Gamma_3} \quad \frac{\Gamma_1 \leq \Gamma_2}{\bot, \Gamma_1 \leq \bot, \Gamma_2} \quad \frac{\Gamma_1 \leq \Gamma_2 \qquad \mathsf{un}(A)}{A, \Gamma_1 \leq \bot, \Gamma_2} \quad \frac{\Gamma_1 \leq \Gamma_2 \qquad A_1 \leq A_2}{A_1, \Gamma_1 \leq A_2, \Gamma_2}$$

Environment combination (◒) $\boxed{\Gamma_1 \rhd \Gamma_2 = \Gamma_3}$

$$\frac{}{\emptyset \rhd \emptyset = \emptyset} \quad \frac{\Gamma_1 \rhd \Gamma_2 = \Gamma_3}{\bot, \Gamma_1 \rhd \bot, \Gamma_2 = \bot, \Gamma_3} \quad \frac{\Gamma_1 \rhd \Gamma_2 = \Gamma_3}{A, \Gamma_1 \rhd \bot, \Gamma_2 = A, \Gamma_3} \quad \frac{\Gamma_1 \rhd \Gamma_2 = \Gamma_3}{\bot, \Gamma_1 \rhd A, \Gamma_2 = A, \Gamma_3}$$

$$\frac{A_1 \rhd A_2 = A_3 \qquad \Gamma_1 \rhd \Gamma_2 = \Gamma_3}{A_1, \Gamma_1 \rhd A_2, \Gamma_2 = A_3, \Gamma_3}$$

Disjoint environment combination (◒) $\boxed{\Gamma_1 + \Gamma_2 = \Gamma_3}$

$$\frac{}{\emptyset + \emptyset = \emptyset} \quad \frac{\Gamma_1 + \Gamma_2 = \Gamma_3}{\bot, \Gamma_1 + \bot, \Gamma_2 = \bot, \Gamma_3} \quad \frac{\Gamma_1 + \Gamma_2 = \Gamma_3}{A, \Gamma_1 + \bot, \Gamma_2 = A, \Gamma_3} \quad \frac{\Gamma_1 + \Gamma_2 = \Gamma_3}{\bot, \Gamma_1 + A, \Gamma_2 = A, \Gamma_3}$$

$$\frac{\Gamma_1 + \Gamma_2 = \Gamma_3}{C, \Gamma_1 + C, \Gamma_2 = C, \Gamma_3}$$

Fig. 10. The definitions of relations on environments.

with modifications to incorporate $\bot$-entries and some corrections that are further discussed at the end of this section. In each relation we add a rule for the case where the first entry in both environments is $\bot$. Similar to partial operations on types, we turn partial operations on environments into relations.

Pat requires subtyping on *environments* rather than just types in order to rewrite types so that they can be used by the type combination operator. Sequential environment combination $\Gamma_1 \rhd \Gamma_2 = \Gamma$ makes use of type combination and is used when type checking let-expressions. This is the main rule used to ensure that subsequent uses of a mailbox variable "balance out". Additionally, we use *disjoint environment combination* to reason about expressions where subterms may not share mailbox-typed variables (e.g. a variable may not be used as the target of a message send and also be contained in its payload). Environments combined using $+$ are only allowed to share variables of base types and be disjoint otherwise.

As our definition of environments includes $\bot$ entries, it is possible for an environment to be empty without being $\emptyset$, necessitating a new notion of emptiness of environments $\mathsf{Empty}(\Gamma)$ (◒) that holds when Γ only contains $\bot$-entries.

Mechanization. Formalizing environments is, by far, the most complex part of this mechanization. The library *dblib* reduces some of the complexity around environments by proving operations and properties on them. However, reasoning about environment combinations and environment subtyping still is far from trivial. Independent of the representation of environments that were considered,

Pattern normal form (🏴‍☠️) $\boxed{E \models F}$

$$\frac{}{E \models \mathbb{0}} \qquad \frac{}{E \models \mathbb{1}} \qquad \frac{E \setminus \mathsf{m} \triangleq E' \qquad F \cong E'}{E \models \langle\!\langle \mathsf{m} \rangle\!\rangle \odot F} \qquad \frac{E \models F_1 \qquad E \models F_2}{E \models F_1 \oplus F_2}$$

Fig. 11. Pattern normal form.

environment splitting and subtyping require dozens of technical lemmas. While the proofs of these properties follow the same general approach, they are lengthy and tedious to perform. For example, for the relation $\Gamma_1 \rhd \Gamma_2 = \Gamma_3$ we have proved that there exists an environment that is the environment combination of both Γ_1 and Γ_2 with the entry at the same position replaced by $\bot$ in both environments (🏴‍☠️). Similar lemmas had to be proven for environment subtyping and disjoint combination. While mechanizing subtyping for environments according to the definition of Pat [9], we noticed that the relation was missing an inference rule, making environment subtyping not transitive. By explicitly including the transitivity rule we were able to prove desired properties of environment subtyping. This issue was subsequently fixed in a revised version [8].

3.4 Pattern Normal Form

When reasoning about the typing of guards, both the Mailbox Calculus and Pat require the type of a mailbox to be in *pattern normal form* (PNF), which is a particular structure of pattern that reflects the subpatterns expected by each guard clause. Specifically, a pattern E is in PNF if it is a sum of subpatterns of the form $\mathbb{0}$ (for a **fail** guard), $\mathbb{1}$ (for a **free** guard) and $\langle\!\langle \mathsf{m} \rangle\!\rangle \odot F$ (for a **receive** guard), where F is equivalent to the residual of E with respect to m. This allows a correct type to be given to a re-bound mailbox name after receiving.

Figure 11 shows the formal definitions for PNF; we write $\models E$ for $E \models E$. The following example derivation shows that pattern $(\langle\!\langle \mathsf{m} \rangle\!\rangle \odot \mathbb{1}) \oplus \mathbb{1}$ is in PNF:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\langle\!\langle \mathsf{m} \rangle\!\rangle \setminus \mathsf{m} \triangleq \mathbb{1}} \quad \dfrac{}{\mathbb{1} \setminus \mathsf{m} \triangleq \mathbb{0}}}{(\langle\!\langle \mathsf{m} \rangle\!\rangle \odot \mathbb{1}) \oplus \mathbb{1} \setminus \mathsf{m} \triangleq (\mathbb{1} \odot \mathbb{1}) \oplus (\langle\!\langle \mathsf{m} \rangle\!\rangle \odot \mathbb{0})} \quad \mathbb{1} \cong (\mathbb{1} \odot \mathbb{1}) \oplus (\langle\!\langle \mathsf{m} \rangle\!\rangle \odot \mathbb{0})}{(\langle\!\langle \mathsf{m} \rangle\!\rangle \odot \mathbb{1}) \oplus \mathbb{1} \models (\langle\!\langle \mathsf{m} \rangle\!\rangle \odot \mathbb{1})} \quad \dfrac{}{(\langle\!\langle \mathsf{m} \rangle\!\rangle \odot \mathbb{1}) \oplus \mathbb{1} \models \mathbb{1}}}{(\langle\!\langle \mathsf{m} \rangle\!\rangle \odot \mathbb{1}) \oplus \mathbb{1} \models (\langle\!\langle \mathsf{m} \rangle\!\rangle \odot \mathbb{1}) \oplus \mathbb{1}}$$

3.5 Typing Rules

We now have all the necessary definitions for specifying the typing rules of Pat. These rules are similar to the original ones by Fowler et al. [9], with a few modifications to fit our definitions of variables, environments and terms.

Figure 12 shows the typing rules for Pat, parameterised by a program $\mathcal{P}$; we omit $\mathcal{P}$ in the rules for readability. We highlight some of the typing rules. Whenever an empty environment is required, we use predicate $\mathsf{Empty}(\Gamma)$ instead of checking for the empty environment $\emptyset$. The insertion operation $[x \mapsto A]\Gamma$ ensures that Γ contains type A at position x.

Typing rules for programs and functions (▶) (▶) $\boxed{\vdash \mathcal{P}}$ $\boxed{\vdash_{\mathcal{P}} \textbf{def } \mathsf{f}(A) : B\ \{t\}}$

$$\frac{\mathcal{P} = (\mathcal{S}, \mathcal{D}, t_0) \qquad \forall \mathsf{f}.\ \vdash_{\mathcal{P}} \mathcal{D}(\mathsf{f}) \qquad \emptyset \vdash t_0 : \mathbf{1}}{\vdash \mathcal{P}} \qquad \frac{A, \emptyset \vdash t : B}{\vdash_{\mathcal{P}} \textbf{def } \mathsf{f}(A) : B\ \{t\}}$$

Typing rules for terms (▶) $\boxed{\Gamma \vdash_{\mathcal{P}} t : A}$

$$\textsc{Var}\ \frac{\mathsf{Empty}(\Gamma)}{[x \mapsto A]\Gamma \vdash \mathsf{Var}\ x : A} \qquad \textsc{Bool}\ \frac{\mathsf{Empty}(\Gamma)}{\Gamma \vdash b : \mathsf{Bool}} \qquad \textsc{Unit}\ \frac{\mathsf{Empty}(\Gamma)}{\Gamma \vdash () : \mathbf{1}} \qquad \textsc{App}\ \frac{\mathcal{D}_{\mathcal{P}}(\mathsf{f}) = \textbf{def } \mathsf{f}(A) : B\ \{t\} \quad \Gamma \vdash v : A}{\Gamma \vdash \mathsf{f}(v) : B}$$

$$\textsc{Let}\ \frac{\Gamma_1 \rhd \Gamma_2 = \Gamma \qquad \Gamma_1 \vdash t_1 : \lfloor A \rfloor \qquad [0 \mapsto \lfloor A \rfloor]\Gamma_2 \vdash t_2 : B}{\Gamma \vdash \textbf{let } t_1 \textbf{ in } t_2 : B} \qquad \textsc{New}\ \frac{\mathsf{Empty}(\Gamma)}{\Gamma \vdash \textbf{new} : ?\mathbb{1}^\bullet}$$

$$\textsc{Send}\ \frac{\mathcal{S}_{\mathcal{P}}(\mathsf{m}) = T \qquad \Gamma_1 \vdash v_1 : !\langle\!\langle \mathsf{m} \rangle\!\rangle^\circ \qquad \Gamma_2 \vdash v_2 : \lceil T \rceil \qquad \Gamma_1 + \Gamma_2 = \Gamma}{\Gamma \vdash v_1 ! \mathsf{m}[v_2] : \mathbf{1}}$$

$$\textsc{Guard}\ \frac{E \sqsubseteq F \qquad \models F \qquad \Gamma_1 \vdash v : ?F^\bullet \qquad \Gamma_2 \vdash \overrightarrow{g} : A :: F \qquad \Gamma_1 + \Gamma_2 = \Gamma}{\Gamma \vdash \textbf{guard } v : E\ \{\overrightarrow{g}\} : A} \qquad \textsc{Sub}\ \frac{\Gamma \leq \Gamma' \qquad A \leq B \qquad \Gamma' \vdash t : A}{\Gamma \vdash t : B}$$

$$\textsc{Spawn}\ \frac{\lceil \Gamma \rceil = \Gamma' \qquad \Gamma' \vdash t : \mathbf{1}}{\Gamma \vdash \textbf{spawn } t : \mathbf{1}}$$

Typing rules for guard sequences (▶) $\boxed{\Gamma \vdash_{\mathcal{P}} \overrightarrow{g} : A :: E}$

$$\frac{\Gamma \vdash g : A :: E}{\Gamma \vdash \{g\} : A :: E}\ \textsc{Single} \qquad \frac{\Gamma \vdash g : A :: E \qquad \Gamma \vdash \overrightarrow{g_s} : A :: E'}{\Gamma \vdash \{g, \overrightarrow{g_s}\} : A :: E \oplus E'}\ \textsc{Seq}$$

Typing rules for guards (▶) $\boxed{\Gamma \vdash_{\mathcal{P}} g : A :: E}$

$$\textsc{Fail}\ \frac{}{\Gamma \vdash \textbf{fail} : A :: \mathbb{0}} \qquad \textsc{Free}\ \frac{\Gamma \vdash t : A}{\Gamma \vdash \textbf{free} \mapsto t : A :: \mathbb{1}} \qquad \textsc{Receive}\ \frac{\mathcal{S}_{\mathcal{P}}(\mathsf{m}) = T \qquad \mathsf{base}(\mathsf{T}) \vee \mathsf{base}(\Gamma) \qquad [0 \mapsto \lceil T \rceil]([0 \mapsto ?E^\bullet]\Gamma) \vdash t : B}{\Gamma \vdash \textbf{receive } \mathsf{m} \mapsto t : B :: \langle\!\langle \mathsf{m} \rangle\!\rangle \odot E}$$

Fig. 12. Pat typing rules

The Send rule types a send expression $v_1 ! \mathsf{m}[v_2]$, where message m with payload v_2 is sent to mailbox v_1. The mailbox associated with v_1 has to be of type $!\langle\!\langle \mathsf{m} \rangle\!\rangle^\circ$, i.e. it must be capable of sending the message m. Payload v_2 is required to be of the type given by the message's signature with second-class usage. Both derivations for v_1 and v_2 must occur under separate environments Γ_1 and Γ_2, i.e. they are only share base types and their combination yields Γ.

Rule Receive types the receive guard **receive** $\mathsf{m} \mapsto t$, retrieving message m from the mailbox, with type B and pattern $\langle\!\langle \mathsf{m} \rangle\!\rangle \odot E$. Either Γ must only consist of base types or the payload's type T is a base type. This is a syntactic restriction, ensuring that no existing mailbox name is shadowed by a message's

payload. The continuation t must be typeable under Γ extended by $?E^\bullet$ and $\lceil T \rceil$. Type $?E^\bullet$ represents the mailbox after m is consumed. Because of the usage of de Bruijn indices, the order in which the types are inserted into the environment is important. Since inserting a type at position 0 corresponds to simply adding it to the front of the environment, the environment $[0 \mapsto \lceil T \rceil]([0 \mapsto ?E^\bullet]\Gamma)$ is equal to $\lceil T \rceil, ?E^\bullet, \Gamma$. Thus, a variable 0 would map to type $\lceil T \rceil$ and a variable 1 would map to type $?E^\bullet$.

The typing relation for a sequence of guards enables us to typecheck each guard individually, while also ensuring the correct pattern structure of the overall expression. In the original definition of typing for guard sequences, Fowler et al. [9] provide only a single rule of the following form:

$$\frac{(\Gamma \vdash g_i : A :: E_i)_i}{\Gamma \vdash \overrightarrow{g} : A :: E_1 \oplus \ldots \oplus E_n}$$

By splitting this single rule into two distinct ones (Single and Seq), we provide a recursive definition for typing of sequences. This avoids using lists of well-typed relations as a premise in the rule, and in turn, Rocq is able to automatically come up with appropriate inductive schemes.

An example of a type derivation in Pat can be found in Appendix A.

Mechanization. The mechanization of Pat's typing rules closely follows the pen-and-paper definitions. However, proving properties about well-typed terms requires technical lemmas, such as inversion lemmas or canonical forms of terms. For example, we had to prove that every well-typed Boolean value under environment Γ implies that Γ is a subenvironment of an empty environment (🐦). These types of proofs would usually be omitted in classical pen-and-paper proofs, but in our case, even these kinds of proofs required technical lemmas such as properties about environment subtyping.

3.6 Substitution

The call-by-value operational semantics of Pat ensures that substitution only needs to replace a variable with a value. However, since Pat enforces a syntactic separation of values and computations (following *fine-grain call-by-value* [16]), variables are classed as values, which makes the proof of substitution nontrivial.

The Rocq library *dblib* [18], which we use for de Bruijn indices, provides a definition of substitution. To use it, a user defines substitution as a *traverse* function (🐦) over terms. It applies a function f at every variable and provides f with the number of binders that have been visited. To use the substitution operation, several lemmas about the traverse function need to be proven.

Lemma 4 (Substitution (🐦)). *If* $[x \mapsto A]\Gamma_1 \vdash t : B$ *and* $\Gamma_2 \vdash v : A'$*, with* $A \leq A'$ *and* $\Gamma_1 + \Gamma_2 = \Gamma$*, then* $\Gamma \vdash \{v/x\}t : B$.

Mechanization. Originally, the proof of this lemma was assumed to be standard [6,9]. The mechanization revealed the opposite to be the case, and the proof is by far the most complex and largest in the mechanization. One of the most complex parts of the proof is the Let case, which requires extensive reasoning about variable lifting and environment combination. Additionally, due to quasi-linear typing, environment combinations and environment subtyping, one has to consider more cases than initially expected. Although *dblib* alleviates some of the proof burden, manual work is still required. Since variables are considered values in Pat, reasoning about substituting one variable for another is required. While *dblib* provides tactics for automatically proving some of these properties, it often would not be able to do so. We are unsure as to why this is the case.

4 Related Work

Since little research exists on mailbox types, we discuss related work with a focus on mechanizations of the π-calculus and session types [12]. These are tailored to languages that make use of *communication channels* rather than mailboxes.

Ambal et al. [1] present a Rocq formalization of a higher-order π-calculus, where messages are able to carry processes. Their work focuses on the representation of binders in name restrictions of processes. The authors provide four versions of their mechanization, each with a different representation of variables. They conclude that their de Bruijn formalization is the most concise, but requires complex manipulation of de Bruijn indices and intricate proofs about renaming lemmas. This fact can also be observed in our work.

Goto et al. [11] provide a mechanization of polymorphic session types for the π-calculus in Rocq. Their mechanization uses the locally-nameless [4] approach for variables. Similar to our work, the authors first provide their definitions for processes and types independently. They are able to prove that their type system guarantees preservation and safety properties.

Thiemann [21] presents a mechanization of session types in Agda. In contrast to other mechanizations, his approach makes use of *intrinsically* typed terms, meaning that only well-typed terms can be constructed. As is common with intrinsically typed mechanizations, variables are represented by de Bruijn indices [23]. The preservation and progress properties are proven by construction. Intrinsic typing can lead to reduced proof sizes and fewer lemmas [23]. Whether such a representation would also work for Pat is worth exploring.

Tirore et al. [22] provide a mechanization of multiparty asynchronous session types in Rocq. They rely on the original definitions by Honda et al. [13]. During their mechanization they discovered flaws with the original definitions, such that type preservation does not hold in that system. They propose a new type system for which preservation and other properties hold, formally proving this claim. To manage the complexity of variables, they use the *Autosubst2* library [19], which generates Rocq code based on the de Bruijn representation. In contrast to our formalization, the authors choose to represent environments as a list of

pairs, with name and type. They claim that this representation, in particular, simplified context splitting for parallel composition.

Zalakain and Dardha [24] present a mechanization of a type system for the π-calculus in Agda. Their approach facilitates *leftover typing*, where the typing judgment includes a second leftover environment. The leftover environment contains types not used in the typing of the term, leaving them available for parallel processes. This technique eliminates the need for environment splits but would likely be difficult to use due to the need for type combination.

5 Conclusion and Future Work

In this paper, we have described the first mechanization of mailbox types and their semantics, as well as the syntax and semantics of the Pat programming language, in the Rocq Prover. We show that mechanizing Pat is nontrivial: in particular, environment combinations and subtyping pose a big challenge due to the numerous technical lemmas needed. Our work successfully provides a significant number of machine-checked proofs for various properties about mailbox types and Pat's static semantics. During the mechanization process, we also identified and fixed multiple oversights in the original definitions and proofs.

The next step for future work is the mechanization of the operational semantics, which would allow us to formulate machine-checked proofs of properties such as preservation and progress. Initial work has shown that a primary difficulty in formalizing Pat's operational semantics is the constant manipulation of de Bruijn indices. While tedious, previous work [24] was able to successfully formalize a similar semantics with this representation of variables. Alternatively, we could make heavier use of Rocq's dependent types. In some mechanizations of the π-calculus [1], the type system keeps track of available channel endpoints for each process, which could potentially reduce the proofs regarding de Bruijn indices.

Acknowledgements. We thank the anonymous reviewers for their helpful comments. Fowler was supported by EPSRC grant EP/T014628/1 (STARDUST).

References

1. Ambal, G., Lenglet, S., Schmitt, A.: HOπ in Coq. Journal of Automated Reasoning **65**(1), 75–124 (Jan 2021). https://doi.org/10.1007/s10817-020-09553-0
2. Armstrong, J.: Making reliable distributed systems in the presence of software errors. Ph.D. thesis, Royal Institute of Technology, Stockholm, Sweden (2003), https://nbn-resolving.org/urn:nbn:se:kth:diva-3658
3. Brzozowski, J.A.: Derivatives of Regular Expressions. J. ACM **11**(4), 481–494 (Oct 1964). https://doi.org/10.1145/321239.321249
4. Charguéraud, A.: The Locally Nameless Representation. Journal of Automated Reasoning **49**(3), 363–408 (Oct 2012). https://doi.org/10.1007/s10817-011-9225-2
5. de Bruijn, N.G.: Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. Indagationes Mathematicae (Proceedings) **75**(5), 381–392 (Jan 1972). https://doi.org/10.1016/1385-7258(72)90034-0
6. de'Liguoro, U., Padovani, L.: Mailbox Types for Unordered Interactions. LIPIcs, Volume 109, ECOOP 2018 **109**, 15:1–15:28 (2018). https://doi.org/10.4230/LIPICS.ECOOP.2018.15
7. Ennals, R., Sharp, R., Mycroft, A.: Linear Types for Packet Processing. In: Schmidt, D. (ed.) Programming Languages and Systems. pp. 204–218. Springer, Berlin, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24725-8_15
8. Fowler, S., Attard, D.P., Marshall, D., Gay, S.J., Trinder, P.: Special Delivery: Programming with Mailbox Types (Extended Version) (Sep 2025). https://doi.org/10.48550/arXiv.2306.12935
9. Fowler, S., Attard, D.P., Sowul, F., Gay, S.J., Trinder, P.: Special Delivery: Programming with Mailbox Types. Proceedings of the ACM on Programming Languages **7**(ICFP), 78–107 (Aug 2023). https://doi.org/10.1145/3607832
10. Fowler, S., Attard, D.P., Sowul, F., Gay, S.J., Trinder, P.: Special Delivery: Programming with Mailbox Types (Extended Version) (Jun 2023). https://doi.org/10.48550/arXiv.2306.12935
11. Goto, M., Jagadeesan, R., Jeffrey, A., Pitcher, C., Riely, J.: An extensible approach to session polymorphism. Mathematical Structures in Computer Science **26**(3), 465–509 (Mar 2016). https://doi.org/10.1017/S0960129514000231
12. Honda, K.: Types for dyadic interaction. In: Best, E. (ed.) CONCUR'93. pp. 509–523. Springer, Berlin, Heidelberg (1993). https://doi.org/10.1007/3-540-57208-2_35
13. Honda, K., Yoshida, N., Carbone, M.: Multiparty Asynchronous Session Types. J. ACM **63**(1), 9:1–9:67 (Mar 2016). https://doi.org/10.1145/2827695
14. Kobayashi, N.: Quasi-linear types. In: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 29–42. POPL '99, Association for Computing Machinery, New York, NY, USA (Jan 1999). https://doi.org/10.1145/292540.292546
15. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM **52**(7), 107–115 (Jul 2009). https://doi.org/10.1145/1538788.1538814
16. Levy, P., Power, J., Thielecke, H.: Modelling environments in call-by-value programming languages. Information and Computation **185**(2), 182–210 (Sep 2003). https://doi.org/10.1016/S0890-5401(03)00088-9
17. Pierce, B.C.: Types and Programming Languages. MIT Press, Cambridge, Mass (2002)

18. Pottier, F., Orr, K.: Dblib. https://github.com/rocq-community/dblib (2021), commit: 25469872c0ba99b046f7e5b86
08205eeea5ac077
19. Stark, K.: Mechanising Syntax with Binders in Coq. Ph.D. thesis, Universität des Saarlandes, Saarbrücken, Germany (2019)
20. Team, T.R.D.: The Rocq Prover. Zenodo (Apr 2025). https://doi.org/10.5281/zenodo.15149629
21. Thiemann, P.: Intrinsically-Typed Mechanized Semantics for Session Types. In: Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming. pp. 1–15. PPDP '19, Association for Computing Machinery, New York, NY, USA (Oct 2019). https://doi.org/10.1145/3354166.3354184
22. Tirore, D., Bengtson, J., Carbone, M.: Multiparty Asynchronous Session Types: A Mechanised Proof of Subject Reduction. In: Aldrich, J., Silva, A. (eds.) 39th European Conference on Object-Oriented Programming (ECOOP 2025). Leibniz International Proceedings in Informatics (LIPIcs), vol. 333, pp. 31:1–31:30. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2025). https://doi.org/10.4230/LIPIcs.ECOOP.2025.31
23. Wadler, P., Kokke, W., Siek, J.G.: Programming Language Foundations in Agda (Aug 2022), https://plfa.inf.ed.ac.uk/22.08/
24. Zalakain, U., Dardha, O.: π with Leftovers: A Mechanisation in Agda. In: Formal Techniques for Distributed Objects, Components, and Systems: 41st IFIP WG 6.1 International Conference, FORTE 2021, Held as Part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14–18, 2021, Proceedings. pp. 157–174. Springer-Verlag, Berlin, Heidelberg (Jun 2021). https://doi.org/10.1007/978-3-030-78089-0_9

A Type Derivation Example

We show an example of how type checking works in Pat. To this end, we model a small system with two processes where one process sends a Ping message to the other process. As a response, a Pong message is sent back. The messages have signature $\mathcal{S} = \{\mathsf{Ping} \mapsto !\langle\!\langle \mathsf{Pong} \rangle\!\rangle, \mathsf{Pong} \mapsto \mathbf{1}\}$, i.e. the payload of Ping contains a reference to a mailbox expecting a Pong message, and the payload of Pong is of the unit type.

For better readability we use explicit names as variables in this example and further syntactic sugar. We write $t_1 \ ; \ t_2$ for **let** $x = t_1$ **in** t_2, where x has type $\mathbf{1}$ and does not occur in t_2. For directly freeing a mailbox V, we write **free** V as a shorthand for **guard** $V : \mathbb{1}$ {**free** $\mapsto$ ()}.

```
1  def client() : 1 {
2    let server = new in
3    spawn serverListen(server);
4    let self = new in
5    server!Ping[self];
6    guard self : ⟪Pong⟫ {
7      receive Pong[reply] from self ↦
8        free self; ()
9    }
10 }
```

```
11 def serverListen(self : ?⟪Ping⟫ ⊕ 𝟙•) : 1 {
12   guard self : ⟪Ping⟫ ⊕ 𝟙 {
13       receive Ping[reply] from self ↦
14         reply!Pong[self];
15         free self; ()
16       free ↦ ()
17   }
18 }
```

The program consists of two functions: client and serverListen. Function client is the initial starting point, where a new mailbox name is bound to variable *server*. Then, a new process with serverListen is created, providing *server* as argument. The function continues by again creating a new mailbox name, binding it to *self*, and sending a Ping message to *server* with *self* as payload. Then, in the guard-expression, a receive guard is triggered when *self* contains a Pong message. The mailbox is freed and the unit value is returned.

Function serverListen requires an argument of type $?\langle\!\langle \mathsf{Ping} \rangle\!\rangle \oplus \mathbb{1}^\bullet$, indicating a mailbox containing a Ping message or being empty. The function's body contains only a single guard-expression consisting of two guards. When a Ping message with some payload *reply* is received, a Pong message is sent to *reply*. After that, *self* is freed and the unit value is returned. The other guard is triggered when *self* does not contain any messages; in this case, the mailbox is freed.

Using the syntax we defined with de Bruijn indices and without syntactic sugar, both functions look as follows:

```
1  def client() : 1 {
2    let new in
3    let spawn serverListen(Var 0) in
4    let new in
5    let (Var 2)!Ping[Var 0] in
6    guard Var 1 : ⟪Pong⟫ {
7      receive Pong ↦
8        guard Var 1 : 𝟙 {free ↦ ()}
9    }
10 }
```

```
11 def serverListen(self : ?Ping ⊕ 𝟙•) : 1 {
12   guard Var 0 : ⟪Ping⟫ ⊕ 𝟙 {
13       receive Ping ↦
14         let (Var 0)!Pong[()] in
15         guard Var 2 : 𝟙 {free ↦ ()}
16       free ↦ ()
17   }
18 }
```

Next, we present a type derivation of the client definition. Due to the amount of rules needed for this derivation, we focus on the subtrees with the essential

reasoning steps and leave those out that mainly contain formal steps. To make environments more readable, the abbreviation $\bot^n$ is used for an environment consisting of n $\bot$-entries. Additionally, $\emptyset$ symbols are omitted from environments.

Since, in our mechanization of Pat, every function requires exactly one argument, we need to give client an argument. Thus, we define client to take an argument of type $\mathbf{1}$. The derivation therefore starts with an environment consisting of a single $\mathbf{1}$ entry.

$$\dfrac{\dfrac{}{\bot \vdash \textbf{new} : ?\mathbb{1}^\bullet}\ \textsc{New}}{\mathbf{1} \vdash \textbf{new} : ?\mathbb{1}^\bullet}\ \textsc{Sub} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{?\langle\!\langle \mathsf{Ping} \rangle\!\rangle \oplus \mathbb{1}^\bullet, \bot \vdash \mathsf{Var}\ 0 : ?\langle\!\langle \mathsf{Ping} \rangle\!\rangle \oplus \mathbb{1}^\bullet}\ \textsc{Var}}{?(\langle\!\langle \mathsf{Ping} \rangle\!\rangle \oplus \mathbb{1}) \odot \mathbb{1}^\bullet, \bot \vdash \mathsf{Var}\ 0 : ?\langle\!\langle \mathsf{Ping} \rangle\!\rangle \oplus \mathbb{1}^\bullet}\ \textsc{Sub}}{?(\langle\!\langle \mathsf{Ping} \rangle\!\rangle \oplus \mathbb{1}) \odot \mathbb{1}^\bullet, \bot \vdash \mathsf{serverListen}(\mathsf{Var}\ 0) : \mathbf{1}}\ \textsc{App}}{?(\langle\!\langle \mathsf{Ping} \rangle\!\rangle \oplus \mathbb{1}) \odot \mathbb{1}^\circ, \bot \vdash \textbf{spawn}\ \mathsf{serverListen}(\mathsf{Var}\ 0) : \mathbf{1}}\ \textsc{Spawn} \qquad \mathbf{D_1}}{?\mathbb{1}^\bullet, \bot \vdash \begin{array}{l}\textbf{let spawn}\ \mathsf{serverListen}(\mathsf{Var}\ 0)\ \textbf{in}\\ \textbf{let new in}\ \ldots\end{array} : \mathbf{1}}\ \textsc{Let}$$

$$\mathbf{1} \vdash \begin{array}{l}\textbf{let new in}\\ \textbf{let spawn}\ \mathsf{serverListen}(\mathsf{Var}\ 0)\ \textbf{in}\\ \textbf{let new in}\\ \textbf{let}\ (\mathsf{Var}\ 2)!\mathsf{Ping}[\mathsf{Var}\ 0]\ \textbf{in}\\ \textbf{guard}\ \mathsf{Var}\ 1 : \langle\!\langle \mathsf{Pong} \rangle\!\rangle \{\\ \quad \textbf{receive}\ \mathsf{Pong} \mapsto\\ \quad\quad \textbf{guard}\ \mathsf{Var}\ 1 : \mathbb{1}\ \{\textbf{free} \mapsto ()\}\\ \}\end{array} : \mathbf{1}$$

For the first let-expression, the environment is split as $\mathbf{1} \rhd \bot = \mathbf{1}$, making $\mathbf{1}$ only available in the left subtree. Since this $\mathbf{1}$ entry is not needed, it is removed using environment subtyping. Then, the left subtree can be typed with $?\mathbb{1}^\bullet$ by the New rule. In the right subtree, $?\mathbb{1}^\bullet$ is added to the environment as the result type of the left term. For the right subtree, we again use the Let rule, where the right derivation is described by tree $\mathbf{D_1}$. Now the environment is split using the combination $?(\langle\!\langle \mathsf{Ping} \rangle\!\rangle \oplus \mathbb{1}) \odot \mathbb{1}^\circ, \bot \rhd\ !\langle\!\langle \mathsf{Ping} \rangle\!\rangle \oplus \mathbb{1}^\bullet, \bot = ?\mathbb{1}^\bullet, \bot$. This specific type combination is used since serverListen requires an argument of type $?\langle\!\langle \mathsf{Ping} \rangle\!\rangle \oplus \mathbb{1}^\bullet$, indicating a mailbox containing a Ping message or being empty. The derivation for term **spawn** serverListen(Var 0) is straightforward by applying the appropriate rules. One additional subtyping step is made before the Var rule to transform type $?(\langle\!\langle \mathsf{Ping} \rangle\!\rangle \oplus \mathbb{1}) \odot \mathbb{1}^\bullet$ into $?\langle\!\langle \mathsf{Ping} \rangle\!\rangle \oplus \mathbb{1}^\bullet$. Since $\mathbb{1}$ is the neutral element for pattern composition, it can be dropped while keeping the pattern's semantics. The next tree describes the derivation at $\mathbf{D}_1$:

$$\dfrac{\dfrac{}{\bot^3 \vdash \textbf{new} : ?\mathbb{1}^\bullet}\ \textsc{New} \qquad \dfrac{\dfrac{\dfrac{}{\bot^2, !\langle\!\langle \mathsf{Ping} \rangle\!\rangle^\circ, \bot \vdash \mathsf{Var}\ 2 : !\langle\!\langle \mathsf{Ping} \rangle\!\rangle^\circ}\ \textsc{Var}}{\bot^2, !\langle\!\langle \mathsf{Ping} \rangle\!\rangle \oplus \mathbb{1}^\bullet, \bot \vdash \mathsf{Var}\ 2 : !\langle\!\langle \mathsf{Ping} \rangle\!\rangle^\circ}\ \textsc{Sub} \qquad \dfrac{\dfrac{}{!\langle\!\langle \mathsf{Pong} \rangle\!\rangle^\circ, \bot^3 \vdash \mathsf{Var}\ 0 : !\langle\!\langle \mathsf{Pong} \rangle\!\rangle^\circ}\ \textsc{Var}}{!\langle\!\langle \mathsf{Pong} \rangle\!\rangle^\circ, \mathbf{1}, \bot^2 \vdash \mathsf{Var}\ 0 : !\langle\!\langle \mathsf{Pong} \rangle\!\rangle^\circ}\ \textsc{Sub}}{!\langle\!\langle \mathsf{Pong} \rangle\!\rangle^\circ, \mathbf{1}, !\langle\!\langle \mathsf{Ping} \rangle\!\rangle \oplus \mathbb{1}^\bullet, \bot \vdash (\mathsf{Var}\ 2)!\mathsf{Ping}[\mathsf{Var}\ 0] : \mathbf{1}}\ \textsc{Send} \qquad \mathbf{D_2}}{?\mathbb{1}^\bullet, \mathbf{1}, !\langle\!\langle \mathsf{Ping} \rangle\!\rangle \oplus \mathbb{1}^\bullet, \bot \vdash \begin{array}{l}\textbf{let}\ (\mathsf{Var}\ 2)!\mathsf{Ping}[\mathsf{Var}\ 0]\ \textbf{in}\\ \textbf{guard}\ \mathsf{Var}\ 1 : \langle\!\langle \mathsf{Pong} \rangle\!\rangle \{\\ \quad \textbf{receive}\ \mathsf{Pong} \mapsto\\ \quad\quad \textbf{guard}\ \mathsf{Var}\ 1 : \mathbb{1}\ \{\textbf{free} \mapsto ()\}\\ \}\end{array} : \mathbf{1}}\ \textsc{Let}}{\mathbf{1}, !\langle\!\langle \mathsf{Ping} \rangle\!\rangle \oplus \mathbb{1}^\bullet, \bot \vdash \begin{array}{l}\textbf{let new in}\\ \textbf{let}\ (\mathsf{Var}\ 2)!\mathsf{Ping}[\mathsf{Var}\ 0]\ \textbf{in}\\ \textbf{guard}\ \mathsf{Var}\ 1 : \langle\!\langle \mathsf{Pong} \rangle\!\rangle \{\\ \quad \textbf{receive}\ \mathsf{Pong} \mapsto\\ \quad\quad \textbf{guard}\ \mathsf{Var}\ 1 : \mathbb{1}\ \{\textbf{free} \mapsto ()\}\\ \}\end{array} : \mathbf{1}}\ \textsc{Let}$$

The derivation starts by applying the LET rule, where **new** from the left subterm of the let-expression is trivially typed using NEW and the empty environment. For the right subterm, another LET rule is used, creating one subtree for typing the send-term and one subtree for typing the guard. The guard is typed using another derivation $\mathbf{D_2}$. To type the sending of message Ping successfully, we require two mailbox types for variables Var 2 and Var 0. Since Var 2 is the receiving mailbox, it must be able to receive a Ping message, i.e. have type $!\langle\!\langle \mathsf{Ping} \rangle\!\rangle^\circ$. Because the current environment contains the similar type $!\langle\!\langle \mathsf{Ping} \rangle\!\rangle \oplus \mathbb{1}^\bullet$, we are able to type Var 2 with the help of subtyping, since $!\langle\!\langle \mathsf{Ping} \rangle\!\rangle \oplus \mathbb{1}^\bullet \leq !\langle\!\langle \mathsf{Ping} \rangle\!\rangle^\circ$. For Var 0, the signature of Ping tells us that Var 0 has to have type $!\langle\!\langle \mathsf{Pong} \rangle\!\rangle^\circ$. Thus, the environment combination from the previous LET rule is $!\langle\!\langle \mathsf{Pong} \rangle\!\rangle^\circ, \mathbf{1}, !\langle\!\langle \mathsf{Ping} \rangle\!\rangle \oplus \mathbb{1}^\bullet, \bot \triangleright ?\langle\!\langle \mathsf{Pong} \rangle\!\rangle \odot \mathbb{1}^\bullet, \bot^3 = ?\mathbb{1}^\bullet, \mathbf{1}, !\langle\!\langle \mathsf{Ping} \rangle\!\rangle \oplus \mathbb{1}^\bullet, \bot$. This introduces $\langle\!\langle \mathsf{Pong} \rangle\!\rangle$ into the environment, letting us correctly type Var 0 with an intermediate SUB step to get rid of the unnecessary $\mathbf{1}$. Tree $\mathbf{D_2}$ describes the typing derivation of the guard:

$$
\dfrac{
\dfrac{}{\bot, ?\langle\!\langle \mathsf{Pong} \rangle\!\rangle \odot \mathbb{1}^\bullet, \bot^3 \vdash \mathsf{Var}\ 1 : ?\langle\!\langle \mathsf{Pong} \rangle\!\rangle \odot \mathbb{1}^\bullet}\ \textsc{Var}
\qquad
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{}{\bot, ?\mathbb{1}^\bullet, \bot^5 \vdash \mathsf{Var}\ 1 : ?\mathbb{1}^\bullet}\ \textsc{Var}
}{\mathbf{1}, ?\mathbb{1}^\bullet, \bot^5 \vdash \mathsf{Var}\ 1 : ?\mathbb{1}^\bullet}\ \textsc{Sub}
}{\mathbf{1}, ?\mathbb{1}^\bullet, \bot^5 \vdash \mathbf{guard}\ \mathsf{Var}\ 1 : \mathbb{1}\ \{\ \mathbf{free} \mapsto ()\ \}\ : \mathbf{1} :: ?\langle\!\langle \mathsf{Pong} \rangle\!\rangle \odot \mathbb{1}^\bullet}\ \textsc{Guard}
}{\bot^5 \vdash \mathbf{receive}\ \mathsf{Pong} \mapsto \mathbf{guard}\ \mathsf{Var}\ 1 : \mathbb{1}\ \{\ \mathbf{free} \mapsto ()\ \}\ : \mathbf{1} :: ?\langle\!\langle \mathsf{Pong} \rangle\!\rangle \odot \mathbb{1}^\bullet}\ \textsc{Receive}
}{\bot^5 \vdash \mathbf{receive}\ \mathsf{Pong} \mapsto \mathbf{guard}\ \mathsf{Var}\ 1 : \mathbb{1}\ \{\ \mathbf{free} \mapsto ()\ \}\ : \mathbf{1} :: ?\langle\!\langle \mathsf{Pong} \rangle\!\rangle \odot \mathbb{1}^\bullet}\ \textsc{Single}
}{\bot, ?\langle\!\langle \mathsf{Pong} \rangle\!\rangle \odot \mathbb{1}^\bullet, \bot^3 \vdash \mathbf{guard}\ \mathsf{Var}\ 1 : \langle\!\langle \mathsf{Pong} \rangle\!\rangle \{\ \mathbf{receive}\ \mathsf{Pong} \mapsto \mathbf{guard}\ \mathsf{Var}\ 1 : \mathbb{1}\ \{\mathbf{free} \mapsto ()\}\ \}\ : \mathbf{1}}\ \textsc{Guard}
}{\mathbf{1}, ?\langle\!\langle \mathsf{Pong} \rangle\!\rangle \odot \mathbb{1}^\bullet, \bot^3 \vdash \mathbf{guard}\ \mathsf{Var}\ 1 : \langle\!\langle \mathsf{Pong} \rangle\!\rangle \{\ \mathbf{receive}\ \mathsf{Pong} \mapsto \mathbf{guard}\ \mathsf{Var}\ 1 : \mathbb{1}\ \{\mathbf{free} \mapsto ()\}\ \}\ : \mathbf{1}}\ \textsc{Sub}
$$

Before typing the guard, subtyping is used to remove the newly introduced, but unused $\mathbf{1}$ from the environment. Then, the GUARD rule can be applied, where the guarded mailbox name Var 1 can be typed using the current environment. GUARD requires some pattern F in pattern normal form and a superpattern of $\langle\!\langle \mathsf{Pong} \rangle\!\rangle$. Thus, we choose $F = \langle\!\langle \mathsf{Pong} \rangle\!\rangle \odot \mathbb{1}$, since $\langle\!\langle \mathsf{Pong} \rangle\!\rangle \sqsubseteq \langle\!\langle \mathsf{Pong} \rangle\!\rangle \odot \mathbb{1}$, which is in PNF:

$$\dfrac{\dfrac{\dfrac{}{\langle\!\langle \mathsf{Pong} \rangle\!\rangle \setminus \mathsf{Pong} \triangleq \mathbb{1}} \quad \dfrac{}{\mathbb{1} \setminus \mathsf{Pong} \triangleq \mathbb{0}}}{\langle\!\langle \mathsf{Pong} \rangle\!\rangle \odot \mathbb{1} \setminus \mathsf{Pong} \triangleq \mathbb{1}} \quad \dfrac{}{\mathbb{1} \odot \mathbb{1} \oplus \langle\!\langle \mathsf{Pong} \rangle\!\rangle \odot \mathbb{0} \cong \mathbb{1} \oplus \mathbb{0} \cong \mathbb{1}}}{\langle\!\langle \mathsf{Pong} \rangle\!\rangle \odot \mathbb{1} \models \langle\!\langle \mathsf{Pong} \rangle\!\rangle \odot \mathbb{1}}$$

For typing the actual guard, we start with Single , as the guard sequence only consists of a single **receive** guard. The Receive rule adds two new types to the environment when typing the continuation term after receive message Pong: type $?\mathbb{1}^\bullet$ and $\mathbf{1}$. The unit type $\mathbf{1}$ comes from the payload type associated with Pong, while $?\mathbb{1}^\bullet$ is derived from the pattern $?\langle\!\langle \mathsf{Pong} \rangle\!\rangle \odot \mathbb{1}^\bullet$ and describes the mailbox after receiving a Pong message. The last guard freeing the mailbox is typed trivially using the appropriate rules, with an additional subtyping step to remove the previously added $\mathbf{1}$ from the environment.

This example illustrates how challenging it can be to find the correct typing derivation. The main difficulty lies in combining environments. While it is relatively straightforward to construct disjoint environment combinations, the general environment combinations required by the Let rule are much harder to determine. One must choose environments that allow each subtree to be typed successfully while ensuring that their combination reconstructs the original environment. Another complication is that the typing rules are not deterministic. In this example, the Sub rule could be applied at several different points in the derivation tree. For example, in $\mathbf{D_2}$ the first rule used is Sub to get rid of $\mathbf{1}$ in the environment. An alternative is to first use Guard and after that Sub in the left subtree when typing Var 1.