Special Delivery

Programming with Mailbox Types*

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

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

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

1 INTRODUCTION

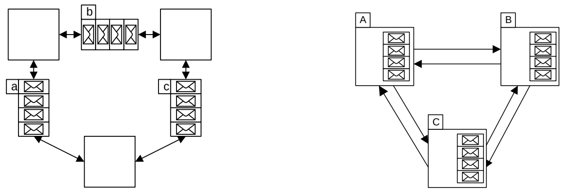
Software is increasingly concurrent and distributed, but coordinating concurrent computations introduces a host of additional correctness issues like communication mismatches and deadlocks. Communication-centric languages such as Go, Erlang, and Elixir make it possible to avoid many of the issues stemming from shared memory concurrency by structuring applications as independent, lightweight processes that communicate through explicit message passing. There are two main classes of communication-centric language. In *channel-based* languages like Go or Rust, processes communicate over channels, where a `send` in one process is paired with a `receive` in the recipient process. In *actor* languages like Erlang or Elixir, a message is sent to the *mailbox* of the recipient process, which is an incoming message queue. The communication patterns are more flexible as the recipient process can choose which message from the mailbox to handle next.

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

Behavioural type systems [33] encode correct communication behaviour to support *correct-by-construction* concurrency. Behavioural type systems, in particular *session types* [27, 28, 54], have been extensively applied to specify communication protocols in channel-based languages [3]. 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 [25, 40, 52, 53] or relies on dynamic typing [42]. We discuss these systems further in (§7).

Our approach is based on *mailbox types*, a behavioural type system for mailboxes first introduced in the context of a process calculus [12]. We present the first programming language design

*Draft (2nd February 2023)



(a) Channels   (b) Actors

Fig. 1. Channel- and actor-based languages [20]

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 use of quasi-linear typing [15, 37] to tame some of the complexity introduced by aliasing, and our algorithmic type system is necessarily co-contextual [16, 38], achieved through a novel use of backwards bidirectional typing [60].

1.1 Channel-based vs Actor Communication

Channel-based languages comprise anonymous processes that communicate over named channels, whereas actor-based languages comprise named processes each equipped with a mailbox. Figure 1 contrasts the approaches, and is taken from a detailed comparison [20].

Actor languages have proven to be effective for implementing reliable and scalable distributed systems [56]. A key benefit of actor languages is that communication is asynchronous and unidirectional: many actors may *send* messages to an actor A, whereas only A may *receive* from its mailbox. Mailboxes provide *data locality* as each message is stored with the process that will handle it. Since channel-based languages allow channel names to be communicated, they must either sacrifice locality and reduce performance, or rely on complex distributed algorithms [7, 32].

Although it is straightforward to add a type system to channel-based languages, adding a type system to actor languages is less straightforward, as process names (process IDs or PIDs) must be parameterised by a type that supports all messages that can be received. The type is therefore less precise, requiring subtyping [26] or synchronisation [10, 55] to avoid a total loss of modularity [20].

The situation becomes even more pronounced when considering behavioural type systems: communication errors might be prevented by giving one end of a channel the session type $\mathsf{!Int.!Int.?Bool.End}$ (send two integers, and receive a Boolean), and the other end the *dual* type $\mathsf{?Int.?Int.!Bool.End}$. Behavioural type systems for actor languages are much less straightforward due to asymmetric communication. In practice, designers of session type systems for actor languages either emulate session-typed channels [40], or use *multiparty session types* to govern the communication actions performed by a process, requiring a fixed communication topology [42].

1.2 Mailbox types

de'Liguoro and Padovani [12] observe that session types require a *strict ordering* of messages, whereas most actor systems use *selective receive* to process messages out-of-order. Concentrating on *unordered* interactions enables behavioural typing for mailboxes with many writers.

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

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

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

Several communication errors can arise in this example:

- **Protocol violation.** Sending two `put` messages to the future will result in a *runtime* error.
- **Unexpected message.** Sending a message other than `get` or `put` to the future will silently succeed, but the message will never be retrieved from the mailbox, 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:

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

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 [12] introduce a small extension of the asynchronous π-calculus [2], which they call the *mailbox calculus*, and endow it with mailbox types. They express the Future example in the mailbox calculus as follows, where the mailbox is denoted *self*.

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

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

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

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

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

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

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

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

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

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

(1) We introduce a declarative type system for Pat (§3), a first-order programming language with support for mailbox types, making essential and novel use of quasi-linear types. We show type preservation, mailbox conformance, and a progress result.

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

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

(a) Using old name

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

(b) Renaming

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

(c) Evaluation contexts

Fig. 3. Use-after-free via aliasing

(2) We introduce a co-contextual algorithmic type system for Pat (§4), making use of backwards bidirectional typing. We prove that the algorithmic type system is sound and complete with respect to the declarative type system.
(3) We extend Pat with sum and product types; interfaces; and higher-order functions (§5).
(4) We detail our implementation (§6), and demonstrate the expressivity of Pat by encoding all of the examples from de'Liguoro and Padovani [12], and 10 of the 11 Savina benchmarks [34] used by Neykova and Yoshida [42] in their evaluation of multiparty session types for actor languages (§6.2).

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

Session typing was originally studied in the context of process calculi (e.g., [28, 57]), but later work [19, 21, 58] introduced session types for languages based on the linear λ-calculus. The more relaxed view of linearity in the mailbox calculus makes language integration far more challenging. A mailbox name may be used several times to *send* messages, but only once to *receive* a message. The intuition is that while sends simply add messages to a mailbox, it is a receive that determines the future behaviour of the actor. To illustrate, Figure 2 shows a fragment of the future example from §1 with two sends to the *future* mailbox (shaded red), and a single receive (shaded blue).

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

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

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 applications. Moreover functional languages provide nested evaluation contexts and sequential evaluation.

2.1 Controlling Mailbox Aliasing

Ensuring appropriate use of a mailbox is challenging in the presence of aliasing, *e.g.* we can write a function that attempts to use a mailbox after it has been freed (Fig. 3a). Such a use-after-free error can be excluded with a fully linear type system, since we *cannot* use a resource after it has been consumed.

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

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

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

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

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

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

Here, although the code suggests that x and b are distinct, aliasing is introduced through communication (violating principle 1). The mailbox calculus rules out such programs by constructing a global *dependency graph*. Dependency graphs are well-suited to process calculi since all names are known *a priori*, but are not practical in a programming language due to renaming, nested evaluation contexts, and the distinction between static and dynamic names.

2.2 Quasi-linear typing

The many-sender, single-receiver pattern is closely linked to *quasi-linear typing* [37], although our formulation is closer to [15]. Quasi-linear types were originally designed to overcome some limitations of full linear types in the context of memory management and programming convenience and allow a value to be used once as a *first-class* (returnable) value, but several times as a *second-class* value [43]. A second-class value can be *consumed* within an expression, for example as the subject of a send operation, but cannot escape the scope in which it is defined.

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

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

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

Ruling out aliasing through communication. Quasi-linear types alone do not safeguard against introducing aliasing through communication. 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

$$
\begin{array}{llcl}
\text{Mailbox types} & J, K & ::= & \mathop{!}E \mid \mathop{?}E \\
\text{Mailbox patterns} & E, F & ::= & \mathbb{0} \mid \mathbb{1} \mid \mathtt{m} \mid E \oplus F \\
 & & \mid & E \odot F \mid \star E
\end{array}
\qquad
\begin{array}{lrcl}
\text{Base types} & C & ::= & \mathbf{1} \mid \mathsf{Int} \mid \mathsf{String} \mid \cdots \\
\text{Types} & T, U & ::= & C \mid J \\
\text{Usage annotations} & \eta & ::= & \circ \mid \bullet \\
\text{Usage-annotated types} & A, B & ::= & C \mid J^{\eta}
\end{array}
$$

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

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

not excessively restrictive by expressing all of the examples shown by de'Liguoro and Padovani [12], and 10 of the 11 Savina benchmarks [34] used by Neykova and Yoshida [42] to demonstrate expressiveness of behavioural type systems for actor languages (§6.2).

3 PAT: A CORE LANGUAGE WITH MAILBOX TYPES

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

3.1 Syntax

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

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

Values. It is convenient for typing to introduce a syntactic distinction between values and computations, in part inspired by *fine-grain call-by-value* [39]. Values V, W include variables x and constants c; we assume that the set of constants includes at least the unit value $()$ of type $\mathbf{1}$.

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

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

The $\mathbf{guard}\ V : E\ \{\overrightarrow{G}\}$ expression asserts that mailbox V contains pattern E, and invokes a guard in $\overrightarrow{G}$. The $\mathbf{fail}$ guard is triggered when an unexpected message has arrived; $\mathbf{free} \mapsto M$ is triggered when a mailbox is empty and there are no more references to it in the system; and $\mathbf{receive}\ \mathtt{m}[\overrightarrow{x}]\ \mathbf{from}\ y \mapsto M$ is triggered when the mailbox contains a message with tag $\mathtt{m}$, binding its payloads to $\overrightarrow{x}$ and continuation mailbox with updated mailbox type to y in continuation term

M. We write **free** V as syntactic sugar for **guard** V $\{\textbf{free} \mapsto ()\}$, and **fail** V as syntactic sugar for **guard** V $\{\textbf{fail}\}$. We require that each clause within a **guard** expression is unique.

3.2 Type system

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

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

Pattern semantics. It follows that different syntactic representations of patterns may have the same meaning, *e.g.* patterns $\mathbb{1} \oplus \mathbb{0} \oplus (\mathtt{m} \odot \mathtt{n})$ and $\mathbb{1} \oplus (\mathtt{n} \odot \mathtt{m})$. Following [12], we define a set-of-multisets semantics for mailbox patterns; the intuition is that each multiset defines a configuration of messages that could be present in the mailbox. For example the semantic representation of both of the patterns above is $\{\langle\rangle, \langle \mathtt{m}, \mathtt{n}\rangle\}$. We let A, B range over multisets.

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

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

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

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

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

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

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

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

[2] Unlike in §1, our formal development does not pair a message tag with its payload; instead, tags are associated with payload types via the program signature. This design choice allows us to more easily compare the declarative system with the algorithmic system in §4, and unlike [12] means we do not need to define types and subtyping coinductively.

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

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

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

Base types are subtypes of themselves. As with previous accounts of subtyping in actor languages [26], subtyping is *covariant* for mailbox types with a receive capability: a mailbox can safely be replaced with another that can receive more messages. Likewise subtyping is *contravariant* for mailboxes with a send capability: a mailbox can safely be replaced with another that can send a smaller set of messages. Intuitively, as returnable usages are more powerful than second-class usages, returnable types can be used when only a second-class type is required.

Following [12] we introduce names for particular classes of mailbox types. Intuitively, relevant mailbox names *must* be used, whereas irrelevant names need not be. Likewise reliable and usable names *can* be used, whereas unreliable and unusable names cannot.

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

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

Our type system ensures that variables with a linear type must be used, whereas variables with an unrestricted type can be discarded. We can then extend subtyping to type environments, making it possible to combine type environments, as in [9, 12].

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

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

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

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

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

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

Following [9], it is convenient to identify types up to commutativity and associativity, *e.g.* we do not distinguish between $?(\mathtt{A} \odot \mathtt{B})^{\bullet}$ and $?(\mathtt{B} \odot \mathtt{A})^{\bullet}$. 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 $?(\star\mathtt{A})$, we would need to use subtyping to first rewrite the latter type to $?(\mathtt{A} \odot \star\mathtt{A})$).

The *usage combination* operator combines usages: it is not commutative as a $\circ$ use of a variable can only occur *before* a $\bullet$ use (ensuring that the returnable use is the last lexical use of a variable). Furthermore, note that $\bullet \triangleright \bullet$ is undefined (ensuring that there is only one returnable instance of a variable per thread).

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

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

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

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

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

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

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

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

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

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

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

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

Rule T-Spawn types spawning a term M of unit type as a new process. The type environment used to type M can contain any number of returnable, but the conclusion of the rule 'masks' any returnable types as second-class. Intuitively, this is because there is no need to impose an ordering on how a variable is used in a separate process. So 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 $?\mathbb{1}^\bullet$, since subsequent sends and receives must "balance out" to an empty mailbox.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 5. Pat declarative term typing

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

Rule T-Guard types the expression $\mathbf{guard}\ V : E\ \{\overrightarrow{G}\}$, that retrieves from mailbox V with some pattern E using guards $\overrightarrow{G}$. The first premise ensures that under a type environment Γ_1, mailbox V has type $?F^{\bullet}$: the mailbox should have a receive capability with pattern F, and must be returnable. Demanding that the mailbox is returnable rules out use-after-free errors since we cannot use the mailbox name in the continuation. The second premise states that under type environment Γ_2, guards $\overrightarrow{G}$ all return a value of type A and correspond to pattern F. The third premise requires that the pattern assertion E is contained within F. The final premise, $\vDash F$, ensures that F is in *pattern* *normal form*: the pattern should be a disjunction of pattern literals. That is $\mathbb{0}$, $\mathbb{1}$, or $\mathtt{m} \odot F$, where F is equivalent to E without message $\mathtt{m}$.

Finally, rule T-Subs 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-Subs makes it possible to rewrite mailbox types so that they can be combined by the type combination operators. We also allow the usual use of subsumption on return types, *e.g.* allowing a value with a subtype of a function argument to be used.

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

Pattern residual. The *pattern residual* $E \,/\, \mathtt{m}$ calculates the pattern E after $\mathtt{m}$ is consumed, and corresponds to the Brzozowski derivative [6] over a commutative regular expression. The residual of $\mathbb{0}$, $\mathbb{1}$, or $\mathtt{n}$ (where $\mathtt{n} \neq \mathtt{m}$) with respect to a message tag $\mathtt{m}$ is the unreliable type $\mathbb{0}$. The derivative of $\mathtt{m}$ with respect to $\mathtt{m}$ is $\mathbb{1}$. The derivative operator distributes over $\oplus$, and the derivative of concatenation is the disjunction of the derivative of each subpattern.

Example. We end this section by showing the derivation for part of the future example from §1, specifically, the body of the client definition which creates a future and self mailbox, initialises the future with a number, and then requests and prints the result. In the following, we abbreviate *future* to f, *self* to s, and *result* to r. We assume that the program includes a signature $\mathcal{S} = [\mathtt{Put} \mapsto \mathsf{Int}, \mathtt{Get} \mapsto\ !\mathtt{Reply}, \mathtt{Reply} \mapsto \mathsf{Int}]$, and the emptyFuture and fullFuture definitions from §1.

We split the derivation into three subderivations. Since it is easier to read derivations top-down, we start by typing the **guard** expression. In the following, we refer to the **receive** guard as G, and name the first derivation $\mathbf{D}_1$:

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

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

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

Runtime syntax

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

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

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

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

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

Fig. 6. Pat operational semantics

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

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

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

3.3 Operational Semantics

Figure 6 shows the runtime syntax and reduction rules for Pat. We extend values V with runtime names a. The concurrent semantics of the language is described as a nondeterministic reduction relation on a language of *configurations*, which resemble terms in the π-calculus. Configuration $(\!| M, \Sigma |\!)$ is a thread evaluating term M, with frame stack Σ; we will discuss frame stacks in the next section. Configuration $a \leftarrow \mathtt{m}[\overrightarrow{V}]$ denotes a message $\mathtt{m}[\overrightarrow{V}]$ in mailbox a; name restriction $(\nu a)\mathcal{C}$ binds name a in $\mathcal{C}$; and $\mathcal{C} \parallel \mathcal{D}$ denotes the parallel composition of $\mathcal{C}$ and $\mathcal{D}$.

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

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

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

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

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

Fig. 7. Pat runtime typing

Frame stacks. We use explicit frame stacks [15, 49] rather than evaluation contexts for technical convenience. A frame $\langle x, M \rangle$ is a pair of a variable x and a continuation M, where x is free in M. A frame stack is an ordered sequence of frames, where ϵ denotes the empty stack.

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

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

3.4 Metatheory

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

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

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

$$\frac{x \notin \text{dom}(\Delta_2) \qquad \Delta_1 \bowtie \Delta_2 = \Delta}{(\Delta_1, x : T) \bowtie \Delta_2 = \Delta, x : T} \qquad \frac{x \notin \text{dom}(\Delta_1) \qquad \Delta_1 \bowtie \Delta_2 = \Delta}{\Delta_1 \bowtie (\Delta_2, x : T) = \Delta, x : T} \qquad \frac{\Delta_1 \bowtie \Delta_2 = \Delta}{(\Delta_1, x : T) \bowtie (\Delta_2, x : U) = \Delta, x : (T \boxplus U)}$$

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

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

Thread and frame stack typing. Rule TC-Thread types a thread, which is a pair of a currently-evaluating term, typable under an environment Γ_1, and a stack frame, typable under an environment Γ_2. The combination $\Gamma_1 \triangleright \Gamma_2$ should result in the returnable lifting of Δ: intuitively, we should be able to use every mailbox variable in Δ as returnable in the thread. TC-Thread makes use of the frame stack typing judgement $\Gamma \vdash A \blacktriangleright \Sigma$ (inspired by [15]), which can be read "under type environment Γ, given a value of type A, frame stack Σ is well-typed". The empty frame stack is typable under the empty environment given any type. A non-empty frame stack $\langle x, M \rangle \cdot \Sigma$ is well-typed if M has some returnable type B, given a variable x of type A. The remainder of the stack must then be well-typed given B. We combine the environments used for typing the head term and the remainder of the stack using $\triangleright$ as we wish to account for sequential uses of a mailbox; for example, in the term $x\,!\,\mathtt{m}[V]; x\,!\,\mathtt{n}[W]$, x would have type $!(\mathtt{m} \odot \mathtt{n})^{\circ}$.

3.4.2 *Properties.*

We can now state some metatheoretical results. We relegate proofs to Appendix E.

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

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

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

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

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

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

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

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

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

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

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

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

Then for each M_i, either:

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

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

Fig. 8. Pat syntax extended for algorithmic typing

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

The key consequence of Theorem 3.16 is the absence of self-deadlocks: since we can only guard on a returnable mailbox, and a returnable name must be the last occurrence in the thread, it cannot be that the **guard** expression is blocking a send to the same mailbox in the same thread.

Remark. *The original formulation of mailbox typing in a process calculus [12] provides a global progress result by exploiting a dependency graph to eliminate cyclic dependencies and hence deadlocks. A language implementation cannot use this approach as it relies on knowing runtime names directly. However quasi-linear typing still allows us to rule out self-deadlocking interactions.*

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 [45] is a typechecker for the mailbox calculus, based on a typechecker for concurrent object usage protocols [46]. The type system used in MC^2 has, however, not been formalised. We use several ideas from MC^2, including algorithmic type combination operations, and adapt the approach for a programming language.

This section describes a co-contextual [16] algorithmic type system based on *backwards bidirectional typing* [60]. The key idea is to *construct* a type environment based on how mailbox variables are used, along with a set of pattern inclusion constraints.

4.1 Algorithmic Type System

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

Constraints. A key challenge for the algorithmic type system is determining whether one pattern is *included* within another: *e.g.* $\mathtt{m} \sqsubseteq \star\mathtt{m}$. Given that patterns may contain pattern variables, we may need to defer inclusion checking until more pattern variables are known, so we introduce inclusion constraints $\gamma <: \delta$ which require that pattern γ is included in pattern δ. We write the equivalence constraint $\gamma \sim \delta$ as syntactic sugar for the constraint set $\{\gamma <: \delta, \delta <: \gamma\}$ and abuse notation to treat $\gamma \sim \delta$ as a single constraint.

4.1.1 *Algorithmic type operations.*

Fig. 9 shows the algorithmic type combination operators.

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

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

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

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

$$\frac{}{\eta \leq \eta} \qquad \frac{}{\bullet \leq \circ} \qquad \frac{}{C \leq C \blacktriangleright \emptyset} \qquad \frac{\eta_1 \leq \eta_2}{!\gamma^{\eta_1} \leq\, !\delta^{\eta_2} \blacktriangleright \delta <: \gamma} \qquad \frac{\eta_1 \leq \eta_2}{?\gamma^{\eta_1} \leq\, ?\delta^{\eta_2} \blacktriangleright \gamma <: \delta} \qquad \frac{\tau \leq \sigma \blacktriangleright \Phi_1 \quad \sigma \leq \tau \blacktriangleright \Phi_2}{\tau \sim \sigma \blacktriangleright \Phi_1 \cup \Phi_2}$$

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

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

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

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

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

Fig. 9. Pat algorithmic type operations

capability are subtypes of each other by generating a contravariant constraint for a send type, and a covariant constraint for a receive type.

Algorithmic type join. Declarative mailbox typing relies on the subtyping rule to manipulate types into a form where they can be combined with the type combination operators, e.g., $!E \boxplus ?(E \odot F) = ?F$. The algorithmic type system cannot apply the same technique as it does not know, *a priori*, the form of each pattern. Instead, the algorithmic *type join* operation allows the combination of two mailbox types irrespective of their syntactic form. Combining two send types is the same as in the declarative system, but combining a send type with a receive type (and vice versa) is more interesting: say we wish to combine $!\gamma$ and $?\delta$. In this case, we generate a fresh pattern variable α; the result is $?\alpha$ along with the constraint that $(\gamma \odot \alpha) <: \delta$: namely, that the send pattern concatenated with the fresh pattern variable is included in the pattern δ.

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

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

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

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

$$\frac{\mathcal{P} = (\mathcal{S}, \overrightarrow{D}, M) \qquad (\vdash_{\mathcal{P}} D_i \triangleright \Phi_i)_i \qquad M \Leftarrow \mathbf{1} \blacktriangleright \cdot; \Phi}{\vdash \mathcal{P} \triangleright \Phi \cup \Phi_1 \cup \cdots \cup \Phi_n} \qquad \frac{M \Leftarrow \sigma \blacktriangleright \Theta; \Phi_1 \qquad \mathsf{check}(\overrightarrow{x}, \overrightarrow{\tau}, \Theta) = \Phi_2 \qquad \Theta - \overrightarrow{x} = \cdot}{\vdash_{\mathcal{P}} \textbf{def}\ f(\overrightarrow{x : \tau}) : \sigma\ \{M\} \triangleright \Phi_1 \cup \Phi_2}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Type system overview. Our algorithmic type system takes a *co-contextual* [16] approach: rather than taking a type environment as an *input* to the type-checking algorithm, we produce a type environment as an *output*. The intuition is that (read bottom-up), *splitting* an environment into two sub-environments is more difficult than *merging* two environments inferred from subexpressions. We also generate *inclusion constraints* on patterns to be solved later.

Bidirectional type systems [14, 48] 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 [60]. The key twist is the variable rule, which becomes a *checking* rule and records the given variable-type mapping in the inferred environment. Our synthesis judgement has the form $M \Rightarrow_{\mathcal{P}} \tau \blacktriangleright \Theta; \Phi$, which can be read "synthesise type τ for term M under program $\mathcal{P}$, inferring type environment Θ and producing constraints Φ". The checking judgement $M \Leftarrow_{\mathcal{P}} \tau \blacktriangleright \Theta; \Phi$ is defined analogously. As in the declarative system we omit the $\mathcal{P}$ annotation for readability.

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.

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

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

Message sending $V\,!\,\mathtt{m}[\overrightarrow{W}]$ is a side-effecting operation, and so we synthesise type $\mathbf{1}$. Rule TS-Send first looks up the payload types $\overrightarrow{\pi}$ in the signature, and *checks* that message target V has mailbox type $!\mathtt{m}^{\circ}$. In performing this check, the type system will produce environment Θ' that 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 $\mathbf{let}\; x\colon T = M \;\mathbf{in}\; N$ has type τ: first, we check that M has type $\lfloor T \rfloor$ noting that only values of returnable type may be returned, producing environment Θ_1 and constraints Φ_1. Next we check that the body N has type τ, producing environment Θ_2 and Φ_2. The next step is to check whether the types of the variable inferred in Θ_2 corresponds with the annotation. The check meta-function ensures that if x is not contained within Θ_2, then the type of x is unrestricted; and conversely if x *is* contained within Θ_2, then the annotation is a subtype of the inferred type as the annotation is a *lower bound* on what the body can expect of x.

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

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

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

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

Rule TC-Guard checks that a guard expression $\mathbf{guard}\; V : E\,\{\overrightarrow{G}\}$ has return type τ. First, the rule checks that the guard sequence $\overrightarrow{G}$ has type τ, producing nullable environment Ψ, constraint set Φ_1, and pattern F in pattern normal form. Next, the rule checks that the mailbox name V has type $?F^{\bullet}$, producing environment Θ' and constraint set Φ_2. Finally, the rule calculates the disjoint combination of Ψ and Θ', producing final environment Θ and constraints Φ_3.

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

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

$$\textsc{TCG-Fail}\ \frac{}{\{E\}\ \textbf{fail} \Leftarrow \tau \blacktriangleright \top;\ \emptyset;\ \mathbb{0}} \qquad \textsc{TCG-Free}\ \frac{M \Leftarrow \tau \blacktriangleright \Theta;\ \Phi}{\{E\}\ \textbf{free} \mapsto M \Leftarrow \tau \blacktriangleright \Theta;\ \Phi;\ \mathbb{1}}$$

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

$$\Theta - \overrightarrow{x} \triangleq \{y : \tau \in \Theta \mid y \notin \overrightarrow{x}\}$$

Fig. 11. Pat algorithmic typing (guards)

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

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

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

4.2 Metatheory

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

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

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

4.2.1 Algorithmic soundness.

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

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

Theorem 4.4 (Algorithmic Soundness).

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

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

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

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

Constraint solving. Padovani [46] shows how to solve a constraint set, relying on a closed-form solution [30]. Since the procedure is not novel, we simply provide an overview in Appendix B.

5 EXTENSIONS

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

5.1 Product and Sum Types

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

An advantage of adding product types is that we can avoid nested **guard** clauses, as we can return both a received value and an updated mailbox name. Consider the following examples of a process that receives two integers and returns their sum. The example on the left requires nested **guard** expressions, whereas the example on the right does not.

```
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
}
```

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

5.2 Using Contextual Type Information

A co-contextual approach is required to generate the pattern inclusion constraints. Sometimes, however, it is useful to have contextual type information *before* the constraint generation pass.

Consider applying a first-class function: $(\lambda(x : \mathsf{Int}) \colon \mathsf{Int}\,.\,x)(5)$. Although the annotated λ expression allows us to synthesise a type and use a rule similar to TS-App, the lack of contextual type information means that the approach founders as soon as we stray from applying function literals as in $\textbf{let}\, f = (\lambda(x : \mathsf{Int}) \colon \mathsf{Int}\,.\,x)\ \textbf{in}\ f(5)$). A typical backwards bidirectional typing approach requires *synthesising* the argument to a function, but this is too inflexible in our setting as each mailbox argument would need a type annotation at the application site.

```
guard self : Ready1 ⊙ Ready2 {
  receive Ready1[reply1] from mb′ ↦
    guard mb′ : Ready2 {
      receive Ready2[reply2] from mb″ ↦
        reply1 ! Go[]; reply2 ! Go[];
        free mb″
    }
}
```

Fig. 12. Term requiring interfaces

In the base system a global signature maps message tags to payload types. While technically convenient, this is inflexible. First, distinct entities may wish to use the same mailbox tags with different payload types. For example, a client may send a **Login** message containing credentials to a server, which may then send a **Login** message containing the credentials and a timestamp to a session management server. Second, we need a syntactic check on a **receive** guard to avoid aliasing, as outlined in §2: either the received payloads or free variables in the guard body must be base types. This conservative check rules out innocuous cases such as in Figure 12, which waits for two actors to both be ready before signalling them to continue.

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

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

6 IMPLEMENTATION AND EXPRESSIVENESS

We outline the implementation of a prototype type checker written in OCaml [59], and evidence the expressiveness of Pat via a selection of example programs taken from the literature. We first show that using quasi-linear typing in place of dependency graphs (cf. §2.2) does not prevent Pat from expressing *all* of the examples in [12]. The Savina benchmarks [34] capture typical concurrent communication patterns and are used both to compare actor languages and to demonstrate the expressiveness of programming models, *e.g.* Neykova and Yoshida [42]. We show that Pat can express 10 of the 11 Savina expressiveness benchmarks used in [42]. Finally, we encode a case study provided by an industrial partner that develops highly concurrent control software for factories.

6.1 Implementation Overview

Pat programs are type checked in a six-phase pipeline:

(1) **Parsing**. Standard lexing and parsing using the OCaml Menhir library, producing the AST;
(2) **Desugaring**. Traverses the AST to expand the sugared form of guards (*i.e.* rewrites **free** V as **guard** $V : \mathbb{1}\ \{\textbf{free} \mapsto ()\}$ and **fail** V as **guard** $V : \mathbb{0}\ \{\textbf{fail}\}$) and adds omitted pattern variables;
(3) **IR conversion**. Transforms the surface language (supporting nested expressions) to our explicitly-sequenced intermediate representation;
(4) **Contextual type-checking**. Performs a (standard) typing pass to propagate contextual type information (refer to §5.2 for details);

#	Name	Description	Strict	Time (ms)
Original mailbox calculus models taken from de'Liguoro and Padovani [12]				
1	Lock	Concurrent lock modelling mutual exclusion	•	28.5
2	Future	Future variable that is written to once and read multiple times	•	22.5
3	Account	Concurrent accounts exchanging debit and credit instructions	•	19.5
4	AccountF	Concurrent accounts where debit instructions are effected via futures	•	33.5
5	Master-Worker	Master-worker parallel network	•	29.0
6	Session Types	Session-typed communicating actors using one arbiter	◦	75.5
Selected micro-benchmarks adapted from Imam and Sarkar [34], based on Neykova and Yoshida [42]				
7	Ping Pong	Process pair exchanging k ping and pong messages	•	24.6
8	Thread Ring	Ring network where actors cyclically relay one token with counter k	◦	37.2
9	Counter	One actor sending messages to a second that sums the count, k	◦	29.8
10	K-Fork	Fork-join pattern where a central actor delegates k requests to workers	•	7.1
11	Fibonacci	Fibonacci server delegating terms $(k-1)$ and $(k-2)$ to parallel actors	•	27.1
12	Big	Peer-to-peer network where actors exchange k messages randomly	◦	62.8
13	Philosopher	Dining philosophers problem	◦	57.1
14	Smokers	Centralised network where one arbiter allocates k messages to actors	◦	31.3
15	Log Map	Computes the term $x_{k+1} = r \cdot x_k(1 - x_k)$ by delegating to parallel actors	◦	57.9
16	Transaction	Request-reply actor communication initiated by a central teller actor	◦	46.7

Tbl. 1. Typechecking concurrent actor examples in Pat

(5) **Constraint generation**. Implements the algorithmic type system from §4 and generates a set of pattern inclusion constraints;
(6) **Constraint solving**. Applies the constraint-solving approach given in Appendix B, and invokes the Z3 SMT solver [11] to determine whether the constraints generated in (5) are satisfiable.

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

6.2 Expressiveness and Typechecking Time

Tbl. 1 lists the examples implemented in Pat. Examples 1-6 are the mailbox calculus examples from [12, Ex. 1–3, and Sec. 4.1–4.3]. Examples 7-16 are the selection of Savina benchmarks [34, Table 1, No. 1–4, 6, 7, 12, 14–16] used in [42]. The table indicates whether a Pat program can be checked in strict (denoted by •), *in addition* to interface mode (denoted by ◦). We report the mean typechecking time, excluding phases 1–3 of the pipeline. Measurements are made on a MacBook M1 Pro with 16GB of memory, running macOS 13.2 and OCaml 5.0, and averaging over 1000 repetitions.

6.2.1 Benchmarks. Tbl. 1 shows that all but one of the mailbox calculus examples from [12] can be checked in strict mode. The Savina examples capture typical concurrent programming patterns, namely, master-worker (K-Fork, Fibonacci, Log Map), client-server (Ping Pong, Counter), and peer-to-peer (Big), and common network topologies such as star (Philosopher, Smokers, Transaction) and ring (Thread Ring). Most of these programs require contextual type information (8, 9, and 12–16) to type check. As Pat does not yet support recursive types, we instead emulate fixed collections using definition parameters in examples 8, 10, 12–16. We could not encode the Sleeping Barber [42, Ex. 8] example since the number of collection elements varies throughout execution.

The examples reveal the benefits of mailbox typing. Boilerplate 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.* [42, 53]. In the latter approach, one typically specifies protocols in external tools and writes code to accommodate the session typing framework. By contrast, mailbox typing *naturally* fits idiomatic actor programming. This flexibility does not incur high typechecking runtime (see Tbl. 1).

6.2.2 *Case Study.*

Finally we describe a real-world use case written by *Actyx AG* [1], who develop control software for factories. The use case captures a scenario where multiple robots on a factory floor acquire parts from a warehouse that provides access through a single door. Robots negotiate with the door to gain entry into the warehouse and obtain the part they require. The behaviour of our three entities, *Robot*, *Door*, and *Warehouse* is shown in Fig. 13. Our concrete syntax closely follows the core calculus of §3, without requiring that pattern variables in mailbox types are specified explicitly. Type checking our completed case study given in Appendix D.2 relies on contextual type information (see §5), and takes ≈89.6 ms.

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

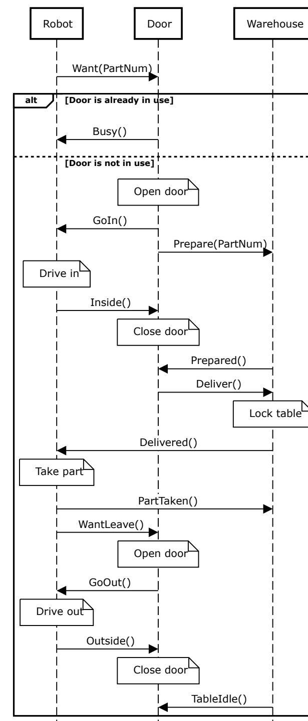


Fig. 13. Factory use case

```
1 def empty(self: wh?): Unit {
2   guard self: Prepare + 1 {
3     free → ()
4     receive Prepare(partNum, door) from self →
5       door ! Prepared(self);
6       engaged(self)
7   }
8 }
```

```
9  def engaged(self: wh?): Unit {
10   guard self: Deliver {
11     receive Deliver(robot, door) from self →
12       robot ! Delivered(self, door);
13       given(self, door)
14   }
15 }
```

7 RELATED WORK

TAkka [26] introduced typed actor communication for Akka [50], where PIDs are parameterised by the type an actor may receive. The authors uncover the *type pollution problem*, where an actor reference must expose all types it can receive, and show how it can be mitigated via subtyping. Akka Typed (now standard in Akka) is inspired by TAkka. Fowler et al. [20] characterise core calculi for typed channels and actors and give translations between the two models. They show that modelling channels with actors is more complex than the modelling actors with channels, underlining the expressiveness mismatch, and show that synchronisation alleviates the type pollution problem; we can achieve a similar effect using multiple mailboxes (*e.g.* as done in example 3 of Tbl. 1).

Developing behavioural type systems for actor languages is challenging due to the unidirectional and asymmetric nature of mailboxes. Mostrous and Vasconcelos [40] investigate session typing for

Core Erlang, using selective message reception and unique references to encode binary session-typed channels. Tabone and Francalanza [52, 53] develop a tool that statically checks Elixir [35] actors against binary session types to prove session fidelity. In contrast, our system is more general and supports many-to-one mailbox communication.

Neykova and Yoshida [42] propose a programming model for dynamically checking actor communication against multiparty session types [29], which was later implemented in Erlang [18]. Later work [41] shows how causality information in global types can support efficient recovery strategies. Harvey et al. [25] use multiparty session types with explicit connection actions [31] to give strong guarantees about actors that support dynamic discovery and code replacement, although an actor can only participate in one session at a time. Using session types to structure communication requires specifying point-to-point interactions, typically using different libraries and formalisms. In contrast, our mailbox typing approach naturally fits idiomatic actor programming paradigms.

Active objects [10] are actor-like entities, which return the result of a remote method invocation via a future. Bagherzadeh and Rajan [4] define a type system for active objects which can rule out data races; unlike our approach, this work targets an imperative calculus and is not validated via an implementation. Kamburjan et al. [36] apply session-based reasoning to a core active object calculus where types encode remote calls and future resolutions. In their calculus, communication correctness is ensured by static checks against session automata [5] derived from session types.

Mailbox types are inspired by behavioural type systems [9] for the *objective join calculus* [17]. The technique can be implemented in Java using code generation via *matching automata* [22], and dependency graphs can rule out deadlocks [44], but the authors do not consider a programming language design. Scalas et al. [51] 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.

Christakis and Sagonas [8] implement a static analysis pass for Erlang that detects communication errors such as receiving when a mailbox is empty, payload mismatches, redundant patterns, and orphan messages. All of these issues can be detected with mailbox types, and mailbox types additionally allow us to specify the mailbox state. Harrison [24] implements an approach incorporating aspects of both typechecking and static analysis to detect message passing errors such as orphan messages and redundant patterns.

8 CONCLUSION AND FUTURE WORK

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

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

REFERENCES

[1] 2023. *Actyx AG*. https://actyx.io

[2] Roberto M. Amadio, Ilaria Castellani, and Davide Sangiorgi. 1998. On Bisimulations for the Asynchronous pi-Calculus. *Theor. Comput. Sci.* 195, 2 (1998), 291–324.

[3] Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. 2016. Behavioral Types in Programming Languages. *Found. Trends Program. Lang.* 3, 2-3 (2016), 95–230. https://doi.org/10.1561/2500000031

[4] Mehdi Bagherzadeh and Hridesh Rajan. 2017. Order types: static reasoning about message races in asynchronous message passing concurrency. In *AGERE!@SPLASH*. ACM, 21–30.

[5] Benedikt Bollig, Peter Habermehl, Martin Leucker, and Benjamin Monmege. 2013. A Fresh Approach to Learning Register Automata. In *Developments in Language Theory (LNCS, Vol. 7907)*. Springer, 118–130.

[6] Janusz A Brzozowski. 1964. Derivatives of regular expressions. *Journal of the ACM (JACM)* 11, 4 (1964), 481–494.

[7] Avik Chaudhuri. 2009. A Concurrent ML library in Concurrent Haskell. In *ICFP*. ACM, 269–280.

[8] 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.

[9] Silvia Crafa and Luca Padovani. 2017. The Chemical Approach to Typestate-Oriented Programming. *ACM Trans. Program. Lang. Syst.* 39, 3 (2017), 13:1–13:45.

[10] 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.

[11] 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.

[12] 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.

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

[14] Jana Dunfield and Neel Krishnaswami. 2022. Bidirectional Typing. *ACM Comput. Surv.* 54, 5 (2022), 98:1–98:38.

[15] Robert Ennals, Richard Sharp, and Alan Mycroft. 2004. Linear Types for Packet Processing. In *Programming Languages and Systems, 13th European Symposium on Programming, ESOP 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 2986)*, David A. Schmidt (Ed.). Springer, 204–218. https://doi.org/10.1007/978-3-540-24725-8_15

[16] 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.

[17] Cédric Fournet and Georges Gonthier. 1996. The Reflexive CHAM and the Join-Calculus. In *POPL*. ACM Press, 372–385.

[18] Simon Fowler. 2016. An Erlang Implementation of Multiparty Session Actors. In *ICE (EPTCS, Vol. 223)*. 36–50.

[19] Simon Fowler, Wen Kokke, Ornela Dardha, Sam Lindley, and J. Garrett Morris. 2021. Separating Sessions Smoothly. In *CONCUR (LIPIcs, Vol. 203)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 36:1–36:18.

[20] 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.

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

[22] Rosita Gerbo and Luca Padovani. 2019. Concurrent Typestate-Oriented Programming in Java. In *PLACES@ETAPS (EPTCS, Vol. 291)*. 24–34.

[23] Seymour Ginsburg and Edwin Spanier. 1966. Semigroups, Presburger formulas, and languages. *Pacific journal of Mathematics* 16, 2 (1966), 285–296.

[24] Joseph R. Harrison. 2018. Automatic detection of core Erlang message passing errors. In *Erlang Workshop*. ACM, 37–48.

[25] 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.

[26] Jiansen He, Philip Wadler, and Philip W. Trinder. 2014. Typecasting actors: from Akka to TAkka. In *SCALA@ECOOP*. ACM, 23–33.

[27] Kohei Honda. 1993. Types for Dyadic Interaction. In *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings (Lecture Notes in Computer Science, Vol. 715)*, Eike Best (Ed.). Springer, 509–523. https://doi.org/10.1007/3-540-57208-2_35

[28] 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.

[29] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2016. Multiparty Asynchronous Session Types. *J. ACM* 63, 1 (2016), 9:1–9:67.

[30] Mark W. Hopkins and Dexter Kozen. 1999. Parikh's Theorem in Commutative Kleene Algebra. In *LICS*. IEEE Computer Society, 394–401.

[31] 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.

[32] 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.

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

[34] Shams Mahmood Imam and Vivek Sarkar. 2014. Savina - An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries. In *AGERE!@SPLASH*. ACM, 67–80.

[35] Saša Jurić. 2019. *Elixir in Action*. Manning.

[36] Eduard Kamburjan, Crystal Chang Din, and Tzu-Chun Chen. 2016. Session-Based Compositional Analysis for Actor-Based Languages Using Futures. In *ICFEM (Lecture Notes in Computer Science, Vol. 10009)*. 296–312.

[37] Naoki Kobayashi. 1999. Quasi-Linear Types. In *POPL*. ACM, 29–42.

[38] 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.

[39] 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.

[40] 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.

[41] Rumyana Neykova and Nobuko Yoshida. 2017. Let it recover: multiparty protocol-induced recovery. In *CC*. ACM, 98–108.

[42] Rumyana Neykova and Nobuko Yoshida. 2017. Multiparty Session Actors. *Log. Methods Comput. Sci.* 13, 1 (2017).

[43] 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.

[44] Luca Padovani. 2018. Deadlock-Free Typestate-Oriented Programming. *Art Sci. Eng. Program.* 2, 3 (2018), 15.

[45] Luca Padovani. 2018. *Mailbox Calculus Checker*. https://boystrange.github.io/mcc/

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

[47] Rohit Parikh. 1966. On Context-Free Languages. *J. ACM* 13, 4 (1966), 570–581.

[48] Benjamin C. Pierce and David N. Turner. 2000. Local type inference. *ACM Trans. Program. Lang. Syst.* 22, 1 (2000), 1–44.

[49] Andrew M. Pitts. 1998. Existential Types: Logical Relations and Operational Equivalence. In *ICALP (Lecture Notes in Computer Science, Vol. 1443)*. Springer, 309–326.

[50] Raymond Roestenburg, Rob Bakker, and Rob Williams. 2015. *Akka in Action*. Manning.

[51] Alceste Scalas, Nobuko Yoshida, and Elias Benussi. 2019. Verifying message-passing programs with dependent behavioural types. In *PLDI*. ACM, 502–516.

[52] Gerard Tabone and Adrian Francalanza. 2021. Session types in Elixir. In *AGERE!@SPLASH*. ACM, 12–23.

[53] 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.

[54] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. 1994. An Interaction-based Language and its Typing System. In *PARLE '94: Parallel Architectures and Languages Europe, 6th International PARLE Conference, Athens, Greece, July 4-8, 1994, Proceedings (Lecture Notes in Computer Science, Vol. 817)*, Constantine Halatsis, Dimitris G. Maritsas, George Philokyprou, and Sergios Theodoridis (Eds.). Springer, 398–413. https://doi.org/10.1007/3-540-58184-7_118

[55] 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.

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

[57] Vasco T. Vasconcelos. 2012. Fundamentals of session types. *Inf. Comput.* 217 (2012), 52–70.

[58] Philip Wadler. 2014. Propositions as sessions. *J. Funct. Program.* 24, 2-3 (2014), 384–418.

[59] John Whitington. 2013. *OCaml from the Very Beginning*. Coherent Press.

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

Appendices

APPENDIX CONTENTS

A OMITTED DEFINITIONS

Here we add in the omitted definitions from the main body of the paper.

A.1 Algorithmic environment combination operators

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

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

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

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

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

$$\frac{x \notin \mathrm{dom}(\Theta_1) \qquad \Theta_1 \sqcap \Theta_2 \blacktriangleright \Theta; \Phi_2}{\Theta_1 \sqcap \Theta_2, x{:}C \blacktriangleright \Theta, x{:}\tau; \Phi} \qquad \frac{x \notin \mathrm{dom}(\Theta_1) \qquad \Theta_1 \sqcap \Theta_2 \blacktriangleright \Theta; \Phi_2}{\Theta_1 \sqcap \Theta_2, x{:}!\gamma^{\eta} \blacktriangleright \Theta, x{:}!(\gamma \oplus \mathbb{1})^{\eta}; \Phi}$$

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

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

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

$$\frac{\begin{array}{cc}\Theta_1 + \Theta_2 \blacktriangleright \Theta; \Phi_1 & \tau \sim \sigma \blacktriangleright \Phi_2 \\ \mathrm{unr}(\tau) \blacktriangleright \Phi_3 & \mathrm{unr}(\sigma) \blacktriangleright \Phi_4\end{array}}{\Theta_1, x{:}\tau + \Theta_2, x{:}\sigma \blacktriangleright \Theta, x{:}\tau; \Phi_1 \cup \cdots \cup \Phi_4}$$

The environment join operator $\fatsemi\Theta_1\Theta_2\Theta\Phi$ concatenates Θ_1 and Θ_2, computing the algorithmic type join of any types for overlapping variables, and produces constraints Φ.

The environment merge computes the algorithmic type merge of any overlapping types. If a variable is in one environment but not another, then if it is a base type, it is simply added to the output environment. If it is a send mailbox type $!\gamma^{\eta}$, then its type is changed to $!(\gamma \oplus \mathbb{1})^{\eta}$ to denote the fact that it may not be used.

Disjoint environment combination combines two environments; if there are two overlapping types then they must be equivalent and unrestricted.

B CONSTRAINT SOLVING OVERVIEW

This appendix outlines how to solve and check the satisfiability of the pattern inclusion constraints generated by the algorithmic type system. As this process isn't novel, and is covered in depth elsewhere [46], we provide only an informal overview.

Identify and group bounds A *pattern bound* is of the form $\gamma <: \alpha$ i.e. a constraint whose right-hand-side is a pattern variable. The first step is to identify and group all pattern bounds using pattern disjunction: for example, given a constraint set $\{\gamma <: \alpha, \delta \sqsubseteq \beta, \mathbb{1} <: \mathbb{1}\}$ we would produce a grouped constraint $\gamma \oplus \delta <: \alpha$.

Calculate closed-form solutions Hopkins and Kozen [30] define a closed-form solution for pattern bounds: given a set of pattern bound constraints $(\gamma_i <: \alpha_i)_i$ there exists a pattern $\delta_i \simeq \gamma_i$ for each γ_i such that $\alpha_i \notin \mathrm{pv}(\delta_i)$. We can then substitute each closed pattern throughout the set of pattern bound constraints to obtain a set of closed pattern bounds, providing a mapping from pattern variables to closed patterns. This allows us to substitute out all pattern variables in the remaining constraints to obtain a system of closed inclusion constraints.

Translate to Presburger formulae and check satisfiability Finally, we translate the system of inclusions into Presburger formulae. Commutative regular expressions, and therefore patterns, can be expressed as semilinear sets [47] that describe Presburger formulae [23]. Since checking the satisfiability of a Presburger formula is decidable, an external solver like Z3 [11] can be used to determine whether each constraint holds.

C DETAILS OF EXTENSIONS

Here we discuss the details of the extensions overviewed in §5.

C.1 Product and sum types

Product and sum types can be added relatively straightforwardly. In both cases, their components must be returnable as otherwise it would not be possible to substitute their contents upon deconstruction.

Product types. We can write the declarative typing rules for products as follows; the rules are unremarkable apart from the requirement that each component of the pair must be returnable in the elimination form.

$$\textsc{T-Pair}\quad \frac{\Gamma_1 \vdash V : A \qquad \Gamma_2 \vdash W : B}{\Gamma_1 + \Gamma_2 \vdash (V, W) : A \times B}$$

$$\textsc{T-LetPair}\quad \frac{\Gamma_1 \vdash V : A_1 \times A_2 \qquad \mathsf{returnable}(A) \qquad \mathsf{returnable}(B) \qquad \Gamma_2, x : A_1, y : A_2 \vdash V : B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\ (x, y) = V\ \mathbf{in}\ M : B}$$

We can write the corresponding algorithmic rules as follows:

$$\textsc{TC-Pair}\quad \frac{V \Leftarrow \tau \blacktriangleright \Theta_1; \Phi_1 \qquad W \Leftarrow \sigma \blacktriangleright \Theta_2; \Phi_2 \qquad \Theta_1 + \Theta_2 \blacktriangleright \Theta; \Phi_3}{(V, W) \Leftarrow \tau \times \sigma \blacktriangleright \Theta; \Phi_1 \cup \Phi_2 \cup \Phi_3}$$

$$\textsc{TC-LetPair}\quad \frac{\begin{array}{c} V \Leftarrow \tau_1 \times \tau_2 \blacktriangleright \Theta_1; \Phi_1 \qquad \mathsf{returnable}(\tau_1) \qquad \mathsf{returnable}(\tau_2) \qquad M \Leftarrow \tau \blacktriangleright \Theta_2; \Phi_2 \\ \mathsf{check}(\Theta_2, x, A_1) = \Phi_3 \qquad \mathsf{check}(\Theta_2, y, A_2) = \Phi_4 \\ \Theta_1 + \Theta_2 \blacktriangleright \Theta; \Phi_5 \end{array}}{\mathbf{let}\ (x, y)\colon (\tau_1 \times \tau_2) = V\ \mathbf{in}\ M \Leftarrow \tau \blacktriangleright \Theta; \Phi_1 \cup \cdots \cup \Phi_5}$$

$$\textsc{TC-LetPairNoAnn}\quad \frac{\begin{array}{c} M \Leftarrow B \blacktriangleright \Theta, x : \tau_1, y : \tau_2; \Theta_1 \Phi_1 \\ \mathsf{returnable}(\tau_1) \qquad \mathsf{returnable}(\tau_2) \qquad V \Leftarrow \tau_1 \times \tau_2 \blacktriangleright \Theta_2; \Phi_2 \qquad \Theta_1 + \Theta_2 \blacktriangleright \Theta; \Phi_3 \end{array}}{\mathbf{let}\ (x, y) = V\ \mathbf{in}\ M \Leftarrow \sigma \blacktriangleright \Theta; \Phi_1 \cup \Phi_2 \cup \Phi_3}$$

Pair construction (TC-Pair) checks that both components have the given types. 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 of the pair 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.

Sum types. Sum types are similar to product types; again, the declarative rules are unremarkable except for the requirement that sum components must be returnable in the elimination rule.

$$\frac{\Gamma \vdash V : A}{\Gamma \vdash \textbf{inl } V : A + B}\ \textsc{T-Inl} \qquad \frac{\Gamma \vdash V : B}{\Gamma \vdash \textbf{inr } V : A + B}\ \textsc{T-Inr}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash V : A_1 + A_2 \\ \text{returnable}(A) \qquad \text{returnable}(B) \qquad \Gamma_2, x : A_1 \vdash M : B \qquad \Gamma_2, y : A_2 \vdash N : B\end{array}}{\Gamma \vdash \textbf{case } V \textbf{ of } \{\textbf{inl } x : A_1 \mapsto M; \textbf{inr } y : A_2 \mapsto N\} : B}\ \textsc{T-Case}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash V : A_1 + A_2 \\ \Gamma_2, x : A_1 \vdash M : B \qquad \Gamma_2, y : A_2 \vdash N : B\end{array}}{\Gamma \vdash \textbf{case } V \textbf{ of } \{\textbf{inl } x \mapsto M; \textbf{inr } y \mapsto N\} : B}\ \textsc{T-CaseNoAnn}$$

We can also write the corresponding algorithmic rules:

$$\frac{V \Leftarrow \tau \blacktriangleright \Theta;\ \Phi}{\textbf{inl } V \Leftarrow \tau + \sigma \blacktriangleright \Theta;\ \Phi}\ \textsc{TC-Inl} \qquad \frac{V \Leftarrow \sigma \blacktriangleright \Theta;\ \Phi}{\textbf{inr } V \Leftarrow \tau + \sigma \blacktriangleright \Theta;\ \Phi}\ \textsc{TC-Inr}$$

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

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

As expected, sum injections are checking cases; similar to the product rules, we also have two separate rules for case expressions which allow annotations to be elided if both x and y are used within continuations M and N.

C.2 Contextual Type Information

The other main extension supplements co-contextual type checking with contextual type information in order to enable extensions such as higher-order functions and interfaces.

C.2.1 Extended syntax. We begin by showing the modified syntax.

Syntax

Modified types	π, ρ	::=	$C \mid\ !^I\gamma \mid\ ?^I\delta \mid \overrightarrow{\tau} \xrightarrow{\diamond} \sigma$
Interfaces	I	::=	$\cdot \mid I, \mathtt{m} \mapsto \pi$
Linearity annotations	$\diamond$	::=	$\square \mid \blacksquare$
Additional values	V, W	::=	$\cdots \mid \lambda^{\diamond}(\overrightarrow{x : \tau}) : \sigma \,.\, M$
Modified computations	M, N	::=	$\cdots \mid V^{\overrightarrow{\tau} \to \sigma}(W) \mid \textbf{new}[I] \mid \textbf{guard}^I\ V : E\,\{\overrightarrow{G}\}$
		$\mid$	$V\,!^I\,\mathtt{m}[\overrightarrow{W}]$

Like signatures in the core calculus, interfaces I map message names to lists of types. We modify types π, ρ to include *interface-annotated* mailbox types, as well as n-ary function types $\overrightarrow{\tau} \xrightarrow{\diamond} \sigma$ that are annotated as either *linear* ($\square$, meaning that the function closes over linear variables and therefore must be used precisely once) or *unrestricted* ($\blacksquare$, meaning that the function only closes over unrestricted variables and therefore can be used an unlimited number of times).

First-class functions. We extend values with fully-annotated, n-ary anonymous functions. We require a return annotation since we wish to check the body type, while also synthesising a return type. We opt for an n-ary function rather than a curried representation because anonymous functions may only close over returnable values, to ensure they do not violate the conditions on lexical scoping once applied.

Example C.1. Consider the following expression:

$$\begin{array}{l}
\textbf{let}\ mb = \textbf{new in} \\
\textbf{let}\ f = (\lambda^{\square}() : \mathbf{1}\,.\, mb\,!\,\mathtt{m}[\,])\ \textbf{in} \\
\textbf{guard}\ mb : \mathtt{m}\,\{ \\
\quad \textbf{receive}\ \mathtt{m}[\,]\ \textbf{from}\ mb \mapsto \textbf{free}\ mb \\
\} \\
f()
\end{array}$$

Here we bind f to a function which sends message $\mathtt{m}$ 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:

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

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.

Mailbox terms. We extend computations so that a user specifies an interface I when creating a mailbox ($\textbf{new}[I]$). Furthermore, we also augment send and guard expressions with the interface of the mailbox they operate on. Unlike the annotation on **new**, this does not need to be specified by the user, but instead is added by a straightforward type-directed translation.

C.2.2 Type-directed translation. We propagate annotations to function application and mailbox terms via a contextual type-directed translation.

To do so, we introduce *pre-types* P, Q: the main difference is that mailbox types *do not* carry a pattern, but only an interface.

$$\begin{array}{llcl}
\text{Pre-types} & P, Q & ::= & C \mid \mathsf{Mailbox}(I) \mid \overrightarrow{P} \xrightarrow{\diamond} Q \\
\text{Pre-type environments} & \Omega & ::= & \cdot \mid \Omega, x : P
\end{array}$$

The type-directed translation pass follows the form of a standard type system for the simply-typed λ calculus so we omit the rules here. However the judgement has the form $\Omega \vdash M : P \rightsquigarrow N$ which can be read "under pre-type environment Ω, term M has pre-type P and produces annotated term N".

C.2.3 Constraint generation rules. Finally, we can see how to write constraint generation rules for the extended calculus. The rules in the declarative setting are similar.

Modified constraint generation rules

$\boxed{M \Rightarrow \tau \blacktriangleright \Theta;\ \Phi}$ $\boxed{M \Leftarrow \tau \blacktriangleright \Theta;\ \Phi}$ $\boxed{\{E; I\}\ G \Leftarrow \tau \blacktriangleright \Psi;\ \Phi;\ F}$

TS-LinLam

$$\dfrac{\begin{array}{cc} M \Leftarrow \sigma \blacktriangleright \Theta';\ \Phi_1 & \Theta = \Theta' - \vec{x} \\ \mathsf{check}(\vec{x}, \vec{\tau}, \Theta') = \Phi_2 & \mathsf{returnable}(\Theta) \end{array}}{\lambda^{\square}(\overrightarrow{x : \tau}) : \sigma\, .\, M \Rightarrow \vec{\tau} \xrightarrow{\square} \sigma \blacktriangleright \Theta;\ \Phi_1 \cup \Phi_2}$$

TS-UnLam

$$\dfrac{\begin{array}{c} M \Leftarrow \sigma \blacktriangleright \Theta';\ \Phi_1 \qquad \Theta = \Theta' - \vec{x} \\ \mathsf{check}(\vec{x}, \vec{\tau}, \Theta') = \Phi_2 \qquad \mathsf{unr}(\Theta) \blacktriangleright \Phi_3 \qquad \mathsf{returnable}(\Theta) \end{array}}{\lambda^{\blacksquare}(\overrightarrow{x : \tau}) : \sigma\, .\, M \Rightarrow \vec{\tau} \xrightarrow{\blacksquare} \sigma \blacktriangleright \Theta;\ \Phi_1 \cup \Phi_2 \cup \Phi_3}$$

TS-FnApp

$$\dfrac{\begin{array}{c} V \Leftarrow \vec{\tau} \xrightarrow{\diamond} \sigma \blacktriangleright \Theta';\ \Phi \\ (W_i \Leftarrow \tau_i \blacktriangleright \Theta_i;\ \Phi_i)_{i \in 1..n} \qquad \Theta' + \Theta_1 + \ldots + \Theta_n \blacktriangleright \Theta; \Phi \end{array}}{V^{\vec{\tau} \xrightarrow{\diamond} \sigma}(\vec{W}) \Rightarrow \sigma \blacktriangleright \Theta;\ \Phi \cup \Phi_1 \cup \ldots \cup \Phi_n}$$

TS-Send

$$\dfrac{\begin{array}{c} I(\mathtt{m}) = \vec{\pi} \qquad V \Leftarrow\ !^I \mathtt{m}^{\circ} \blacktriangleright \Theta';\ \Phi \\ (W_i \Leftarrow \lceil \pi_i \rceil \blacktriangleright \Theta_i;\ \Phi'_i)_i \qquad \Theta' + \Theta_1 + \cdots + \Theta_n \blacktriangleright \Theta; \Phi'' \end{array}}{V\,!^I\,\mathtt{m}[\vec{W}] \Rightarrow \mathbf{1} \blacktriangleright \Theta;\ \Phi \cup \Phi'_1 \cup \cdots \cup \Phi'_n \cup \Phi''}$$

TS-New

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

TC-Guard

$$\dfrac{\begin{array}{c} \{E; I\}\ \vec{G} \Leftarrow \tau \blacktriangleright \Psi;\ \Phi_1;\ F \\ V \Leftarrow ?^I F^{\bullet} \blacktriangleright \Theta';\ \Phi_2 \qquad \Psi + \Theta' \blacktriangleright \Theta; \Phi_3 \end{array}}{\mathbf{guard}^I\ V : E\,\{\vec{G}\} \Leftarrow \tau \blacktriangleright \Theta;\ \Phi_1 \cup \Phi_2 \cup \Phi_3 \cup \{E <: F\}}$$

TCG-Recv

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

We require three new rules for first-class functions: 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 types an annotated function application, checking that the function has the given annotation and that the arguments 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.

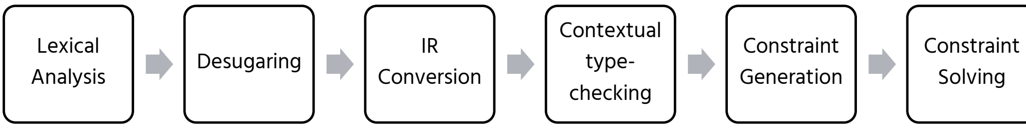


Fig. 14. Pat type checking pipeline

D SUPPLEMENTARY IMPLEMENTATION AND EVALUATION MATERIAL

Our typechecking tool processes Pat programs specified in plain text files that are structured in three segments:

(1) **Interface definitions** that establish the set of messages that a mailbox can receive;
(2) **Function definitions** that specify processes that are instantiable using **`spawn`**;
(3) **Body** an invocation of a function defined in (2) that acts as the program entry point.

Pat programs are type checked following the six-stage pipeline of Fig. 14; see §6.1 for an overview.

D.1 Experimental Conditions

We report the mean typechecking time, excluding phases 1–3 of the pipeline. Measurements are made on a MacBook M1 Pro with 16GB of memory, running macOS 13.2 and OCaml 5.0. To ensure minimal variability in our measurements show in Tbl. 1, we preformed 1000 repetition of each experiment. The number of repetitions was determined empirically by calculating the coefficient of variation (CV) [13], *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.

D.2 Case Study

We give the full Pat program for our factory case study described in §6.2.2. The interaction sequence between our different entities, Robots, Door, and Warehouse, is captured in Fig. 13 and naturally translates to the code that follows.

Interfaces. The messages that Robot, Door, and Warehouse accept are defined by the interfaces:

```
interface Robot {
  GoIn(Door!), GoOut(Door!), Busy(), Delivered(Warehouse!, Door!)
}

interface Door {
  Want(Int, Robot!), Inside(Robot!), Outside(), WantLeave(Robot!), Prepared(Warehouse!), TableIdle()
}

interface Warehouse {
  Prepare(Int, Door!), Deliver(Robot!, Door!), PartTaken()
}
```

Robot. Robots are initially `idle` and issue a `Want` message to the Door to obtain Warehouse access (line 13). The Door replies either with the message `Busy`, in which case the Robot terminates (lines 15–16), or `GoIn`, to which the Robot replies by an `Inside` message before transitioning to the `working` state (lines 17–19). When in `working` state, the Robot expects one `Delivered` message, as the guard on line 23 asserts. This informs the Robot that the part is delivered by the Warehouse. The recipient Robot replies by sending `PartTaken`, and notifies the Door that it wants to exit via `WantLeave` on lines 25–26. It then awaits `GoOut` and finalises its negotiation with the Door through an `Outside` message.

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

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

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 transitioning to the `busy` state (lines 36–39). Both `GoIn` and `Prepare` include an updated self-reference to ensure precise types. The clause **`free`** on line 35 handles the case where no Robots are present. 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 44). These `Want` messages are answered with `Busy`, as lines 45–47 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 line 48–51). Eventually, the Robot demands exit by sending `WantLeave` to the Door, which handles it on lines line 53–54. 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 64–73). Finally, the Door transitions back to `clear` on line 74, ready to service other Robots.

```
33 def clear(self: Door?, wh: Warehouse!): Unit {
34   guard self: *Want {
35     free → ()
36     receive Want(part, robot) from self →
37       robot ! GoIn(self);
38       wh ! Prepare(part, self);
39       busy(self)
40   }
41 }
42
43 def busy(self: Door?): Unit {
44   guard self: Inside.Prepared.*Want {
45     receive Want(partNum, robot) from self →
46       robot ! Busy();
47       busy(self)
48     receive Inside(robot) from self →
49       guard self: Prepared.*Want {
50         receive Prepared(wh) from self →
51           wh ! Deliver(robot, self);
52           guard self: WantLeave.TableIdle.*Want {
53             receive WantLeave(robot) from self →
54               robot ! GoOut(self);
55               ready(self, wh)
56           }
57       }
58   }
59 }
```

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

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 `Prepared + 1` on line 78. 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 86–92). The Robot acknowledges the delivery by sending `PartTaken`, as the guard on line 94 stipulates. To conclude its interaction with the Door, the Warehouse sends `TableIdle` before transitioning back to `empty`.

```
77 def empty(self: wh?): Unit {
78   guard self: Prepare + 1 {
79     free → ()
80     receive Prepare(partNum, door) from self →
81       door ! Prepared(self);
82       engaged(self)
83   }
84 }
85
86 def engaged(self: wh?): Unit {
87   guard self: Deliver {
88     receive Deliver(robot, door) from self →
89       robot ! Delivered(self, door);
90       given(self, door)
91   }
92 }
93 def given(self: wh?, door: Door!): Unit {
94   guard self : PartTaken {
95     receive PartTaken() from self →
96       door ! TableIdle(self);
97       empty(self)
98   }
99 }
100
101 def main(): Unit { # Launcher function.
102   let robot_i = new[Robot] in # n Robot mailboxes.
103   let door = new[Door] in
104   let wh = new[Warehouse] in
105   spawn { clear(door, wh) }; # Door.
106   spawn { idle(robot_i, door) }; # n Robots.
107   spawn { empty(wh) } # Warehouse.
108 }
```

The function `main()` creates n `Robot` mailboxes, together with a `Door` and `Warehouse` mailbox, spawning the respective processes on lines 105–107.

E PROOFS

E.1 Preservation

E.1.1 Auxiliary Definitions and Lemmas. We extend $\mathsf{returnable}(-)$ to typing environments, writing $\mathsf{returnable}(\Gamma)$ if $\mathsf{returnable}(A)$ for each $x : A \in \Gamma$. Similarly, we write $\mathsf{irrelevant}(A)$ if A is irrelevant (i.e., it is a mailbox type $!E^{\eta} \leq !\mathbb{1}^{\eta}$), and extend this to environments.

We write $\mathsf{fv}(M)$ to return the free variables of a term.

Environment subtyping includes a notion of weakening. Read top-down, environment subtyping rules allow us to add a variable with mailbox type $!\mathbb{1}$, replace a type with its subtype, and add in base types. It is therefore useful to introduce a definition referring to the class of types which can be added in through T-Subs which may not be used by a term. We call these types *cruft*. Cruft is a refinement of $\mathsf{un}(-)$ since it also encompasses types which are *subtypes* of $!\mathbb{1}$.

Definition E.1 (Cruft). A type T is *cruft*, written $\mathsf{cruft}(T)$, if either T is a base type, or $\mathsf{irrelevant}(T)$. A usage-aware type T^{η} is cruft if $\eta = \circ$ and T is cruft.

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

Definition E.2 (Strict environment subtyping). An environment Γ is a *strict subtype environment* of an environment Γ', written $\Gamma \preccurlyeq \Gamma'$ if $\Gamma \leq \Gamma'$ and $\mathsf{dom}(\Gamma) = \mathsf{dom}(\Gamma')$.

We extend $\mathsf{cruft}(-)$ to type environments and usage-aware type environments in the usual way.

Definition E.3 (Cruftless). We say that an environment is *cruftless for a term M* if $\Gamma \vdash M : A$ and $\mathsf{dom}(\Gamma) = \mathsf{fv}(M)$.

Let us also use Π to range over type environments. A crucial lemma for taming the complexity of environment subtyping is the following, which allows us to separate the type environment required for typing the term from the cruft introduced by environment subtyping.

Lemma E.4. *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* $\Pi_1 \preccurlyeq \Pi_3$
- $A' \leq A$
- *cruft*(Π_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. □

The substitution lemma is only defined on disjoint environments: we should not be substituting a name into a term where it is already free. This is ensured by distinguishing between returnable and second-class usages of a variable: if a variable is returnable, then we know it cannot be used within the term into which it is being substituted. If a variable is second-class, then there will be no applicable reduction rules which result in substitution.

Lemma E.5 (Substitution). *If:*

- $\Gamma_1, x : A \vdash M : B$
- $\Gamma_2 \vdash V : A'$
- $A' \leq A$
- $\Gamma_1 + \Gamma_2$ *is defined*

then $\Gamma_1 + \Gamma_2 \vdash M\{V/x\} : B$.

Proof. By induction on the derivation of $\Gamma_1, x : A \vdash M : B$. □

Lemma E.6 (Subtyping preserves reliablility / usability [12]). *If $A \leq B$, then:*

(1) A reliable implies B reliable
(2) B usable implies A usable

Corollary E.7. *If $\Gamma_1 \leq \Gamma_2$ then:*

(1) Γ_1 reliable implies Γ_2 reliable
(2) Γ_2 usable implies Γ_2 usable

Lemma E.8 (Balancing [12]). *If $\mathtt{m} \odot F \sqsubseteq E$ where $F \not\sqsubseteq \mathbb{0}$ and $E \,/\, \mathtt{m}$ is defined, then $F \sqsubseteq E \,/\, \mathtt{m}$.*

Lemma E.9. *If $A \leq B$ and returnable(B), then returnable(A)*

Proof. Follows from the fact that $\bullet \leq \circ$. □

Corollary E.10. *If $\Gamma_1 \preccurlyeq \Gamma_2$ and returnable(Γ_2), then returnable(Γ_1).*

Lemma E.11. *If $\Gamma \vdash V : A$ where returnable(A) and Γ is cruftless for V, then returnable(Γ).*

Proof. By case analysis on the derivation of $\Gamma \vdash V : A$. □

Lemma E.12. *If $(\Pi_1, \Pi_2) \triangleright \Gamma$ is defined and returnable(Π_1), then $\Pi_1, \Pi_2 \triangleright \Gamma = \Pi_1 + (\Pi_2 \triangleright \Gamma)$.*

Proof. Follows from the definition of usage combination: the operation is not symmetric 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. □

Lemma E.13. *If $\Gamma_1 \triangleright \Gamma_2$ is defined, with Γ_1 and Γ_2 sharing only variables of base type, then $\Gamma_1 + \Gamma_2$ is defined.*

Proof. Immediate from the definitions. □

Lemma E.14 ($\triangleright$ is associative). $$A_1 \triangleright (A_2 \triangleright A_3) \iff (A_1 \triangleright A_2) \triangleright A_3$$

Proof. Follows from the fact that usage combination is associative, and that we identify patterns up to commutativity and associativity. □

Extending to usage-aware type environments, we get the following corollary:

Corollary E.15. $$\Gamma_1 \triangleright (\Gamma_2 \triangleright \Gamma_3) \iff (\Gamma_1 \triangleright \Gamma_2) \triangleright \Gamma_3$$

The same result holds for runtime type environments and $\bowtie$:

Lemma E.16 ($\bowtie$ is associative). $$\Delta_1 \bowtie (\Delta_2 \bowtie \Delta_3) \iff (\Delta_1 \bowtie \Delta_2) \bowtie \Delta_3$$

Proof. Follows the same reasoning as for $\triangleright$. □

Lemma E.17. *The $\bowtie$ operator is commutative: $\Delta_1 \bowtie \Delta_2 = \Delta_2 \bowtie \Delta_1$.*

Proof. Follows from the fact that $\boxplus$ is commutative. □

Lemma E.18. $\Gamma_1 + (\Gamma_2 \triangleright \Gamma_3) = (\Gamma_1 + \Gamma_2) \triangleright \Gamma_3$.

Proof. Follows directly from the definitions. □

Lemma E.19. *If $\Gamma_1, \Gamma_2 = \Gamma$, then $\Gamma_1 \triangleright \Gamma_2 = \Gamma$*

Proof. Follows from the definition of $\triangleright$ given that Γ_1 and Γ_2 are disjoint. □

Lemma E.20. *If* $\Gamma_1 \triangleright \Gamma_2 = \Gamma$*, then* $|\Gamma_1| \bowtie |\Gamma_2| = |\Gamma|$.

Proof. Follows directly from the definitions, since $\bowtie$ is more liberal than $\triangleright$. ◻

Lemma E.21. $|\Gamma| \bowtie \Delta = |\lceil \Delta \rceil \triangleright \Gamma|$.

Proof. For each x such that $x : T \in |\Gamma|$ and $x : U \in \Delta$, since $|\Gamma| \bowtie \Delta$ is defined, we have that $T \boxplus U$ is defined. The result then follows from the definition of $\triangleright$, noting that all types in $\lceil \Delta \rceil$ are usable and therefore combinable with any other usage. ◻

Lemma E.22. *If* $\Gamma_1 \triangleright \Gamma_2 = \Gamma$ *and* $\mathit{irrelevant}(\Gamma_1)$*, then* $\Gamma_2 \simeq \Gamma$.

Proof. Since $\mathrm{irrelevant}(\Gamma_1)$, we have that for each $x : A$, it is the case that $A = \,!E^{\circ}$ where $E \sqsubseteq \mathbb{1}$. ◻

E.1.2 Preservation proof.

Lemma E.23 (Preservation (Equivalence)). *If* $\Gamma \vdash \mathcal{C}$ *and* $\mathcal{C} \equiv \mathcal{D}$*, then* $\Gamma \vdash \mathcal{D}$.

Proof. By induction on the derivation of $\mathcal{C} \equiv \mathcal{D}$, relying on Lemmas E.16 and E.17 and TC-Subs. ◻

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

Proof. By induction on the derivation of $\Gamma \vdash \mathcal{C}$.

Case E-Let

Assumption:

$$\dfrac{\dfrac{\Gamma_1 \triangleright \Gamma_2 = \lfloor \Delta' \rfloor \qquad \dfrac{\Gamma_3 \triangleright \Gamma_4 = \Gamma_1 \qquad \Gamma_3 \vdash M : \lfloor T \rfloor \qquad \Gamma_4, x : \lfloor T \rfloor \vdash N : B}{\Gamma_1 \vdash \mathbf{let}\ x \colon T = M\ \mathbf{in}\ N : B} \qquad \Gamma_2 \vdash B \blacktriangleright \Sigma}{\Delta' \vdash (\!| \mathbf{let}\ x \colon T = M\ \mathbf{in}\ N, \Sigma |\!)}}{\Delta \vdash (\!| \mathbf{let}\ x \colon T = M\ \mathbf{in}\ N, \Sigma |\!)}$$

where

- $\Delta \leq \Delta'$
- $\lfloor \Delta' \rfloor = \Gamma_1 \triangleright \Gamma_2$
- $\Gamma_1 = \Gamma_3 \triangleright \Gamma_4$

so, $\lfloor \Delta' \rfloor = (\Gamma_3 \triangleright \Gamma_4) \triangleright \Gamma_2$.
By Lemma E.14, $\lfloor \Delta' \rfloor = \Gamma_3 \triangleright (\Gamma_4 \triangleright \Gamma_2)$.
Recomposing:

$$\dfrac{\dfrac{\Gamma_3 \triangleright (\Gamma_4 \triangleright \Gamma_2) = \lfloor \Delta' \rfloor \qquad \Gamma_3 \vdash M : \lfloor T \rfloor \qquad \dfrac{\Gamma_4, x : \lfloor T \rfloor \vdash N : B \qquad \Gamma_2 \vdash B \blacktriangleright \Sigma}{(\Gamma_4 \triangleright \Gamma_2) \vdash \lfloor T \rfloor \blacktriangleright \langle x, N \rangle \cdot \Sigma}}{\Delta' \vdash (\!| M, \langle x, N \rangle \cdot \Sigma |\!)}}{\Delta \vdash (\!| M, \langle x, N \rangle \cdot \Sigma |\!)}$$

Case E-Return

Assumption:

$$\dfrac{\Gamma_1 \triangleright \Gamma_2 = \lfloor \Delta \rfloor \qquad \Gamma_1 \vdash V : A \qquad \mathsf{returnable}(A) \qquad \dfrac{\Gamma_2 = \Gamma_3 \triangleright \Gamma_4 \quad \Gamma_3, x : A \vdash M : B \qquad \Gamma_4 \vdash B \blacktriangleright \Sigma}{\Gamma_2 \vdash A \blacktriangleright \langle x, M \rangle \cdot \Sigma}}{\Delta \vdash (\!| V, \langle x, M \rangle \cdot \Sigma |\!)}$$

Environments:

- $\lfloor \Delta \rfloor = \Gamma_1 \triangleright \Gamma_2$
- $\Gamma_2 = \Gamma_3 \triangleright \Gamma_4$

so, $\lfloor \Delta \rfloor = \Gamma_1 \triangleright (\Gamma_3 \triangleright \Gamma_4)$.

By Lemma E.4, we have that there exist Π_1, Π_2, Π_3 such that:

- $\Gamma_1 = \Pi_1, \Pi_2$
- $\Pi_3 \vdash V : A'$
- Π_1 is cruftless for V, and $\Pi_1 \preccurlyeq \Pi_3$
- $A' \leq A$
- $\mathsf{cruft}(\Pi_2)$

By Lemma E.9, $\mathsf{returnable}(A')$, and by Lemma E.11, $\mathsf{returnable}(\Pi_3)$.
By Corollary E.10, $\mathsf{returnable}(\Pi_1)$.
By Lemma E.5, $\Pi_1 + \Gamma_3 \vdash M\{V/x\} : B$.
Equational reasoning on environments:

$$\begin{array}{l} \lfloor \Delta \rfloor \\ \quad = (\text{expanding}) \\ \Gamma_1 \triangleright \Gamma_2 \\ \quad = (\text{expanding}) \\ \Gamma_1 \triangleright (\Gamma_3 \triangleright \Gamma_4) \\ \quad = (\text{expanding}) \\ \Pi_1, \Pi_2 \triangleright (\Gamma_3 \triangleright \Gamma_4) \\ \quad = (\text{Lemma E.14}) \\ (\Pi_1, \Pi_2 \triangleright \Gamma_3) \triangleright \Gamma_4 \\ \quad = (\text{Lemma E.12}) \\ (\Pi_1 + (\Pi_2 \triangleright \Gamma_3)) \triangleright \Gamma_4 \end{array}$$

Let $\Delta' = |(\Pi_1 + (\Pi_2 \triangleright \Gamma_3)) \triangleright \Gamma_4|$.
By the definitions of environment subtyping, follows that $\Pi_1 + (\Pi_2 \triangleright \Gamma_3) \leq \Pi_1 + \Gamma_3$.
Therefore:

$$\dfrac{\dfrac{(\Pi_1 + (\Pi_2 \triangleright \Gamma_3)) \triangleright \Gamma_4 = \lfloor \Delta' \rfloor \qquad \dfrac{\Pi_1 + \Gamma_3 \vdash M\{V/x\} : B}{\Pi_1 + (\Pi_2 \triangleright \Gamma_3) \vdash M\{V/x\} : B} \qquad \Gamma_4 \vdash B \blacktriangleright \Sigma}{\Delta' \vdash (\!| M\{V/x\}, \Sigma |\!)}}{\Delta \vdash (\!| M\{V/x\}, \Sigma |\!)}$$

as required.

Case E-App

Assumption:

$$\frac{\lfloor \Delta \rfloor = \Gamma_1 \triangleright \Gamma_2 \qquad \dfrac{\dfrac{\mathcal{P}(f) = \overrightarrow{A} \longrightarrow B \qquad (\Gamma'_i \vdash V_i : A_i)_i}{\Gamma'_1 + \cdots + \Gamma'_n \vdash f(\overrightarrow{V}) : B}}{\Gamma_1 \vdash f(\overrightarrow{V}) : B} \qquad \Gamma_2 \vdash B \blacktriangleright \Sigma}{\Delta \vdash (\!| f(\overrightarrow{V}), \Sigma |\!)}$$

Since we also assume $\vdash \mathcal{P}$, we know by definition typing that:

$$\frac{\overrightarrow{x : A} \vdash_{\mathcal{P}} M : B}{\vdash_{\mathcal{P}} \mathbf{def}\ f(\overrightarrow{x : A}) \colon B\ \{M\}}$$

By Lemma E.5 we have that $\Gamma'_1 + \cdots + \Gamma'_n \vdash M\{\overrightarrow{V}/\overrightarrow{x}\} : B$.
Thus we can recompose:

$$\frac{\lfloor \Delta \rfloor = \Gamma_1 \triangleright \Gamma_2 \qquad \dfrac{\Gamma'_1 + \cdots + \Gamma'_n \vdash M\{\overrightarrow{V}/\overrightarrow{x}\} : B}{\Gamma_1 \vdash M\{\overrightarrow{V}/\overrightarrow{x}\} : B} \qquad \Gamma_2 \vdash B \blacktriangleright \Sigma}{\Delta \vdash (\!| M\{\overrightarrow{V}/\overrightarrow{x}\}, \Sigma |\!)}$$

as required.

Case E-New

Assumption:

$$\cfrac{\cfrac{\Gamma_1 \triangleright \Gamma_2 = \lfloor \Delta' \rfloor \qquad \dfrac{\overline{\cdot \vdash \mathbf{new} : ?\mathbb{1}^\bullet}}{\Gamma_1 \vdash \mathbf{new} : ?\mathbb{1}^\bullet} \qquad \Gamma_2 \vdash ?\mathbb{1}^\bullet \blacktriangleright \Sigma}{\Delta' \vdash (\!| \mathbf{new}, \Sigma |\!)}}{\Delta \vdash (\!| \mathbf{new}, \Sigma |\!)}$$

where $\Delta \leq \Delta'$.

$$\cfrac{\cfrac{\cfrac{\Gamma_1, a : ?\mathbb{1}^\bullet \triangleright \Gamma_2 = \lfloor \Delta', a : ?\mathbb{1} \rfloor \qquad \dfrac{\overline{a : ?\mathbb{1}^\bullet \vdash a : ?\mathbb{1}^\bullet}}{\Gamma_1, a : ?\mathbb{1} \vdash a : ?\mathbb{1}^\bullet} \qquad \Gamma_2 \vdash ?\mathbb{1}^\bullet \blacktriangleright \Sigma}{\Delta', a : ?\mathbb{1} \vdash (\!| a, \Sigma |\!)}}{\Delta' \vdash (\nu a)((\!| a, \Sigma |\!))}}{\Delta \vdash (\nu a)((\!| a, \Sigma |\!))}$$

as required.

Case E-Send

$$\dfrac{\dfrac{\Gamma_1 \triangleright \Gamma_2 = \lfloor \Delta \rfloor \qquad \Gamma_1 \vdash a\,!\,\mathtt{m}[\,\overrightarrow{V}\,] : \mathbf{1} \qquad \Gamma_2 \vdash \mathbf{1} \blacktriangleright \Sigma}{\Delta' \vdash (\!|\, a\,!\,\mathtt{m}[\,\overrightarrow{V}\,], \Sigma \,|\!)}}{\Delta \vdash (\!|\, a\,!\,\mathtt{m}[\,\overrightarrow{V}\,], \Sigma \,|\!)}$$

By Lemma E.4 we have that:

- $\Gamma_1 = \Pi_1, \Pi_2$
- $\Pi_3 \vdash a\,!\,\mathtt{m}[\,\overrightarrow{V}\,] : \mathbf{1}$
- Π_1 is cruftless for $a\,!\,\mathtt{m}[\,\overrightarrow{V}\,]$ and $\Pi_1 \preccurlyeq \Pi_3$
- $\mathrm{cruft}(\Pi_2)$

Therefore we have that $\Pi_3 = \Pi_3', a : \,!\mathtt{m}$ such that:

$$\dfrac{a : \,!\mathtt{m}^{\circ} \vdash a : !\mathtt{m} \qquad \Pi_3' \vdash \overrightarrow{V} : \overrightarrow{A} \qquad \overrightarrow{A} \leq \mathcal{P}(\mathtt{m})}{\Pi_3', a : \,!\mathtt{m}^{\circ} \vdash a\,!\,\mathtt{m}[\,\overrightarrow{V}\,] : \mathbf{1}}$$

$$\begin{array}{l}
\Delta' \\
\quad = \text{(expanding)} \\
|\Gamma_1 \triangleright \Gamma_2| \\
\quad = \text{(expanding)} \\
|(\Pi_1, \Pi_2) \triangleright \Gamma_2| \\
\quad = \text{(expanding)} \\
|(\Pi_1', a : \,!\mathtt{m}^{\circ}, \Pi_2) \triangleright \Gamma_2| \\
\quad = \text{(Lemma E.19)} \\
|(\Pi_1', a : \,!\mathtt{m}^{\circ} \triangleright \Pi_2) \triangleright \Gamma_2| \\
\quad = \text{(Lemma E.15)} \\
|\Pi_1', a : \,!\mathtt{m}^{\circ} \triangleright (\Pi_2 \triangleright \Gamma_2)| \\
\quad = \text{(Lemma E.20)} \\
|\Pi_1', a : \,!\mathtt{m}^{\circ}| \bowtie |(\Pi_2 \triangleright \Gamma_2)| \\
\quad = (\bowtie \text{ is commutative}) \\
|\Pi_2 \triangleright \Gamma_2| \bowtie |\Pi_1', a : \,!\mathtt{m}^{\circ}|
\end{array}$$

Recomposing:

$$\dfrac{\dfrac{\dfrac{\Pi_2 \triangleright \Gamma_2 = \lfloor |\Pi_2 \triangleright \Gamma_2| \rfloor \qquad \Pi_2 \vdash () : \mathbf{1} \qquad \Gamma_2 \vdash \mathbf{1} \blacktriangleright \Sigma}{|\Pi_2 \triangleright \Gamma_2| \vdash (\!|\, (), \Sigma \,|\!)} \qquad \dfrac{a : \,!\mathtt{m} \vdash a : !\mathtt{m} \qquad \lceil \Pi_1' \rceil \vdash \overrightarrow{V} : \overrightarrow{A} \qquad \overrightarrow{A} \leq \lceil \mathcal{P}(\mathtt{m}) \rceil}{|\Pi_1', a : \,!\mathtt{m}| \vdash a \leftarrow \mathtt{m}[\,\overrightarrow{V}\,]}}{|\Pi_2 \triangleright \Gamma_2| \bowtie |\Pi_1', a : \,!\mathtt{m}| \vdash (\!|\, (), \Sigma \,|\!) \parallel a \leftarrow \mathtt{m}[\,\overrightarrow{V}\,]}}{\Delta \vdash (\!|\, (), \Sigma \,|\!) \parallel a \leftarrow \mathtt{m}[\,\overrightarrow{V}\,]}$$

as required.

Case E-Spawn

Assumption:

$$\dfrac{\dfrac{\lceil\Gamma_1\rceil \triangleright \Gamma_2 = \lfloor\Delta'\rfloor \qquad \dfrac{\dfrac{\Gamma_1' \vdash M : \mathbf{1}}{\Gamma_1 \vdash M : \mathbf{1}}}{\lceil\Gamma_1\rceil \vdash \mathbf{spawn}\ M : \mathbf{1}} \qquad \Gamma_2 \vdash \mathbf{1} \blacktriangleright \Sigma}{\Delta' \vdash (\!| \mathbf{spawn}\ M, \Sigma |\!)}}{\Delta \vdash (\!| \mathbf{spawn}\ M, \Sigma |\!)}$$

where $\Gamma_1 \leq \Gamma_1'$.
By Lemma E.4, there exist Π_1, Π_2, Π_3 such that:

- $\Gamma_1 = \Pi_1, \Pi_2$
- $\Pi_3 \vdash M : \mathbf{1}$
- Π_1 is cruftless for M and $\Pi_1 \preccurlyeq \Pi_3$
- $\mathrm{cruft}(\Pi_2)$

We now prove that $\Delta' \leq |\Pi_2 \triangleright \Gamma_2| \bowtie |\Pi_1|$.

$$\begin{array}{l}
\Delta' \\
\quad = \text{(expanding)} \\
|\lceil\Gamma_1\rceil \triangleright \Gamma_2| \\
\quad = \text{(expanding)} \\
|\lceil\Pi_1, \Pi_2\rceil \triangleright \Gamma_2| \\
\quad = (\Pi_2 \text{ cruft, so usable}) \\
|(\lceil\Pi_1\rceil, \Pi_2) \triangleright \Gamma_2| \\
\quad = \text{(Lemma E.19)} \\
|(\lceil\Pi_1\rceil \triangleright \Pi_2) \triangleright \Gamma_2| \\
\quad = \text{(Lemma E.15)} \\
|\lceil\Pi_1\rceil \triangleright (\Pi_2 \triangleright \Gamma_2)| \\
\quad = \text{(Lemma E.20)} \\
|\lceil\Pi_1\rceil| \bowtie |(\Pi_2 \triangleright \Gamma_2)| \\
\quad = (\bowtie \text{ is commutative}) \\
|(\Pi_2 \triangleright \Gamma_2)| \bowtie |\lceil\Pi_1\rceil| \\
\quad = (|{-}| \text{ cancels } \lceil{-}\rceil) \\
|(\Pi_2 \triangleright \Gamma_2)| \bowtie |\Pi_1|
\end{array}$$

It follows that since $\bullet \leq \circ$ and $\Pi_1 \preccurlyeq \Pi_3$, that $\lfloor\Pi_1\rfloor \leq \Pi_3$.
From that, we can construct the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Pi_2 \triangleright \Gamma_2 = \lfloor|\Pi_2 \triangleright \Gamma_2|\rfloor \qquad \dfrac{\cdot \vdash () : \mathbf{1}}{\Pi_2 \vdash () : \mathbf{1}} \qquad \Gamma_2 \vdash \mathbf{1} \blacktriangleright \Sigma}{|\Pi_2 \triangleright \Gamma_2| \vdash (\!| (), \Sigma |\!)} \qquad \dfrac{\lfloor|\Pi_1|\rfloor \triangleright \cdot = \lfloor|\Pi_1|\rfloor \qquad \dfrac{\Pi_3 \vdash M : \mathbf{1}}{\lfloor\Pi_1\rfloor \vdash M : \mathbf{1}} \qquad \dfrac{}{\cdot \vdash \mathbf{1} \blacktriangleright \epsilon}}{|\Pi_1| \vdash (\!| M, \epsilon |\!)}}{|\Pi_2 \triangleright \Gamma_2| \bowtie |\Pi_1| \vdash (\!| (), \Sigma |\!) \parallel (\!| M, \epsilon |\!)}}{\Delta' \vdash (\!| (), \Sigma |\!) \parallel (\!| M, \epsilon |\!)}}{\Delta \vdash (\!| (), \Sigma |\!) \parallel (\!| M, \epsilon |\!)}$$

as required.

Case E-Free

Assumption (assuming WLOG that the **free** guard is the first guard in the sequence):

$$\dfrac{\dfrac{\dfrac{\Gamma_1, a : ?F^{\bullet}_{env} \triangleright \Gamma_2 = \lfloor \Delta', a : ?F^{\bullet}_{env} \rfloor \qquad \Gamma_1, a : ?F^{\bullet}_{env} \vdash \mathbf{guard}\; a : E\, \{\mathbf{free} \mapsto M \cdot \overrightarrow{G}\} : A \qquad \Gamma_2 \vdash A \blacktriangleright \Sigma}{\Delta', a : ?F_{env} \vdash \llparenthesis \mathbf{guard}\; a : E\, \{\mathbf{free} \mapsto M \cdot \overrightarrow{G}\}, \Sigma \rrparenthesis}}{\Delta, a : ?\mathbb{1} \vdash \llparenthesis \mathbf{guard}\; a : E\, \{\mathbf{free} \mapsto M \cdot \overrightarrow{G}\}, \Sigma \rrparenthesis}}{\Delta \vdash (\nu a)(\llparenthesis \mathbf{guard}\; a : E\, \{\mathbf{free} \mapsto M \cdot \overrightarrow{G}\}, \Sigma \rrparenthesis)}$$

where $\Delta \leq \Delta'$, and $?\mathbb{1} \leq ?F$, and (by Lemma E.12) $a \notin \mathrm{dom}(\Gamma_2)$.
Furthermore:

$$\dfrac{\dfrac{\Pi_1, a : ?F'^{\bullet}_{env} \vdash a : ?(\mathbb{1} \oplus F') \qquad \dfrac{\dfrac{\Pi_2 \vdash M : A}{\Pi_2 \vdash \mathbf{free} \mapsto M : A :: \mathbb{1}} \qquad \Pi_2 \vdash M : A :: F'}{\Pi_2 \vdash \mathbf{free} \mapsto M \cdot \overrightarrow{G} : A :: \mathbb{1} \oplus F'}}{\Pi_1, a : ?F'^{\bullet}_{env} + \Pi_2 \vdash \mathbf{guard}\; a : E\, \{\mathbf{free} \mapsto M \cdot \overrightarrow{G}\} : A}}{\Gamma_1, a : ?F^{\bullet}_{env} \vdash \mathbf{guard}\; a : E\, \{\mathbf{free} \mapsto M \cdot \overrightarrow{G}\} : A}$$

where $\mathrm{cruft}(\Pi_1)$. Thus, $\Pi_1 + \Pi_2 \leq \Pi_2$ and $\Pi_1 + \Pi_2 \vdash M : A$. Furthermore, $\Gamma_1 \leq \Pi_1 + \Pi_2$.
Thus, recomposing:

$$\dfrac{\dfrac{\Gamma_1 \triangleright \Gamma_2 = \lfloor \Delta' \rfloor \qquad \dfrac{\Pi_1 + \Pi_2 \vdash M : A}{\Gamma_1 \vdash \mathbf{free} \mapsto M : A} \qquad \Gamma_2 \vdash A \blacktriangleright \Sigma}{\Delta' \vdash \llparenthesis M, \Sigma \rrparenthesis}}{\Delta \vdash \llparenthesis M, \Sigma \rrparenthesis}$$

as required.

Case E-Recv

Assumption:

$$\dfrac{\dfrac{\dfrac{\Gamma_1 \triangleright \Gamma_2 = \lfloor \Delta_1 \rfloor \qquad \mathbf{D} \qquad \Gamma_2 \vdash A \blacktriangleright \Sigma}{\Delta_1 \vdash \llparenthesis \mathbf{guard}\; a : E_{ann}\, \{\mathcal{G}[\mathbf{receive}\; \mathtt{m}[\overrightarrow{x}]\; \mathbf{from}\; y \mapsto M]\}, \Sigma \rrparenthesis} \qquad \dfrac{\lceil \Delta_2 \rceil \vdash \overrightarrow{V} : \overrightarrow{B} \qquad \overrightarrow{U} \leq \lceil \mathcal{P}(\mathtt{m}) \rceil}{\Delta_2, a : !\mathtt{m} \vdash a \leftarrow \mathtt{m}[\overrightarrow{V}]}}{\Delta_1 \bowtie (\Delta_2, a : !\mathtt{m}) \vdash \llparenthesis \mathbf{guard}\; a : E_{ann}\, \{\mathcal{G}[\mathbf{receive}\; \mathtt{m}[\overrightarrow{x}]\; \mathbf{from}\; y \mapsto M]\}, \Sigma \rrparenthesis \parallel a \leftarrow \mathtt{m}[\overrightarrow{V}]}}{\Delta \vdash \llparenthesis \mathbf{guard}\; a : E_{ann}\, \{\mathcal{G}[\mathbf{receive}\; \mathtt{m}[\overrightarrow{x}]\; \mathbf{from}\; y \mapsto M]\}, \Sigma \rrparenthesis \parallel a \leftarrow \mathtt{m}[\overrightarrow{V}]}$$

where **D** is the following derivation:

$$\dfrac{\Gamma_1 = \Gamma_3 + \Gamma_4 \qquad \dfrac{a : ?E^{\bullet} \vdash a : ?E^{\bullet}}{\Gamma_3 \vdash a : ?E^{\bullet}_{ty}} \qquad \Gamma_4 \vdash \mathcal{G}[\mathbf{receive}\; \mathtt{m}[\overrightarrow{x}]\; \mathbf{from}\; y \mapsto M] : A :: E_{ty} \qquad E_{ty} \sqsubseteq E_{ann} \qquad \vDash E_{ty}}{\Gamma_1 \vdash \mathbf{guard}\; a\; \{E_{ann}\} \mathcal{G}[\mathbf{receive}\; \mathtt{m}[\overrightarrow{x}]\; \mathbf{from}\; y \mapsto M] : A}$$

By Lemma E.4, $\Gamma_3 = \Pi, a : ?E^{\bullet}_{env}$, where:

- $?E_{env} \leq ?E$
- $?E \leq ?E_{ty}$

and thus $?E_{env} \leq ?E \leq ?E_{ty}$.

Without loss of generality, let us consider the case where the **receive** is the first guard. We can therefore write $\mathcal{G}[\textbf{receive}\ \texttt{m}[\overrightarrow{x}]\ \textbf{from}\ y \mapsto M]$ as $\textbf{receive}\ \texttt{m}[\overrightarrow{x}]\ \textbf{from}\ y \mapsto M \cdot \overrightarrow{G}$ for some sequence $\overrightarrow{G}$.

By T-GuardSeq and TG-Recv, and since $\vDash E_{ty}$, we have that $E_{ty} = F_1 \oplus \cdots \oplus F_n$, where $F_1 = \texttt{m} \odot F'$ and $F' \simeq E_{ty} \,/\, \texttt{m}$.

Furthermore:

$$\frac{\mathcal{P}(\texttt{m}) = \overrightarrow{U'} \qquad \mathsf{un}(\Gamma_4) \qquad \Gamma_4, \overrightarrow{x} : \overrightarrow{\lceil U' \rceil}, y : ?F'^{\bullet} \vdash M : A}{\Gamma_4 \vdash \textbf{receive}\ \texttt{m}[\overrightarrow{x}]\ \textbf{from}\ y \mapsto M : A :: (\texttt{m} \odot F')}$$

Now, by the definition of $\bowtie$, we know that $E_{env} = \texttt{m} \odot E_{pat}$ for some pattern E_{pat}.

By the definition of $\triangleright$, we also know that $a \notin \mathrm{dom}(\Gamma_4)$.

By Lemma E.8, we have that $E_{pat} \sqsubseteq E \,/\, \texttt{m}$, and thus $?E^{\bullet}_{pat} \leq ?(E \,/\, \texttt{m})^{\bullet}$.

Further, it follows that $?E \,/\, \texttt{m} \leq ?F'$.

By Lemma E.5, $\lceil \Delta_2 \rceil + \Gamma_4, a : ?F'^{\bullet} \vdash M\{\overrightarrow{V}/\overrightarrow{x}, a/y\} : A$.

Equational reasoning:

$$\begin{array}{l}
\Delta_1 \bowtie (\Delta_2, a : !\texttt{m}) \\
\quad = (\text{expanding}) \\
|\Gamma_1 \triangleright \Gamma_2| \bowtie (\Delta_2, a : !\texttt{m}) \\
\quad = (\text{expanding}) \\
|(\Gamma_3 + \Gamma_4) \triangleright \Gamma_2| \bowtie (\Delta_2, a : !\texttt{m}) \\
\quad = (\text{expanding}) \\
|(\Pi, a : ?E^{\bullet}_{env} + \Gamma_4) \triangleright \Gamma_2| \bowtie (\Delta_2, a : !\texttt{m}) \\
\quad = (\text{expanding}) \\
|(\Pi, a : ?(\texttt{m} \odot E_{pat})^{\bullet} + \Gamma_4) \triangleright \Gamma_2| \bowtie (\Delta_2, a : !\texttt{m}) \\
\quad = (\text{Lemma E.12}) \\
|(\Pi + \Gamma_4) \triangleright \Gamma_2, a : ?(\texttt{m} \odot E_{pat})^{\bullet}| \bowtie (\Delta_2, a : !\texttt{m}) \\
\quad = (\text{Definition of } \bowtie) \\
(|(\Pi + \Gamma_4) \triangleright \Gamma_2| \bowtie \Delta_2), a : ?E_{pat} \\
\quad = (\text{Lemma E.21, since } \mathsf{un}(\Gamma_4)) \\
(|\lceil \Delta_2 \rceil \triangleright ((\Pi + \Gamma_4) \triangleright \Gamma_2)|), a : ?E_{pat} \\
\quad = (\text{Lemma E.14}) \\
(|(\lceil \Delta_2 \rceil \triangleright (\Pi + \Gamma_4)) \triangleright \Gamma_2|), a : ?E_{pat} \\
\quad = (\text{Lemma E.13}) \\
(|(\lceil \Delta_2 \rceil + (\Pi + \Gamma_4)) \triangleright \Gamma_2|), a : ?E_{pat} \\
\quad = (a \text{ disjoint from environments}) \\
|(\lceil \Delta_2 \rceil + (\Pi + \Gamma_4), a : ?E^{\bullet}_{pat}) \triangleright \Gamma_2|
\end{array}$$

Let $\Delta' = |(\lceil \Delta_2 \rceil + (\Pi + \Gamma_4), a : ?E^{\bullet}_{pat}) \triangleright \Gamma_2|$

Recomposing:

$$\frac{\dfrac{\lfloor \Delta \rfloor = \lceil \Delta_2 \rceil + (\Pi + \Gamma_4), a : ?E^{\bullet}_{pat} \triangleright \Gamma_2 \qquad \lceil \Delta_2 \rceil + \Gamma_4, a : ?F'^{\bullet} \vdash M\{\overrightarrow{V}/\overrightarrow{x}, a/y\} : A \qquad \Gamma_2 \vdash A \blacktriangleright \Sigma}{\Delta' \vdash (\!| M\{\overrightarrow{V}/\overrightarrow{x}, a/y\}, \Sigma |\!)}}{\Delta \vdash (\!| M\{\overrightarrow{V}/\overrightarrow{x}, a/y\}, \Sigma |\!)}$$

as required.

Case E-Nu

Follows immediately from the induction hypothesis.

Case E-Par

Follows immediately from the induction hypothesis.

Case E-Struct

Follows immediately from Lemma E.23 and the induction hypothesis. □

E.2 Progress

Lemma E.24 (Canonical forms). *If $\cdot \vdash C$ then there exists some $\mathcal{D}$ such that $C \equiv \mathcal{D}$ and $\mathcal{D}$ is in canonical form.*

Proof. Follows from the structural congruence rules. □

Lemma E.25 (Progress (Thread Reduction)). *If $\Gamma \vdash_{\mathcal{P}} (\!| M, \Sigma |\!)$, then either:*

- *M is a value and $\Sigma = \epsilon$; or*
- *there exists 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** *$V : E \{\overrightarrow{G}\}$.*

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

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

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

Then for each M_i, either:

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

Proof. Functional reduction enjoys progress (E.25), and the constructs **new**, **spawn** M, and $a \,!\, \mathtt{m}[\overrightarrow{V}]$ can all always reduce. Therefore, the body of an irreducible thread $(\!| M_k, \Sigma_k |\!)$ must be waiting for a message $\mathtt{m}$ on some name a. To be waiting, name a must be returnable, and therefore cannot occur free in M_k or Σ_k. It cannot be the case that message $\mathtt{m}$ for a is contained in $\mathcal{M}$ (in which case the configuration could reduce), so typing ensures that a is free in one of the other threads. Extending this reasoning we see that if a configuration contains irreducible non-value threads, then C must contain a cyclic inter-thread dependency. □

E.3 Algorithmic Soundness

A solution for a set of constraints is also a solution for a subset of those constraints.

Lemma E.26. *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 $\mathrm{dom}(\Phi_1 \cup \Phi_2) \subseteq \mathrm{dom}(\Xi)$. The result follows from the fact that $\mathrm{dom}(\Phi_1) \subseteq \mathrm{dom}(\Phi_1 \cup \Phi_2) \subseteq \mathrm{dom}(\Xi)$. □

The pattern variables in an inferred environment must either occur in the type, program, or constraint set.

Lemma E.27. *If $M \Rightarrow_{\mathcal{P}} \tau \blacktriangleright \Theta; \Phi$ or $M \Leftarrow_{\mathcal{P}} \tau \blacktriangleright \Theta; \Phi$, then $pv(\Theta) \subseteq pv(\tau) \cup pv(\mathcal{P}) \cup pv(\Phi)$.*

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. □

Application of a usable substitution preserves algorithmic subtyping in the declarative setting.

Lemma E.28. *If $\tau \leq \sigma \blacktriangleright \Phi$ and Ξ is a usable solution of Φ with $pv(\sigma) \subseteq dom(\Xi)$, then $\Xi(\tau) \leq \Xi(\sigma)$.*

Proof. By case analysis on the derivation of $\tau \leq \sigma \blacktriangleright \Phi$. □

As a direct corollary, we can show that constraints generated by equivalence preserve subtyping in both directions.

Corollary E.29. *If $\tau \sim \sigma \blacktriangleright \Phi$ and Ξ is a usable solution of Φ with $pv(\sigma) \subseteq dom(\Xi)$, then both $\Xi(\tau) \leq \Xi(\sigma)$ and $\Xi(\sigma) \leq \Xi(\tau)$.*

Similarly, application of a usable substitution preserves unrestrictedness.

Lemma E.30. *If $unr(\tau) \blacktriangleright \Phi$ and Ξ is a usable solution of Φ with $pv(\sigma) \subseteq dom(\Xi)$, then there exists some A such that $un(A)$ and $A \leq \Xi(\tau)$.*

Proof. By case analysis on the derivation of $\mathrm{unr}(\tau) \blacktriangleright \Phi$, noting that cases are undefined for linear types, and the result follows straightforwardly for base types.

The only interesting case is $\mathrm{unr}(\,!\gamma^{\circ}) \blacktriangleright \mathbb{1} <: \gamma$; since Ξ is a usable solution, we have that $\mathbb{1} \sqsubseteq \Xi(\gamma)$ and $\Xi(\gamma) \not\sqsubseteq \mathbb{0}$.

Thus, it follows that $\Xi(\gamma) \simeq \mathbb{1}$ and therefore $!\mathbb{1} \leq \Xi(\gamma)$, noting that $\mathrm{un}(\,!\mathbb{1})$ as required. □

Again, this is straightforward to extend to environments.

Corollary E.31. *If Ξ is a usable solution of $unr(\Theta) \blacktriangleright \Phi$ where $pv(\Theta) \subseteq dom(\Xi)$, then there exists some Γ such that $\Gamma \leq \Xi(\Theta)$ and $un(\Gamma)$.*

If two environments are combinable, and we have a solution for the constraints generated by their algorithmic combination, then their combination is defined.

Lemma E.32. *If $\Theta_1 + \Theta_2 \blacktriangleright \Theta; \Phi$ and Ξ is a usable solution of Φ where $pv(\Theta_1) \cup pv(\Theta_2) \subseteq dom(\Xi)$, then there exists some Γ such that $\Gamma \leq \Xi(\Theta_2)$ and $\Xi(\Theta_1) + \Gamma = \Xi(\Theta)$.*

Proof. By induction on the derivation of $\Theta_1 + \Theta_2 \blacktriangleright \Theta; \Phi$.

Case $\Theta_1, x : \tau + \Theta_2 \blacktriangleright \Theta; \Phi$

Assumption:

$$\frac{x \notin \mathrm{dom}(\Theta_2) \qquad \Theta_1 + \Theta_2 \blacktriangleright \Theta; \Phi}{\Theta_1, x : \tau + \Theta_2 \blacktriangleright \Theta, x : \tau; \Phi}$$

We also assume that Ξ is a usable solution for Φ.
By the IH, we have that there exists some $\Gamma \leq \Xi(\Theta_2)$ such that $\Xi(\Theta_1) + \Gamma = \Xi(\Theta)$.
Since $x \notin \mathsf{dom}(\Theta_2)$, by the definition of $+$ in the declarative setting, we have that

$$\Xi(\Theta_1), x : \Xi(\tau) + \Xi(\Theta_2) \leq \Xi(\Theta), x : \Xi(\tau)$$

as required.

Case $\Theta_1 + \Theta_2, x : \tau \blacktriangleright \Theta; \Phi$

Symmetric to the first case.

Case $\Theta_1, x : \tau + \Theta_2, x : \sigma \blacktriangleright \Theta; \Phi$

Assumption:

$$\frac{\begin{array}{cc}\Theta_1 + \Theta_2 \blacktriangleright \Theta; \Phi_1 & \tau \sim \sigma \blacktriangleright \Phi_2 \\ \mathsf{unr}(\tau) \blacktriangleright \Phi_3 & \mathsf{unr}(\sigma) \blacktriangleright \Phi_4\end{array}}{\Theta_1, x : \tau + \Theta_2, x : \sigma \blacktriangleright \Theta, x : \tau; \Phi_1 \cup \cdots \cup \Phi_4}$$

We also assume that Ξ is a usable solution for $\Phi_1 \cup \cdots \cup \Phi_4$.
By the IH, there exists some Γ such that $\Gamma \leq \Xi(\Theta_2)$ and

$$\Xi(\Theta_1) + \Gamma = \Xi(\Theta)$$

By the definitions of $\sim$ and $\mathsf{unr}(-)$, and knowing that Ξ is a usable solution for $\Phi_2 \cup \Phi_3 \cup \Phi_4$, we have that either $\tau = \sigma = C$ for some base type C (in which case we can conclude with logic similar to the previous case), or $\tau = \,!\gamma^\circ$ and $\sigma = \,!\delta^\circ$ where $\Xi(\gamma), \Xi(\delta) \sqsubseteq \mathbb{1}$.
Since Ξ is usable, we know that $\Xi(\gamma), \Xi(\delta) \not\sqsubseteq \mathbb{0}$. Therefore, we have that $\tau, \sigma \leq \,!\mathbb{1}^\circ$.
We can then show that $\Gamma, x : \Gamma, x : \Xi(!\tau^\circ) \leq \Xi(\Theta_2), x : \Xi(!\sigma^\circ)$
and further that $\Xi(\Theta_1), x : \Xi(!\tau^\circ) + \Gamma, x : \Xi(!\tau^\circ) = \Xi(\Theta), x : \Xi(!\tau^\circ)$ as required.

□

We can generalise the previous result to an n-ary combination:

Corollary E.33. *If $\Theta_1 + \ldots + \Theta_n \blacktriangleright \Theta; \Phi$ where Ξ is a usable solution for Φ such that $pv(\Theta_1) \cup \cdots \cup pv(\Theta_n) \subseteq dom(\Xi)$, then there exist $(\Gamma_i \leq \Theta_i)_i$ such that $\Gamma_1 + \ldots + \Gamma_n = \Xi(\Theta)$.*

We now turn our attention to the relation between the algorithmic join and type combination operators.

Lemma E.34. *If $\tau_1 \,⨾\, \tau_2 \blacktriangleright \sigma; \Phi$ and Ξ is a usable solution of Φ such that $pv(\tau_1) \cup pv(\tau_2) \subseteq dom(\Xi)$, then there exist $\tau'_1 \leq \Xi(\tau_1)$, $\tau'_2 \leq \Xi(\tau_2)$ where $\tau'_1 \triangleright \tau'_2 = \Xi(\sigma)$.*

Proof. By case analysis on the derivation of $\tau_1 \,⨾\, \tau_2 \blacktriangleright \sigma; \Phi$.

Case $!\gamma^{\eta_1} \,⨾\, !\delta^{\eta_2} \blacktriangleright \,!(\gamma \odot \delta)^{\eta_1 \triangleright \eta_2}; \emptyset$

We can immediately conclude with $!(\Xi(\gamma))^{\eta_1} \triangleright \,!(\Xi(\delta))^{\eta_2} = \,!(\Xi(\gamma) \odot \Xi(\delta))^{\eta_1 \triangleright \eta_2}$ as required.

Case $!\gamma^{\eta_1} \,⨾\, ?\delta^{\eta_2} \blacktriangleright \,?(\alpha)^{\eta_1 \triangleright \eta_2}; \gamma \odot \alpha <: \delta$

Since Ξ is a solution for $\gamma \odot \alpha <: \delta$, we have that $\Xi(\gamma \odot \alpha) \sqsubseteq \Xi(\delta)$.
By expansion of $\Xi(-)$, we have that $\Xi(\gamma) \odot \Xi(\delta) \sqsubseteq \Xi(\delta)$.
Since receive mailbox types are covariant in their patterns, we can show that $?(\Xi(\gamma) \odot \Xi(\alpha)) \leq$ $?\Xi(\delta)$
and we can conclude that

$$!(\Xi(\gamma))^{\eta_1} \triangleright \,?(\Xi(\gamma) \odot \Xi(\alpha))^{\eta_2} = \,?\Xi(\alpha)^{\eta_1 \triangleright \eta_2}$$

as required.

Case $?\gamma \,⨟\, !\delta \blacktriangleright ?\alpha; \delta \odot \alpha <: \gamma$

Symmetric to the previous case.

Case $\tau \,⨟\, \sigma \blacktriangleright \tau; \Phi$

Assumption: τ, σ are not mailbox types and $\tau \sim \sigma \blacktriangleright \Phi$.
By Lemma E.28, $\Xi(\tau) \simeq \Xi(\sigma)$. Since neither type is a mailbox type we have that $\Xi(\tau) = \tau = \sigma = \Xi(\sigma) = C$ for some base type C, as required. □

We can extend this result to environments.

Lemma E.35. *If* $\Theta_1 \,⨟\, \Theta_2 \blacktriangleright \Theta; \Phi$ *and* Ξ *is a usable solution of* Φ *such that* $pv(\Theta_1) \cup pv(\Theta_2) \subseteq dom(\Xi)$, *then there exist* $\Gamma_1 \leq \Xi(\Theta_1)$ *and* $\Gamma_2 \leq \Xi(\Theta_2)$ *such that* $\Gamma_1 \triangleright \Gamma_2 = \Xi(\Theta)$.

Proof. A direct consequence of Lemma E.34. □

Lemma E.36. *If* $\tau_1 \sqcap \tau_2 \blacktriangleright \sigma; \Phi$ *and* Ξ *is a usable substitution of* Φ *such that* $pv(\tau_1) \cup pv(\tau_2) \subseteq dom(\Xi)$, *then* $\Xi(\sigma) \leq \Xi(\tau_1)$ *and* $\Xi(\sigma) \leq \Xi(\tau_2)$.

Proof. By case analysis on the derivation of $\tau_1 \sqcap \tau_2 \blacktriangleright \sigma; \Phi$.
For two mailbox types J^{η_1} and J^{η_2}, since $\bullet \leq \circ$ it is always the case that $\max(\eta_1, \eta_2) \leq \eta_1$ and $\max(\eta_1, \eta_2) \leq \eta_2$, so therefore it suffices to consider the non-usage-annotated merge $\pi_1 \sqcap \pi_2 \blacktriangleright \rho; \emptyset$

Case $!\gamma \sqcap !\delta \blacktriangleright !(\gamma \oplus \delta); \emptyset$

By appeal to the definition of $[\![-]\!]$ we have that $[\![\Xi(\gamma) \oplus \Xi(\delta)]\!] = [\![\Xi(\gamma)]\!] \uplus [\![\Xi(\delta)]\!]$ and therefore:

- $[\![\Xi(\gamma)]\!] \subseteq [\![\Xi(\gamma) \oplus \Xi(\delta)]\!]$; and
- $[\![\Xi(\delta)]\!] \subseteq [\![\Xi(\gamma) \oplus \Xi(\delta)]\!]$

By the reflexivity of subtyping and the definition of pattern inclusion, it follows that:

- $\Xi(\gamma) \sqsubseteq (\Xi(\gamma) \oplus \Xi(\delta))$; and
- $\Xi(\delta) \sqsubseteq (\Xi(\gamma) \oplus \Xi(\delta))$

Therefore, since output mailbox types are contravariant in their patterns, it follows that both:

- $!(\Xi(\gamma) \oplus \Xi(\delta)) \leq !(\Xi(\gamma))$; and
- $!(\Xi(\gamma) \oplus \Xi(\delta)) \leq !(\Xi(\delta))$

as required.

Case $?\gamma \sqcap ?\delta \blacktriangleright ?\alpha; \{\alpha <: \gamma, \alpha <: \delta\}$

(where α fresh).
Since Ξ is a usable solution, it follows that:

- $\Xi(\alpha) \sqsubseteq \Xi(\gamma)$; and
- $\Xi(\alpha) \sqsubseteq \Xi(\delta)$

Since receive mailbox types are covariant in their pattern arguments, it follows that:

- $?(\Xi(\alpha)) \leq ?(\Xi(\gamma))$; and
- $?(\Xi(\alpha)) \leq ?(\Xi(\delta))$

as required.

Case $\tau \sqcap \sigma \blacktriangleright \tau; \Phi$

where τ, σ are not mailbox types and $\tau \sim \sigma \blacktriangleright \Phi$.

By Lemma E.29:

(1) $\Xi(\tau) \leq \Xi(\sigma)$; and
(2) $\Xi(\sigma) \leq \Xi(\tau)$.

By the reflexivity of subtyping we have that $\Xi(\tau) \leq \Xi(\tau)$, and for the second obligation we can conclude with (2) as required. □

Lemma E.37. *If $\Theta_1 \sqcap \Theta_2 \blacktriangleright \Theta; \Phi$ and Ξ is a usable solution of Φ such that $pv(\Theta_1) \cup pv(\Theta_2) \subseteq dom(\Xi)$, then $\Xi(\Theta) \leq \Xi(\Theta_1)$ and $\Xi(\Theta) \leq \Xi(\Theta_2)$.*

Proof. By induction on the derivation of $\Theta_1 \sqcap \Theta_2 \blacktriangleright \Theta; \Phi$ with appeal to Lemma E.36. □

Lemma E.38 (Subpattern PNF). *If $E \vDash F$ and $E \sqsubseteq F$, then $\vDash F$.*

Proof. For it to be the case that $E \vDash F$ it must be the case that $F = F_1 \oplus \cdots \oplus F_n$ where $E \vDash_{\mathsf{lit}} F_i$ for $i \in 1..n$.

It suffices to consider the case where we have some $F_j = \mathtt{m}_j \odot F'_j$ where $F_j \not\sqsubseteq E$. In this case, the following must hold:

$$\frac{F_j \simeq E \,/\, \mathtt{m}_j}{E \vDash_{\mathsf{lit}} \mathtt{m}_j \odot F_j}$$

and by the definition of pattern residual and the fact that $\mathtt{m}_j \not\sqsubseteq E$ it must be the case that $E \,/\, \mathtt{m}_j \simeq \mathbb{0}$. Consequently we know that $F_j \simeq \mathbb{0}$.

To ensure that $\vDash F$ we need to show $F \vDash F$ and therefore that $\mathbb{0} \simeq F \,/\, \mathtt{m}_j$, which follows by the definition of pattern derivative as required. □

Algorithmic soundness relies on the following generalised result:

Lemma E.39 (Algorithmic Soundness (Generalised)).

- *If $\vdash \mathcal{P} \triangleright \Phi_1$ and $M \Rightarrow_{\mathcal{P}} \tau \blacktriangleright \Theta;\ \Phi_2$ where Ξ is a usable solution of $\Phi_1 \cup \Phi_2$ and $pv(\tau) \cup pv(\mathcal{P}) \subseteq dom(\Xi)$, then $\Xi(\Theta) \vdash_{\Xi(\mathcal{P})} M : \Xi(\tau)$.*
- *If $\vdash \mathcal{P} \triangleright \Phi_1$ and $M \Leftarrow_{\mathcal{P}} \tau \blacktriangleright \Theta;\ \Phi_2$ where Ξ is a usable solution of $\Phi_1 \cup \Phi_2$ and $pv(\tau) \cup pv(\mathcal{P}) \subseteq dom(\Xi)$, then $\Xi(\Theta) \vdash_{\Xi(\mathcal{P})} M : \Xi(\tau)$.*
- *If $\vdash \mathcal{P} \triangleright \Phi_1$ and $\{E\}\ G \Leftarrow_{\mathcal{P}} \tau \blacktriangleright \Theta;\ \Phi;\ F$ where Ξ is a usable solution of $\Phi_1 \cup \Phi_2$ and $pv(\tau) \cup pv(\mathcal{P}) \subseteq dom(\Xi)$, then $\Xi(\Theta) \vdash_{\Xi(\mathcal{P})} G : \Xi(\tau) :: F$ and $E \vDash_{lit} F$.*
- *If $\vdash \mathcal{P} \triangleright \Phi_1$ and $\{E\}\ \overrightarrow{G} \Leftarrow_{\mathcal{P}} \tau \blacktriangleright \Theta;\ \Phi;\ F$ where Ξ is a usable solution of $\Phi_1 \cup \Phi_2$ and $pv(\tau) \cup pv(\mathcal{P}) \subseteq dom(\Xi)$, then $\Xi(\Theta) \vdash_{\Xi(\mathcal{P})} G : \Xi(\tau) :: F$ and $E \vDash_{lit} F$.*

Proof. By mutual induction on all statements. We inline our proof of statement 4 with TC-Guard.

We know in all cases that the solution covers the pattern variables in the program, return type, and constraints. Therefore by Lemma E.27 we know that any produced environment will contain pattern variables contained in the solution. We make use of this fact implicitly throughout the proof.

Statement 1: Synthesis.

Case TS-Base

Assumption:

$$\frac{c \text{ has base type } D}{c \Rightarrow D \blacktriangleright \cdot; \emptyset}$$

By T-Const:

$$\frac{}{\cdot \vdash c : D}$$

noting that:

- $\mathsf{un}(\cdot)$
- $\Xi(c) = c$
- $\Xi(D) = D$

as required.

Case TS-Unit

Similar to TS-Base.

Case TS-New

Similar to TS-Base.

Case TS-Spawn

Assumption:

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

Furthermore, we assume that Ξ is a usable solution for Φ.
By the IH (2),

$$\Xi(\Theta) \vdash M : \mathbf{1}$$

Recomposing by T-Spawn:

$$\frac{\Xi(\Theta) \vdash M : \mathbf{1}}{\lceil \Xi(\Theta) \rceil \vdash \mathbf{spawn}\ M : \mathbf{1}}$$

as required.

Case TS-Send

Assumption:

$$\frac{\begin{array}{cc} \mathcal{P}(\mathtt{m}) = \overrightarrow{\pi} & V \Leftarrow\ !\mathtt{m}^{\circ} \blacktriangleright \Theta'; \Phi \\ (W_i \Leftarrow \lceil \pi_i \rceil \blacktriangleright \Theta'_i; \Phi'_i)_{i \in 1..n} & \Theta' + \Theta'_1 + \cdots + \Theta'_n \blacktriangleright \Theta; \Phi'' \end{array}}{V\ !\ \mathtt{m}[\overrightarrow{W}] \Rightarrow \mathbf{1} \blacktriangleright \Theta; \Phi \cup \Phi'_1 \cup \cdots \cup \Phi'_n \cup \Phi''}$$

Also, we assume $\vdash \mathcal{P} \triangleright \Phi_{prog}$.

Furthermore, we assume that Ξ is a solution for $\Phi_{prog} \cup \Phi \cup \Phi'_1 \cup \cdots \cup \Phi'_n \cup \Phi''$. By Lemma E.26, we have that Ξ is also a solution for each constraint set individually.

Thus, by the IH:

- $\Xi(\Theta') \vdash V : !\mathtt{m}^{\circ}$
- $\Xi(\Theta'_i) \vdash W_i : \lceil \Xi(\pi_i) \rceil$ for $i \in 1..n$

By Corollary E.33, there exist $\Gamma' \leq \Xi(\Theta')$ and $\Gamma'_i \leq \Xi(\Theta'_i)$ for $i \in 1..n$ such that $\Gamma' + \Gamma'_1 + \ldots + \Gamma'_n = \Xi(\Theta)$. Therefore:

$$\frac{\Xi(\mathcal{P})(\mathtt{m}) = \overrightarrow{\Xi(\pi)} \qquad \dfrac{\Xi(\Theta') \vdash V : !\mathtt{m}^{\circ}}{\Gamma' \vdash V : !\mathtt{m}^{\circ}} \qquad \dfrac{(\Xi(\Theta'_i) \vdash W_i : \Xi(A_i))_{i \in 1..n}}{(\Gamma'_i \vdash W_i : \Xi(A_i))_{i \in 1..n}}}{\Gamma' + \Gamma'_1 + \ldots + \Gamma'_n \vdash V \,!\, \mathtt{m}[\overrightarrow{W}] : \mathbf{1}}$$

as required.

Case TS-App

Assumption:

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

Also, we assume $\vdash \mathcal{P} \triangleright \Phi_{prog}$.
We can also assume that there exists some Ξ which is a usable solution of $\Phi_{prog} \cup \Phi_1 \cup \ldots \cup \Phi_n$.
By Lemma E.26, we have that Ξ is a solution for all Φ_i individually.
By the IH, $\Xi(\Theta_i) \vdash V_i : \Xi(\tau_i)$ for all i.
By Corollary E.33, there exist $(\Gamma_i \leq \Theta_i)_{i \in 1..n}$ such that $\Gamma_1 + \ldots + \Gamma_n = \Xi(\Theta)$.
Thus by T-Subs and T-App:

$$\frac{\Xi(\mathcal{P}(f)) = \overrightarrow{\Xi(\tau)} \to \Xi(\sigma) \qquad \dfrac{(\Xi(\Theta_i) \vdash V_i : \Xi(\tau_i))_{i \in 1..n}}{(\Gamma_i \vdash V_i : \Xi(\tau_i))_{i \in 1..n}}}{\Gamma_1 + \cdots + \Gamma_n \vdash f(\overrightarrow{V}) : \Xi(\sigma)}$$

as required.

Statement 2: Checking.

Case TC-Var

Assumption:

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

By T-Var:

$$\frac{}{x : \Xi(\tau) \vdash x : \Xi(\tau)}$$

as required.

Case TC-Let

Assumption:

$$\frac{\begin{array}{cc} \text{TC-Let} & \\ M \Leftarrow \lfloor T \rfloor \blacktriangleright \Theta_1;\, \Phi_1 & N \Leftarrow \sigma \blacktriangleright \Theta_2;\, \Phi_2 \\ \mathsf{check}(\Theta_2, x, \lfloor T \rfloor) = \Phi_3 & \Theta_1 - x \,\fatsemi\, \Theta_2 \blacktriangleright \Theta; \Phi_4 \end{array}}{\mathbf{let}\; x \colon T = M \;\mathbf{in}\; N \Leftarrow \sigma \blacktriangleright \Theta;\, \Phi_1 \cup \cdots \cup \Phi_4}$$

We also assume that we have some usable solution Ξ for $\Phi_1 \cup \cdots \cup \Phi_4$, and by Lemma E.26, we know that Ξ is a usable solution for all Φ_i individually.

By the IH, we have that:

- $\Xi(\Theta_1) \vdash M : \lfloor \Xi(T) \rfloor$
- $\Xi(\Theta_2) \vdash N : \Xi(\sigma)$

Since T does not contain any type variables we have that $\Xi(\lfloor T \rfloor) = \lfloor T \rfloor$.

By Lemma E.35, there exist some Γ_1, Γ_2 such that $\Gamma_1 \leq \Xi(\Theta_1 - x)$, $\Gamma_2 \leq \Xi(\Theta_2)$ and $\Gamma_1 \triangleright \Gamma_2 = \Xi(\Theta)$

By the definition of check, we have two subcases based on whether $x \in \mathrm{dom}(\Theta_2)$:

Subcase $x \notin \mathbf{dom}(\Theta_2)$

In this case we have that $\mathsf{unr}(\lfloor T \rfloor) \blacktriangleright \Phi$.

By Lemma E.30, we have that $\mathsf{un}(\lfloor T \rfloor)$.

Thus by T-Let and T-Subs:

$$\frac{\Xi(\Theta_1) \vdash M : \lfloor T \rfloor \qquad \dfrac{\dfrac{\Xi(\Theta_2) \vdash N : \Xi(\sigma)}{\Xi(\Theta_2), x : \lfloor T \rfloor \vdash N : \Xi(\sigma)}}{\Gamma \vdash N : \Xi(\sigma)}}{\Xi(\Theta) \vdash \mathbf{let}\; x \colon T = M \;\mathbf{in}\; N : \Xi(\sigma)}$$

as required.

Subcase $x \in \mathbf{dom}(\Theta_2)$

In this case, we have that $x : \lfloor T \rfloor \in \Theta_2$ and $\lfloor T \rfloor \leq \sigma \blacktriangleright \Phi_1$.

By Lemma E.28, $\lfloor T \rfloor \leq \Xi(\sigma)$.

Thus by T-Let and T-Subs:

$$\frac{\Xi(\Theta_1) \vdash M : \lfloor T \rfloor \qquad \dfrac{\Xi(\Theta_2), x : \lfloor T \rfloor \vdash N : \Xi(\sigma)}{\Gamma \vdash N : \Xi(\sigma)}}{\Xi(\Theta) \vdash \mathbf{let}\; x \colon T = M \;\mathbf{in}\; N : \Xi(\sigma)}$$

as required.

Case TC-Guard

Assumption:

$$\frac{\dfrac{\begin{array}{c}(\{E\}\; G_i \Leftarrow \tau \blacktriangleright \Psi_i;\, \Phi_i;\, F_i)_{i \in 1..n} \\ \Psi_1 \sqcap \ldots \sqcap \Psi_n \blacktriangleright \Psi; \Phi \qquad \Phi' = \bigcup_{i \in 1..n} \Phi_i \end{array}}{\{E\}\; \overrightarrow{G} \Leftarrow \tau \blacktriangleright \Psi;\, \Phi \cup \Phi';\, F_1 \oplus \ldots \oplus F_n} \qquad M \Leftarrow ?F^{\bullet} \blacktriangleright \Theta';\, \Phi_2 \qquad \Psi + \Theta' \blacktriangleright \Theta; \Phi_3}{\mathbf{guard}\; V : E\; \{\overrightarrow{G}\} \Leftarrow \tau \blacktriangleright \Theta;\, \Phi \cup \Phi' \cup \Phi_2 \cup \Phi_3 \cup \{E <: F\}}$$

where $F = F_1 \oplus \cdots \oplus F_n$.

Since guards must be unique we know that there will be at most one **fail** branch in $\overrightarrow{G}$. Without loss of generality assume that $G_1 = \textbf{fail}$ (the order of guards does not matter, and the argument is the same if there is no **fail** guard).

Let us assume without loss of generality that $n > 1$ (i.e., **fail** is not the only guard).

Thus we have that:

- $\{E\}\ \textbf{fail} \Leftarrow \tau \blacktriangleright \top;\ \emptyset;\ \mathbb{0}$ (i.e., $\Psi_1 = \top, \Phi_1 = \emptyset, F_1 = \mathbb{0}$)
- $\{E\}\ G_i \Leftarrow \tau \blacktriangleright \Theta_i;\ \Phi_i;\ F_i$ for $2 \leq i \leq n$
- $\Theta_2 \sqcap \ldots \sqcap \Theta_n \blacktriangleright \Theta; \Phi$

By repeated use of the induction hypothesis (statement 3), we have that $\Xi(\Theta_i) \vdash G_i : \Xi(\tau) :: F_i$ where $E \vDash_{\mathsf{lit}} F_i$ for $2 \leq i \leq n$.

Since $F = F_1 \oplus \cdots \oplus F_n$ and $E \vDash_{\mathsf{lit}} F_i$ for $i \in 1..n$, it follows by the definition of pattern normal form that $E \vDash F$.

Since Ξ is a usable solution of the constraint set we have that $E \sqsubseteq F$.

Now since $E \vDash F$ and $E \sqsubseteq F$, by Lemma E.38 we have that $\vDash F$.

By Lemma E.37, we have that there exists some Γ such that $\Gamma \leq \Theta_i$ for each i. Thus, by T-Subs, we can show: $\Gamma \vdash G_i : \Xi(\tau) :: F_i$.

Therefore, by T-GuardSeq we can show that $\Gamma \vdash \textbf{fail} \cdot G_2 \cdot \ldots \cdot G_n : \Xi(\tau) :: F$.

By the IH (statement 2), we have that $\Xi(\Theta') \vdash V : ?F$.

By Lemma E.32, there exists some $\Theta'' \leq \Theta'$ such that $\Gamma + \Xi(\Theta'') = \Xi(\Theta)$.

Thus, we can show:

$$\frac{\dfrac{\Xi(\Theta') \vdash V : ?F^{\bullet}}{\Xi(\Theta'') \vdash V : ?F^{\bullet}} \qquad \Gamma \vdash \textbf{fail} \cdot \overrightarrow{G} : \Xi(\tau) :: F \qquad E \sqsubseteq F \qquad \vDash F}{\Xi(\Theta'') + \Gamma \vdash \textbf{guard}\ V : E\ \{\textbf{fail} \cdot \overrightarrow{G}\} : \Xi(\tau)}$$

as required.

Case TC-Sub

Assumption:

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

By the IH (statement 1), $\Xi(\Theta) \vdash M : \Xi(\tau)$.

By Lemma E.28, $\Xi(\tau) \leq \Xi(\sigma)$.

Therefore by T-Sub:

$$\frac{\Xi(\Theta) \vdash M : \Xi(\tau)}{\Xi(\Theta) \vdash M : \Xi(\sigma)}$$

as required.

Statement 3: Guards. Note that there is no case for TCG-Fail since (contrary to the theorem statement) it is not typable under a non-null typing environment.

Case TCG-Free

Assumption:

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

By the IH (statement 2), we have that $\Xi(\Theta) \vdash M : \Xi(\tau)$.
Trivially, $E \vDash_{\text{lit}} \mathbb{1}$.
Therefore, we can reconstruct by TG-Free:

$$\frac{\Xi(\Theta) \vdash M : \Xi(\tau)}{\Xi(\Theta) \vdash \textbf{free} \mapsto M : \Xi(\tau) :: \mathbb{1}}$$

as required.

Case TCG-Recv

Assumption:

$$\text{TCG-Recv}\ \frac{\mathcal{P}(\mathtt{m}) = \overrightarrow{\pi} \qquad \Theta = \Theta' - \overrightarrow{x} \qquad \begin{array}{c} M \Leftarrow \sigma \blacktriangleright \Theta', y : ?\delta^{\bullet};\ \Phi_1 \\ \mathsf{base}(\overrightarrow{\pi}) \vee \mathsf{base}(\Theta') \end{array} \qquad \mathsf{check}(\Theta', \overrightarrow{x}, \overrightarrow{\lceil \pi \rceil}) = \Phi_3}{\{E\}\ \textbf{receive}\ \mathtt{m}[\overrightarrow{x}]\ \textbf{from}\ y \mapsto M \Leftarrow \sigma \blacktriangleright \Theta;\ \Phi_1 \cup \Phi_2 \cup \Phi_3 \cup \{E \,/\, \mathtt{m} <: \delta\};\ \mathtt{m} \odot (E \,/\, \mathtt{m})}$$

We also assume that we have some usable solution Ξ for $\Phi_1 \cup \Phi_2 \cup \Phi_3 \cup \{E \,/\, \mathtt{m} <: \delta\}$.
As usual, by Lemma E.26 we can assume that Ξ is a usable solution for all Φ_i.
By the IH, $\Xi(\Theta'), y : ?\Xi(\delta)^{\bullet} \vdash M : \Xi(\sigma)$.
Suppose $\Theta' = \Theta, x_1 : \tau_1, \ldots, x_m : \tau_m$ and $\overrightarrow{x} = x_1, \ldots, x_m$.
Then by the definition of check we have that:

- $(\tau_i \leq \lceil \pi_i \rceil \blacktriangleright \Phi'_i)_{i \in 1..m}$
- $(\mathsf{unr}(\tau_i) \blacktriangleright \Phi'_i)_{i \in m+1..n}$

Thus by Lemma E.28, $\lceil \Xi(\pi_i) \rceil \leq \Xi(\tau_i)$ for each $i \in 1..m$.
By Lemma E.30, there exist $A_j \leq \Xi(\tau_j)$ such that $\mathsf{un}(A_j)$ for $j \in m+1..n$.
Thus it follows by the definition of environment subtyping that $\Xi(\Theta) \leq \Xi(\Theta')$.
It follows from the fact that pattern substitution preserves type shape that if $\mathsf{base}(\overrightarrow{T}) \vee \mathsf{base}(\Theta')$, we have that $\mathsf{base}(\overrightarrow{\Xi(T)}) \vee \mathsf{base}(\Xi(\Theta'))$.
Since Ξ is a usable solution of $E \,/\, \mathtt{m} <: \delta$ we know that $E \,/\, \mathtt{m} \sqsubseteq \Xi(\delta)$ and therefore that $?(E \,/\, \mathtt{m}) \leq ?(\Xi(\delta))$.
It remains to be shown that $E \vDash_{\text{lit}} \mathtt{m} \odot (E \,/\, \mathtt{m})$:

$$\frac{\mathtt{m} \odot (E \,/\, \mathtt{m}) \simeq E}{E \vDash_{\text{lit}} \mathtt{m} \odot (E \,/\, \mathtt{m})}$$

The pattern residual and concatenation cancel, so the premise holds and therefore we can conclude that $E \vDash_{\text{lit}} \mathtt{m} \odot (E \,/\, \mathtt{m})$.
Finally, we can reconstruct using TG-Recv:

$$\frac{\Xi(\mathcal{P})(\mathtt{m}) = \overrightarrow{\Xi(\pi)} \qquad \mathrm{base}(\overrightarrow{\Xi(\pi)}) \vee \mathrm{base}(\Xi(\Theta)) \qquad \dfrac{\Xi(\Theta'), y : ?\Xi(\delta)^{\bullet} \vdash M : \Xi(\sigma)}{\Xi(\Theta), y : ?(E \,/\, \mathtt{m})^{\bullet}, \overrightarrow{x} : \overrightarrow{\lceil T \rceil} \vdash M : \Xi(\sigma)}}{\Xi(\Theta) \vdash \textbf{receive}\ \mathtt{m}[\overrightarrow{x}]\ \textbf{from}\ y \mapsto M : \Xi(\sigma) :: \mathtt{m} \odot (E \,/\, \mathtt{m})}$$

as required.

□

Theorem 4.4 (Algorithmic Soundness).

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

Proof. A direct consequence of Lemma E.39. □

E.4 Algorithmic Completeness

Every Γ is also a valid Θ and every A is a valid τ. We will therefore allow ourselves to use Γ and A in algorithmic type system derivations directly.

Definition E.40 (Closed program). A program $(\mathcal{S}, \overrightarrow{D}, M)$ is *closed* if $\text{pv}(\mathcal{S}) = \emptyset$ and $\text{pv}(\overrightarrow{D}) = \emptyset$.

E.4.1 Useful auxiliary lemmas. We begin by stating two useful results. The first lemma states that values are typable in the algorithmic system without constraints.

Lemma E.41. *If $\Gamma \vdash V : A$, then there exists some Γ' such that $\Gamma \leq \Gamma'$ and $V \Leftarrow A \blacktriangleright \Gamma'; \emptyset$.*

Proof. The proof is by induction on the derivation of $\Gamma \vdash V : A$.

Case T-Var

We assume that $x : A \vdash x : A$. By TC-Var we can show $x \Leftarrow A \blacktriangleright x : A; \emptyset$, as required.

Case T-Const

We assume that $\cdot \vdash c : C$, where c has base type C. By TS-Base, we can show that $c \Rightarrow C \blacktriangleright \cdot; \emptyset$. Finally, by TC-Sub (noting that $C \leq C \blacktriangleright \emptyset$) we have that $c \Leftarrow C \blacktriangleright \cdot; \emptyset$, as required.

Case T-Subs

Assumption:

$$\frac{\Gamma \leq \Gamma' \qquad A \leq B \qquad \Gamma' \vdash V : A}{\Gamma \vdash V : B}$$

By the IH, there exists some Γ'' such that $\Gamma' \leq \Gamma''$ and $\Gamma'' \Leftarrow V \blacktriangleright A; \emptyset$.
By the transitivity of subtyping, we have that $\Gamma \leq \Gamma' \leq \Gamma''$, as required. □

Lemma E.42. *If $M \Rightarrow \tau \blacktriangleright \Theta; \Phi$, then $M \Leftarrow \tau \blacktriangleright \Theta; \Phi$.*

Proof. Follows from the definition of TC-Subs, noting that the subtyping constraint is instantiated as $\tau \leq \tau \blacktriangleright \Phi$. There are two ways we can create a derivation of $\tau \leq \tau \blacktriangleright \Phi$: either if $\tau = C$ and we have $\tau \leq \tau \blacktriangleright \emptyset$, or if τ is a mailbox type (take an output mailbox type here, although the reasoning is the same for an input mailbox). In this case, we would have a derivation of $!\gamma^{\eta} \leq\, !\gamma^{\eta} \blacktriangleright \gamma <: \gamma$. Since $\gamma <: \gamma$ is a tautology, it follows that we need not add an additional constraint and can show $!\gamma^{\eta} \leq\, !\gamma^{\eta} \blacktriangleright \emptyset$.

Using TC-Subs we can construct:

$$\frac{M \Rightarrow \tau \blacktriangleright \Theta; \Phi \qquad \tau \leq \tau \blacktriangleright \emptyset}{M \Leftarrow \tau \blacktriangleright \Theta; \Phi}$$

as required. □

E.4.2 Completeness of auxiliary definitions. We now need to show completeness for all auxiliary judgements (e.g., subtyping, environment combination).

Lemma E.43 (Completeness of type join). *If:*

- $A_1 \triangleright A_2 = B$,
- $A_1 \leq \Xi_1(\tau_1)$,
- $A_2 \leq \Xi_2(\tau_2)$; *and*
- $pv(\Xi_1) \cap pv(\Xi_2) = \emptyset$

then there exist Θ, 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 $B \leq \Xi(\sigma)$.

Proof. We proceed by case analysis on the derivation of $A_1 \triangleright A_2 \;=\; B$. Base types follow straightforwardly, so we concentrate on mailbox types.

Case $A_1 = {!}E_1^{\eta_1}, A_2 = {!}E_2^{\eta_2}$

Assumption:

$$\dfrac{\dfrac{}{{!}E_1 \triangleright {!}E_2 = {!}E_1 \odot E_2}}{{!}E_1^{\eta_1} \triangleright {!}E_2^{\eta_2} = {!}(E_1 \odot E_2)^{\eta_1 \triangleright \eta_2}}$$

Since the domains of Ξ_1, Ξ_2 are disjoint, let $\Xi = \Xi_1 \cup \Xi_2$.

In order for ${!}E_i \leq \Xi_i(\tau_i)$ (for $i \in 1, 2$) to hold, it must be the case that $\tau_i = {!}\gamma_i$ with $\Xi(\gamma_i) \sqsubseteq E_i$.

In this case we can construct a derivation:

$$\dfrac{{!}\gamma_1 \fatsemi {!}\gamma_2 \blacktriangleright {!}(\gamma_1 \odot \gamma_2); \emptyset}{{!}\gamma_1^{\eta_1} \fatsemi {!}\gamma_2^{\eta_2} \blacktriangleright {!}(\gamma_1 \odot \gamma_2)^{\eta_1 \triangleright \eta_2}; \emptyset}$$

Noting that $\Xi(\gamma_i) \sqsubseteq E_i$, it follows by the definition of pattern semantics that $\Xi(\gamma_1 \odot \gamma_2) \sqsubseteq E_1 \odot E_2$ and therefore that ${!}(\gamma_1 \odot \gamma_2)^{\eta_1 \triangleright \eta_2} \leq \Xi({!}(\gamma_1 \odot \gamma_2))^{\eta_1 \triangleright \eta_2}$

with $\Xi \supseteq \Xi_1 \cup \Xi_2$ and a solution of $\emptyset$, as required.

Case $A_1 = {!}E^{\eta_1}, A_2 = {?}(E \odot F)^{\eta_2}$

Assumption:

$$\dfrac{\dfrac{}{{!}E \triangleright {?}(E \odot F) = {?}F}}{{!}E^{\eta_1} \triangleright {?}(E \odot F)^{\eta_2} = F^{\eta_1 \triangleright \eta_2}}$$

Since the domains of Ξ_1, Ξ_2 are disjoint, let $\Xi = \Xi_1 \cup \Xi_2$.

In order for ${!}E \leq \Xi_1(\tau_1)$ to hold, it must be the case that $\tau_1 = {!}\gamma$ with $\Xi_1(\gamma) \sqsubseteq E$.

Similarly, for ${?}(E \odot F) \leq \Xi_2(\tau_2)$ to hold, it must be the case that $\tau_2 = {?}\delta$ with $(E \odot F) \sqsubseteq \Xi_2(\delta)$.

In this case we can construct a derivation: Using algorithmic type joining, we can construct the following derivation:

$$\dfrac{\dfrac{\alpha \text{ fresh}}{{!}\gamma \fatsemi {?}\delta \blacktriangleright {?}\alpha; \{\delta \odot \alpha <: \delta\}}}{{!}\gamma^{\eta_1} \fatsemi {?}\delta^{\eta_2} \blacktriangleright {?}\alpha^{\eta_1 \triangleright \eta_2}; \{\gamma \odot \alpha <: \delta\}}$$

At this point we know that the domains of Ξ_1 and Ξ_2 are disjoint. Let us construct $\Xi = \Xi_1 \cup \Xi_2 \cup \alpha \mapsto F$.

It remains to be shown that Ξ is a solution; it suffices to show that $(\Xi(\gamma) \odot F) \sqsubseteq \Xi(\delta)$.

By the transitivity of pattern inclusion we have that

$$(\Xi(\gamma) \odot F) \sqsubseteq (E \odot F) \sqsubseteq \Xi(\delta)$$

We have that $\Xi({?}\alpha) = {?}F$

and therefore we have that (trivially) ${?}F \leq {?}F$

with $\Xi \supset \Xi_1 \cup \Xi_2$ a solution for the constraint set, as required.

Case $A_1 = \mathbf{?}E \odot F^{\eta_1}, A_2 = \mathbf{!}E^{\eta_2}$

Similar to the above case.

□

Lemma E.44 (Completeness of environment join). *If:*

- $\Gamma_1 \triangleright \Gamma_2 = \Gamma$,
- $\Gamma_1 \preccurlyeq \Xi_1(\Theta_1)$,
- $\Gamma_2 \preccurlyeq \Xi_2(\Theta_2)$*; and*
- $pv(\Xi_1) \cap pv(\Xi_2) = \emptyset$

then there exist Θ, Φ *such that* $\Theta_1 \fatsemi \Theta_2 \blacktriangleright \Theta; \Phi$*, and there exists a usable solution* $\Xi \supseteq \Xi_1 \cup \Xi_2$ *of* Φ *such that* $\Gamma \leq \Xi(\Theta)$.

Proof. By induction on the derivation of $\Gamma_1 \triangleright \Gamma_2$, with appeal to Lemma E.43. □

Lemma E.45 (Completeness of disjoint environment combination). *If:*

- $\Gamma_1 + \Gamma_2 = \Gamma$,
- $\Gamma_1 \preccurlyeq \Xi_1(\Theta_1)$,
- $\Gamma_2 \preccurlyeq \Xi_2(\Theta_2)$*; and*
- $pv(\Xi_1) \cap pv(\Xi_2) = \emptyset$

then there exist Θ, Φ *such that* $\Theta_1 + \Theta_2 \blacktriangleright \Theta; \Phi$*, and there exists a usable solution* $\Xi \supseteq \Xi_1 \cup \Xi_2$ *of* Φ *such that* $\Gamma \leq \Xi(\Theta)$.

Proof. By induction on the derivation of $\Gamma_1 + \Gamma_2 = \Gamma$.

Case $\Gamma_1 = \cdot$ and $\Gamma_2 = \cdot$

$$\frac{}{\cdot + \cdot = \cdot}$$

By the definition of environment subtyping, the only environment that can be a supertype of the empty environment is $\cdot$. Therefore, we can immediately conclude with the corresponding base case in algorithmic type environment combination:

$$\frac{}{\cdot + \cdot \blacktriangleright \cdot; \emptyset}$$

Case $x \notin \mathrm{dom}(\Gamma_2)$

Assumption:

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

where:

- $\Gamma_1, x : A \preccurlyeq \Xi_1(\Theta_1)$
- $\Gamma_2 \preccurlyeq \Xi_2(\Theta_2)$
- $\mathrm{pv}(\Xi_1) \cap \mathrm{pv}(\Xi_2) = \emptyset$

Since we are considering strict subtyping on environments rather than general subtyping, we can assume that $x : A \in \mathrm{dom}(\Theta_1)$. Therefore, let $\Theta_1 = \Theta_1', x : \tau$ with $A \leq \Xi_1(\tau)$.

By the IH, $\Theta_1' + \Theta_2 \blacktriangleright \Theta; \Phi$ for some Θ, Φ and there exists some usable solution $\Xi \supseteq \Xi_1 \cup \Xi_2$ of Φ such that $\Gamma_1 + \Gamma_2 \leq \Xi(\Theta)$.

Since $A \leq \Xi_1(\tau)$ and $\Xi_1 \subseteq \Xi$, it follows that $A \leq \Xi(\tau)$.

Therefore it follows that $\Gamma_1 + \Gamma_2, x : A \leq \Xi(\Theta, x : \tau)$ as required.

Case $x \notin \mathsf{dom}(\Gamma_1)$

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

Symmetric to the above case.

Case $x \in \mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2)$

$$\frac{\mathsf{un}(A) \qquad \Gamma_1 + \Gamma_2 = \Gamma}{\Gamma_1, x : A + \Gamma_2, x : A = \Gamma, x : A}$$

In this case, we have that:

- $\Theta_1, = \Theta'_1, x : \sigma_1$
- $\Theta_2 = \Theta'_2, x : \sigma_2$

By the IH, there exist Θ, Φ such that $\Theta'_1 + \Theta'_2 \blacktriangleright \Theta; \Phi$ and some usable solution $\Xi \supseteq \Xi_1 \cup \Xi_2$ of Φ such that $\Gamma \leq \Xi(\Theta)$.

By algorithmic environment combination we have:

$$\frac{\Theta'_1 + \Theta'_2 \blacktriangleright \Theta; \Phi_1 \qquad \sigma_1 \sim \sigma_2 \blacktriangleright \Phi_2 \qquad \mathsf{unr}(\sigma_1) \blacktriangleright \Phi_3 \qquad \mathsf{unr}(\sigma_2) \blacktriangleright \Phi_4}{\Theta'_1, x : \sigma_1 + \Theta'_2, x : \sigma_2 \blacktriangleright \Theta, x : B_1; \Phi_1 \cup \Phi_2 \cup \Phi_3 \cup \Phi_4}$$

From $\mathsf{un}(A)$, we have two subcases based on whether A is a base type C, or a mailbox type $!\mathbb{1}^\circ$.

Subcase $A = C$

In this case, by the definition of subtyping we have that $B_1 = B_2 = C$ and therefore:

$$\frac{\Theta'_1 + \Theta'_2 \blacktriangleright \Theta; \Phi \qquad C \sim C \blacktriangleright \emptyset \qquad \mathsf{unr}(C) \blacktriangleright \emptyset \qquad \mathsf{unr}(C) \blacktriangleright \emptyset}{\Theta'_1, x : C + \Theta'_2, x : C \blacktriangleright \Theta, x : C; \Phi}$$

with Ξ remaining a usable solution of Φ.
It follows that $\Theta'_1, x : C + \Theta'_2, x : C \leq \Xi(\Theta), x : C$, as required.

Subcase $A = !\mathbb{1}^\circ$

In this case, we have that $B_1 = !\delta_1^\circ$ and $B_2 = !\delta_2^\circ$.
and:

$$\frac{\begin{array}{c}\Theta'_1 + \Theta'_2 \blacktriangleright \Theta; \Phi \\ !\delta_1^\circ \sim !\delta_2^\circ \blacktriangleright \{\delta_1 <: \delta_2, \delta_2 <: \delta_1\} \qquad \mathsf{unr}(\delta_1) \blacktriangleright \{\delta_1 <: \mathbb{1}\} \qquad \mathsf{unr}(\delta_2) \blacktriangleright \{\delta_2 <: \mathbb{1}\}\end{array}}{\Theta'_1, !\delta_1^\circ + \Theta'_2, !\delta_2^\circ \blacktriangleright \Theta, x : !\delta_1^\circ; \Phi \cup \{\delta_1 <: \delta_2, \delta_2 <: \delta_1, \delta_1 <: \mathbb{1}, \delta_2 <: \mathbb{1}\}}$$

Let $\Xi' = \Xi[\delta_1 \mapsto \mathbb{1}, \delta_2 \mapsto \mathbb{1}]$, which is now a usable solution for the additional constraints.
Finally, we have that $\Gamma, x : !\mathbb{1}^\circ \leq \Xi'(\Theta, x : !\delta_1^\circ) \leq \Xi'(\Theta), x : !\mathbb{1}^\circ$, as required. □

As a corollary we can show the completeness of combining nullable environments:

Corollary E.46. *If:*

- $\Gamma_1 + \Gamma_2 = \Gamma$,
- $\Gamma_1 \leq \Xi_1(\Psi_1)$,
- $\Gamma_2 \leq \Xi_2(\Psi_2)$*; and*
- $pv(\Xi_1) \cap pv(\Xi_2) = \emptyset$

then there exist Θ, Φ *such that* $\Psi_1 + \Psi_2 \blacktriangleright \Theta; \Phi$*, and there exists a usable solution* $\Xi \supseteq \Xi_1 \cup \Xi_2$ *of* Φ *such that* $\Gamma \leq \Xi(\Theta)$.

Next, we need to show completeness of merging.

Lemma E.47. *If:*

- $A \leq \Xi_1(\tau_1)$,
- $A \leq \Xi_2(\tau_2)$; *and*
- $pv(\Xi_1) \cap pv(\Xi_2) = \emptyset$

then there exist τ, 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* $A \leq \Xi(\sigma)$.

Proof. By case analysis on the structure of A.

Base types follow directly, so we need instead to examine mailbox types.

Case $A = \,!E^{\eta}$

In this case, by the definition of subtyping, we have that:

- $!E^{\eta} \leq \Xi_1(!\gamma^{\eta_1})$
- $!E^{\eta} \leq \Xi_2(!\delta^{\eta_2})$

We first show that $\eta \leq \max(\eta_1, \eta_2)$. If $\eta = \circ$ then it must be the case that $\eta_1, \eta_2 = \circ$ and $\max(\eta_1, \eta_2) = \circ$. If $\eta = \bullet$ then we have that $\eta \leq \max(\eta_1, \eta_2)$.

Using algorithmic type merging, we can construct:

$$\frac{!\gamma \sqcap !\delta \blacktriangleright !(\gamma \oplus \delta); \emptyset}{!\gamma^{\eta_1} \sqcap !\delta^{\eta_2} \blacktriangleright !(\gamma \oplus \delta)^{\max(\eta_1, \eta_2)}; \emptyset}$$

Since $\mathrm{dom}(\Xi_1) \cap \mathrm{dom}(\Xi_2) = \emptyset$, we can set $\Xi = \Xi_1 \cup \Xi_2$ (which is trivially a solution of $\emptyset$).

It remains to be shown that $!E^{\eta} \leq !(\Xi(\gamma \oplus \delta))^{\max(\eta_1, \eta_2)}$.

First we note that: $\Xi(!(\gamma \oplus \delta)) = !(\Xi(\gamma) \oplus \Xi(\delta))$

Now since $!E \leq !\Xi(\gamma)$ and $!E \leq !\Xi(\delta)$, it follows by the definition of subtyping that $\Xi(\gamma) \sqsubseteq E$ and $\Xi(\delta) \sqsubseteq E$. Therefore it follows by the definition of pattern semantics that $\Xi(\gamma) \oplus \Xi(\delta) \sqsubseteq E$ and therefore that $!E^{\eta} \leq !(\Xi(\gamma \oplus \delta))^{\max(\eta_1, \eta_2)}$ as required.

Case $A = \,?E^{\eta}$

In this case, by the definition of subtyping, we have that:

- $!E^{\eta} \leq \Xi_1(!\gamma^{\eta_1})$
- $!E^{\eta} \leq \Xi_2(!\delta^{\eta_2})$

The reasoning for usage subtyping follows from the previous case, so we take for given that $\eta \leq \max(\eta_1, \eta_2)$.

Next, we construct the following derivation using algorithmic type merging:

$$\frac{\dfrac{\alpha \text{ fresh}}{?\gamma \sqcap ?\delta \blacktriangleright ?\alpha; \{\alpha <: \gamma, \alpha <: \delta\}}}{\eta_1^{\eta} ?\gamma \sqcap \eta_2^{\eta} ?\delta \blacktriangleright \max(\eta_1, \eta_2)^{\eta} ?\alpha; \{\alpha <: \gamma, \alpha <: \delta\}}$$

Since $\mathrm{dom}(\Xi_1) \cap \mathrm{dom}(\Xi_2) = \emptyset$, we can set $\Xi = \Xi_1 \cup \Xi_2 \cup \{\alpha \mapsto E\}$.

To show that Ξ is a usable solution of the constraint set, it remains to be shown that:

- $E \sqsubseteq \Xi(\gamma)$
- $E \sqsubseteq \Xi(\delta)$

Since $?E \leq ?\Xi_1(\gamma)$ it follows that $E \sqsubseteq \Xi_1(\gamma)$ and likewise for $\Xi_2(\delta)$; since $\mathrm{dom}(\Xi_1) \cap \mathrm{dom}(\Xi_2) = \emptyset$ it follows that $?E \sqsubseteq \Xi(\gamma)$ and likewise for δ, as required.

□

Lemma E.48. *If:*

- $\Gamma \preccurlyeq \Xi_1(\Theta_1)$,
- $\Gamma \preccurlyeq \Xi_2(\Theta_2)$*; and*
- $pv(\Xi_1) \cap pv(\Xi_2) = \emptyset$

then there exist Θ, Φ *such that* $\Theta_1 \sqcap \Theta_2 \blacktriangleright \Theta; \Phi$ *and there exists a usable solution* $\Xi \supseteq \Xi_1 \cup \Xi_2$ *of* Φ *such that* $\Gamma \leq \Xi(\Theta)$.

Proof. By induction on the size of Γ and inspection of Θ_1 and Θ_2, noting that due to the definition of $\preccurlyeq$, all must be of the same length; merging of types relies on Lemma E.47. □

Corollary E.49 (Completeness of merging (nullable environments)). *If:*

- $\Gamma \preccurlyeq \Xi_1(\Psi_1)$,
- $\Gamma \preccurlyeq \Xi_2(\Psi_2)$*; and*
- $pv(\Xi_1) \cap pv(\Xi_2) = \emptyset$

then there exist Θ, Φ *such that* $\Theta_1 \sqcap \Theta_2 \blacktriangleright \Psi; \Phi$ *and there exists a usable solution* $\Xi \supseteq \Xi_1 \cup \Xi_2$ *of* Φ *such that* $\Gamma \leq \Xi(\Psi)$.

Lemma E.50 (Completeness of subtyping). *If* $\Xi(\tau) \leq \Xi(\sigma)$ *then* $\tau \leq \sigma \blacktriangleright \Phi$ *where* Ξ *is a usable solution of* Φ.

Proof. By case analysis on $\Xi(\tau) \leq \Xi(\tau)$.

Case $C \leq C$

Here we can show $C \leq C \blacktriangleright \emptyset$, where Ξ is trivially a usable solution of $\emptyset$, as required.

Case $\Xi(\mathbf{!}\gamma^{\eta_1}) \leq \Xi(\mathbf{!}\delta^{\eta_2})$

Since $\Xi(\mathbf{!}\gamma^{\eta_2}) = \mathbf{!}\Xi(\gamma)^{\eta_2}$ we can assume:

$$\frac{\eta_1 \leq \eta_2 \qquad \Xi(\delta) \sqsubseteq \Xi(\gamma)}{\mathbf{!}\Xi(\gamma)^{\eta_1} \leq \mathbf{!}\Xi(\delta)^{\eta_2}}$$

Using algorithmic subtyping we can derive:

$$\frac{\eta_1 \leq \eta_2 \qquad \overline{\mathbf{!}\gamma \leq \mathbf{!}\delta \blacktriangleright \delta <: \gamma}}{\mathbf{!}\gamma^{\eta_1} \leq \mathbf{!}\delta^{\eta_2} \blacktriangleright \delta <: \gamma}$$

And since $\Xi(\gamma) \sqsubseteq E$ it follows that Ξ is a usable solution of $\delta <: \gamma$, as required.

Case $\mathbf{?}\gamma^{\eta_1} \leq \Xi(\mathbf{?}\delta^{\eta_2})$

Since $\Xi(\mathbf{?}\delta^{\eta_2}) = \mathbf{?}\Xi(\delta)^{\eta_2}$ we can assume:

$$\frac{\eta_1 \leq \eta_2 \qquad \Xi(\gamma) \sqsubseteq \Xi(\gamma)}{\mathbf{?}\Xi(\gamma)^{\eta_1} \leq \mathbf{?}\Xi(\delta)^{\eta_2}}$$

Using algorithmic subtyping we can derive:

$$\dfrac{\eta_1 \leq \eta_2 \qquad \dfrac{}{?\gamma \leq ?\delta \blacktriangleright \gamma <: \delta}}{?\gamma^{\eta_1} \leq ?\delta^{\eta_2} \blacktriangleright \gamma <: \delta}$$

And since $\Xi(\gamma) \sqsubseteq \Xi(\delta)$ it follows that Ξ is a usable solution of $\gamma <: \delta$, as required. □

Lemma E.51 (Completeness of check meta-function). *If* $\Xi(\Theta, x : \tau) \preccurlyeq \Xi(\Theta')$ *then* $check(\Theta', x, \tau) = \Phi$ *where* Ξ *is a usable solution of* Φ.

Proof. A direct consequence of Lemma E.50. □

The n-ary version of Lemma E.51 follows as a corollary:

Corollary E.52 (Completeness of n-ary check meta-function). *If* $\Xi(\Theta, \overrightarrow{x} : \overrightarrow{\tau}) \preccurlyeq \Xi(\Theta')$ *then* $check(\Theta', \overrightarrow{x}, \overrightarrow{\tau}) = \Phi$ *where* Ξ *is a usable solution of* Φ.

E.4.3 Supertype checkability. In order to show the completeness of T-Subs, we must show that if a term is checkable at a subtype, then it is also checkable at a supertype.

To do this we require several intermediate results.

We firstly define *closed* and *satisfiable* constraint sets.

Definition E.53 (Closed constraint set). A constraint set Φ is *closed* if $\mathrm{pv}(\Phi) = \emptyset$.

Definition E.54 (Satisfiable constraint set). A closed constraint set Φ is *satisfiable* if the empty solution is a solution for Φ (i.e., $\Phi = (E_i <: F_i)_i$ and $(E_i \sqsubseteq F_i)_i$).

If we have two types which do not contain pattern variables, algorithmic subtyping does not introduce any pattern variables into the constraint set.

Lemma E.55 (Subtyping introduces no fresh variables). *If* $A \leq B \blacktriangleright \Phi$, *then* $pv(\Phi) = \emptyset$.

Proof. A straightforward case analysis on the derivation of $\tau \leq \sigma \blacktriangleright \Phi$. □

Next, if we have an algorithmic subtyping judgement which produces a satisfiable constraint set, and a subtyping relation with a supertype, then we can show that the algorithmic subtyping judgement instantiated with the supertype will produce a satisfiable constraint set.

Lemma E.56 (Widening 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 $A \leq A' \blacktriangleright \Phi$.

Base types hold trivially, so we need only consider two cases:

Case $!E \leq !F$

Assumption:

$$\dfrac{\eta_1 \leq \eta_2}{!E^{\eta_1} \leq !F^{\eta_2} \blacktriangleright \{F <: E\}}$$

also we know that $F <: E$ is satisfiable (therefore that $F \sqsubseteq E$), and $!F \leq B$.

By the definition of subtyping we have that $B = !F'$ for some pattern F', and therefore that $F' \sqsubseteq F$.

By transitivity we have that

$F' \sqsubseteq F \sqsubseteq E$ and therefore

$$\frac{\eta_1 \leq \eta_2}{!E^{\eta_1} \leq\, !F'^{\eta_2} \blacktriangleright \{F' <: E\}}$$

where $F' <: E$ is satisfiable, as required.

Case $?E \leq ?F$

Assumption:

$$\frac{\eta_1 \leq \eta_2}{?E^{\eta_1} \leq ?F^{\eta_2} \blacktriangleright \{E <: F\}}$$

also we know that $E <: F$ is satisfiable (therefore that $E \sqsubseteq F$), and $?F \leq B$.

By the definition of subtyping we have that $B = ?F'$ for some pattern F' and therefore that $F \sqsubseteq F'$.

Thus by transitivity we have that $E \sqsubseteq F \sqsubseteq F'$ and therefore that:

$$\frac{\eta_1 \leq \eta_2}{?E^{\eta_1} \leq ?F'^{\eta_2} \blacktriangleright \{E <: F'\}}$$

where $E <: F'$ is satisfiable, as required.

□

We also need to show that environment joining respects subtyping, which we do by firstly showing that type joining respects subtyping.

Lemma E.57 (Algorithmic type join respects subtyping). *If:*

- $\tau_1 \fatsemi \tau_2 \blacktriangleright \Theta; \Phi$
- *Ξ is some usable solution of Φ such that $\Xi(\tau_2) \leq \Xi(\tau_3)$ for some τ_3*

then $\tau_1 \fatsemi \tau_3 \blacktriangleright \tau'; \Phi'$ for some τ', Φ' such that $\Xi(\tau) \leq \Xi(\tau')$ and Ξ is a usable solution of $\Xi(\Phi')$.

Proof. Base type combination follows straightforwardly, so we have:

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

so it suffices to proceed by case analysis on the derivation of $\varsigma_1 \fatsemi \varsigma_2 \blacktriangleright \varsigma; \Phi$.

Case $\varsigma_1 = !\gamma$ and $\varsigma_2 = !\delta$

Assumption:

$$\frac{}{!\gamma \fatsemi !\delta \blacktriangleright !(\gamma \odot \delta); \emptyset}$$

We also assume that $\Xi(!\delta) \leq \Xi(\tau)$ for some τ, which by the definition of subtyping means that $\tau = !\delta'$ for some δ', where $\Xi(\delta') \sqsubseteq \Xi(\delta)$.

It follows by the compositionality of pattern semantics that $\gamma \odot \delta' \sqsubseteq \gamma \odot \delta$ and thus $!(\gamma \odot \delta) \leq !(\gamma \odot \delta')$, and we have that

$$\frac{}{!\gamma \fatsemi !\delta' \blacktriangleright !(\gamma \odot \delta'); \emptyset}$$

as required.

Case $\varsigma_1 = !\gamma$ and $\varsigma_2 = ?\delta$

Assumption:

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

By the assumptions we know that Ξ is a usable solution of Φ such that $\Xi(?\delta) \leq \Xi(\tau)$ for some τ. By the definition of subtyping it must be the case that $\tau = ?\delta'$ for some pattern δ'.

Since $\Xi(?\delta) \leq \Xi(?\delta')$ it follows that $\Xi(\delta) \sqsubseteq \Xi(\delta')$.

Since Ξ is a usable solution of Φ we have that $\Xi(\gamma \odot \alpha) \sqsubseteq \Xi(\delta)$.

Therefore by transitivity of subtyping we have that $\Xi(\gamma \odot \alpha) \sqsubseteq \Xi(\delta')$ and thus know that Ξ is a usable solution of $\{(\gamma \odot \alpha) <: \delta'\}$.

Recomposing:

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

Case $\varsigma_1 = ?\gamma$ and $\varsigma_2 = !\delta$

Similar to the previous case.

□

The desired result falls out as a corollary:

Corollary E.58 (Algorithmic environment join respects subtyping). *If:*

- $\Theta_1 \fatsemi \Theta_2 \blacktriangleright \Theta; \Phi$
- Ξ *is some usable solution of* Φ *such that* $\Xi(\Theta_1) \leq \Xi(\Theta_3)$ *for some* Θ_3

then $\Theta_1 \fatsemi \Theta_3 \blacktriangleright \Theta'; \Phi'$ *for some* Θ', Φ' *such that* $\Xi(\Theta) \leq \Xi(\Theta')$ *and* Ξ *is a usable solution of* $\Xi(\Phi')$.

Finally we want to see that algorithmic environment combination respects subtyping.

Lemma E.59 (Algorithmic combination respects subtyping). *If:*

- $\Theta_1 + \Theta_2 \blacktriangleright \Theta; \Phi$
- Ξ *is some usable solution of* Φ *such that* $\Xi(\Theta_1) \leq \Xi(\Theta_3)$ *for some* Θ_3

then $\Theta_1 + \Theta_3 \blacktriangleright \Theta'; \Phi'$ *for some* Θ', Φ' *such that* $\Xi(\Theta) \leq \Xi(\Theta')$ *and* Ξ *is a usable solution of* $\Xi(\Phi')$.

Proof. By induction on the derivation of $\Theta_1 + \Theta_2 \blacktriangleright \Theta; \Phi$. □

Corollary E.60 (Algorithmic combination respects subtyping (nullable environments)). *If:*

- $\Psi_1 + \Psi_2 \blacktriangleright \Psi; \Phi$
- Ξ *is some usable solution of* Φ *such that* $\Xi(\Psi_1) \leq \Xi(\Psi_3)$ *for some* Ψ_3

then $\Psi_1 + \Psi_3 \blacktriangleright \Psi'; \Phi'$ *for some* Ψ', Φ' *such that* $\Xi(\Psi) \leq \Xi(\Psi')$ *and* Ξ *is a usable solution of* $\Xi(\Phi')$.

Relying on the previous results, we can now show the supertype checkability lemma.

Lemma E.61 (Supertype checkability).
Suppose $\mathcal{P}$ is closed.

- *If:*
  - $M \Leftarrow_{\mathcal{P}} A \blacktriangleright \Theta; \Phi$
  - Ξ *is a usable solution of* Φ
  - $A \leq B$

  then $M \Leftarrow_{\mathcal{P}} B \blacktriangleright \Theta'; \Phi'$, *where* Ξ *is a usable solution of* Φ' *and* $\Xi(\Theta) \preccurlyeq \Xi(\Theta')$.
- *If:*
  - $\{E\}\ \overrightarrow{G} \Leftarrow_{\mathcal{P}} A \blacktriangleright \Theta; \Phi; F$
  - Ξ *is a usable solution of* Φ
  - $A \leq B$

  then $\{E\}\ \overrightarrow{G} \Leftarrow_{\mathcal{P}} B \blacktriangleright \Theta'; \Phi'; F$ *where* Ξ *is a usable solution of* Φ' *and* $\Xi(\Theta) \preccurlyeq \Xi(\Theta')$.
- *If:*
  - $\{E\}\ G \Leftarrow_{\mathcal{P}} A \blacktriangleright \Theta; \Phi; F$
  - Ξ *is a usable solution of* Φ
  - $A \leq B$

  then $\{E\}\ G \Leftarrow_{\mathcal{P}} B \blacktriangleright \Theta'; \Phi'; F$ *where* Ξ *is a usable solution of* Φ' *and* $\Xi(\Theta) \preccurlyeq \Xi(\Theta')$.

Proof. By mutual induction on the three premises. We concentrate on proving premise 1 in detail, and TCG-Recv for premise 3; premise 2 follows from premise 3, and the remaining guard cases are straightforward.

By induction on the derivation of $M \Leftarrow A \blacktriangleright \Theta; \Phi$.

Case TC-Var

Assumption:

$$\frac{}{x \Leftarrow A \blacktriangleright x : A; \emptyset}$$

Now given that we have $A \leq B$, we can construct:

$$\frac{}{x \Leftarrow B \blacktriangleright x : B; \emptyset}$$

As $\Phi = \cdot$ it straightforwardly follows that Ξ is a usable solution, and since $A \leq B$ we have that $x : A \leq x : B$ as required.

Case TC-Let

Assumption:

$$\frac{\begin{array}{cc} M \Leftarrow \lfloor T \rfloor \blacktriangleright \Theta_1; \Phi_1 & N \Leftarrow A \blacktriangleright \Theta_2; \Phi_2 \\ \mathsf{check}(\Theta_2, x, \lfloor T \rfloor) = \Phi_3 & \Theta_1 \mathbin{⨟} \Theta_2 \blacktriangleright \Theta; \Phi_4 \end{array}}{\mathbf{let}\ x{:}\ T = M\ \mathbf{in}\ N \Leftarrow A \blacktriangleright \Theta; \Phi_1 \cup \cdots \cup \Phi_4}$$

By the IH we have that:

- $N \Leftarrow B \blacktriangleright \Theta_3; \Phi_5$ for some Θ_3, Φ_5
- $\Xi(\Theta_2) \leq \Xi(\Theta_3)$
- Ξ is a usable solution of Θ_3

By Corollary E.58 we have that $\Theta_1 \fatsemi \Theta_3 \blacktriangleright \Theta'; \Phi_6$, where $\Xi(\Theta) \leq \Xi(\Theta')$ and Ξ is a usable solution of Φ_6.

By Lemma E.51, we have that $\mathsf{check}(\Theta_3, x, \lfloor T \rfloor) = \Phi_5$ where Ξ is a usable solution of Φ_5.

Therefore we can show that:

$$\frac{\begin{array}{cc} M \Leftarrow \lfloor T \rfloor \blacktriangleright \Theta_1;\ \Phi_1 & N \Leftarrow B \blacktriangleright \Theta_3;\ \Phi_4 \\ \mathsf{check}(\Theta_3, x, \lfloor T \rfloor) = \Phi_5 & \Theta_1 \fatsemi \Theta_3 \blacktriangleright \Theta'; \Phi_6 \end{array}}{\mathbf{let}\ x\colon T = M\ \mathbf{in}\ N \Leftarrow B \blacktriangleright \Theta';\ \Phi_1 \cup \Phi_4 \cup \Phi_5 \cup \Phi_6}$$

as required.

Case TC-Guard

$$\frac{\begin{array}{c} \{E\}\ \overrightarrow{G} \Leftarrow A \blacktriangleright \Psi;\ \Phi_1;\ F \\ V \Leftarrow {?F^\bullet} \blacktriangleright \Theta';\ \Phi_2 \qquad \Psi + \Theta' \blacktriangleright \Theta; \Phi_3 \end{array}}{\mathbf{guard}\ V : E\ \{\overrightarrow{G}\} \Leftarrow A \blacktriangleright \Theta;\ \Phi_1 \cup \Phi_2 \cup \Phi_3}$$

By the IH:

- $\{E\}\ \overrightarrow{G} \Leftarrow B \blacktriangleright \Psi';\ \Phi_1';\ F$ with Ξ a usable solution of Φ_1' and $\Xi(E) \sqsubseteq \Xi(F)$ and $\Xi(\Psi) \leq \Xi(\Psi')$
- $V \Leftarrow {?F^\bullet} \blacktriangleright \Theta'';\ \Phi_2'$ with Ξ a usable solution of Φ_2' and $\Xi(\Theta) \leq \Xi(\Theta'')$

By Corollary E.60 $\Psi' + \Theta'' \blacktriangleright \Theta'''; \Phi_3'$.

Recomposing:

$$\frac{\begin{array}{c} \{E\}\ \overrightarrow{G} \Leftarrow B \blacktriangleright \Psi';\ \Phi_1';\ F \\ V \Leftarrow {?F^\bullet} \blacktriangleright \Theta'';\ \Phi_2' \qquad \Psi' + \Theta'' \blacktriangleright \Theta'''; \Phi_3' \end{array}}{\mathbf{guard}\ V : E\ \{\overrightarrow{G}\} \Leftarrow B \blacktriangleright \Theta''';\ \Phi_1' \cup \Phi_2' \cup \Phi_3'}$$

with Ξ a usable solution of $\Phi_1' \cup \Phi_2' \cup \Phi_3'$ and $\Xi(\Theta) \leq \Xi(\Theta''')$ as required.

Case TC-Sub

Assumptions:

$$\frac{M \Rightarrow A \blacktriangleright \Theta;\ \Phi_1 \qquad A \leq A' \blacktriangleright \Phi_2}{M \Leftarrow A' \blacktriangleright \Theta;\ \Phi_1 \cup \Phi_2}$$

and:

- Ξ is a usable solution of $\Phi_1 \cup \Phi_2$
- $A' \leq B$

Since A, A', and B contain no pattern variables, by Lemma E.55 we have that $\mathsf{pv}(\Phi_2) = \emptyset$ (however, since Ξ is a usable solution of $\Phi_1 \cup \Phi_2$, it follows that Φ_2 is satisfiable).

By Lemma E.56, we have that $A \leq B \blacktriangleright \Phi_3$, where Φ_3 is satisfiable.

Since Φ_3 is satisfiable and (again by Lemma E.55) $\mathsf{pv}(\Phi_3) = \emptyset$, it follows that Ξ is a usable solution of $\Phi_1 \cup \Phi_3$.

Thus by TC-Sub we have that:

$$\frac{M \Rightarrow A \blacktriangleright \Theta;\ \Phi_1 \qquad A \leq B \blacktriangleright \Phi_3}{M \Leftarrow B' \blacktriangleright \Theta;\ \Phi_1 \cup \Phi_3}$$

where Ξ is a usable solution of $\Phi_1 \cup \Phi_3$, as required.

Case TCG-Recv

Assumption:

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

Also we have that:

- Ξ is a usable solution of $\Phi_1 \cup \Phi_2 \cup \{E \,/\, \mathtt{m} <: \gamma\}$
- $\Xi(\tau) \leq \Xi(\sigma)$

By the IH we have that $M \Leftarrow B \blacktriangleright \Theta'', y : \delta^{\bullet};\ \Phi'_1$
where $\Xi(\Theta', y : ?\gamma^{\bullet}) \preccurlyeq \Xi(\Theta'', y : ?\delta^{\bullet})$ and where Ξ is a usable solution of Φ'_1.
By the definition of strict environment subtyping we have that $?\gamma \leq ?\delta$ and therefore $\gamma \sqsubseteq \delta$.
Let $\Theta''' = \Theta'' - \overrightarrow{x}$. It follows by the definition of environment subtyping that $\Xi(\Theta) \preccurlyeq \Xi(\Theta''')$.
Due to the definition of the subtyping relation it remains the case that $\mathsf{base}(\overrightarrow{T}) \vee \mathsf{base}(\Theta''')$.
By Lemma E.51 we have that $\mathsf{check}(\Theta'', \overrightarrow{x}, \lceil \overrightarrow{T} \rceil) = \Phi'_2$ where Ξ is a usable solution of Φ'_2.
Recomposing:

$$\frac{\mathcal{P}(\mathtt{m}) = \overrightarrow{T} \qquad \Theta''' = \Theta'' - \overrightarrow{x} \qquad \mathsf{base}(\overrightarrow{T}) \vee \mathsf{base}(\Theta) \qquad \begin{array}{c} M \Leftarrow \tau \blacktriangleright \Theta'', y : ?\delta^{\bullet};\ \Phi'_1 \\ \mathsf{check}(\Theta', \overrightarrow{x}, \overrightarrow{\lceil T \rceil}) = \Phi'_2 \end{array}}{\{E\}\ \mathbf{receive}\ \mathtt{m}[\overrightarrow{x}]\ \mathbf{from}\ y \mapsto M \Leftarrow B \blacktriangleright \Theta;\ \Phi'_1 \cup \Phi'_2 \cup \{E \,/\, \mathtt{m} <: \delta\};\ \mathtt{m} \odot \delta}$$

where $\Xi(\mathtt{m} \odot \gamma) \sqsubseteq \Xi(\mathtt{m} \odot \delta)$ and Ξ is a usable solution of $\Phi'_1 \cup \Phi'_2 \cup \{E \,/\, \mathtt{m} <: \delta\}$ and $\Xi(\Theta''') \preccurlyeq \Xi(\Theta)$, as required. □

The following specific result, used within the completeness result, is a corollary.

Corollary E.62 (Supertype checkability). *If:*

- $M \Leftarrow_{\mathcal{P}} A \blacktriangleright \Theta;\ \Phi$
- *$\mathcal{P}$ is closed*
- *Ξ is a usable solution of Φ*
- $A \leq B$

then there exist Θ', Φ' such that $M \Leftarrow_{\mathcal{P}} B \blacktriangleright \Theta';\ \Phi'$ where Ξ is a usable solution of Φ' and $\Xi'(\Theta) \preccurlyeq \Xi'(\Theta')$.

E.4.4 *Freshness of type variables.*

It is convenient to reason about fresh variables.

Definition E.63 (Created fresh). A pattern variable α is *created fresh* in a derivation $\mathbf{D}$ if there exists some subderivation $\mathbf{D}'$ of $\mathbf{D}$ which is of the form:

$$\frac{\alpha\ \text{fresh}}{\mathbf{D}'}$$

Lemma E.64 (Pattern variable freshness). *If* $\mathbf{D} = M \Leftarrow_{\mathcal{P}} A \blacktriangleright \Theta; \Phi$ *or* $\mathbf{D} = M \Rightarrow_{\mathcal{P}} A \blacktriangleright \Theta; \Phi$ *where* $\mathcal{P}$ *is closed, then all pattern variables in* $pv(\Theta) \cup pv(\Phi)$ *are created fresh in* $\mathbf{D}$.

Proof. By induction on the respective derivation, noting that since the signature and types are closed, pattern variables are only introduced through the type join and type merge operators, where they are created fresh. □

E.4.5 Completeness proof. Finally, we can tie the above results together to show algorithmic completeness.

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

Proof. A direct consequence of Lemma E.65. □

Lemma E.65 (Algorithmic Completeness (Generalised)).

- *If* $\Gamma \vdash_{\mathcal{P}} M : A$ *where* $\mathcal{P}$ *is closed, then there exist some* Θ, Φ *and usable solution* Ξ *of* Φ *such that* $M \Leftarrow A \blacktriangleright \Theta; \Phi$ *where* $dom(\Xi) = pv(\Theta) \cup pv(\Phi)$ *and* $\Gamma \preccurlyeq \Xi(\Theta)$.
- *If* $\Gamma \vdash_{\mathcal{P}} \overrightarrow{G} : A :: F$ *where* $E \vDash_{lit} F$ *for some pattern* E *and* $\mathcal{P}$ *is closed, then there exist some* Θ, Φ *and usable solution* Ξ *of* Φ *such that* $\{E\}\ \overrightarrow{G} \Leftarrow A \blacktriangleright \Theta; \Phi; F$ *where* $dom(\Xi) = pv(\Theta) \cup pv(\Phi)$ *and* $\Gamma \preccurlyeq \Xi(\Theta)$.
- *If* $\Gamma \vdash_{\mathcal{P}} G : A :: F$ *where* $E \vDash_{lit} F$ *for some pattern* E *and* $\mathcal{P}$ *is closed, then there exist some* Θ, Φ *and usable solution* Ξ *of* Φ *such that* $\{E\}\ G \Leftarrow A \blacktriangleright \Theta; \Phi; F$ *where* $dom(\Xi) = pv(\Theta) \cup pv(\Phi)$ *and* $\Gamma \preccurlyeq \Xi(\Theta)$.

Proof. By mutual induction on both premises.

Premise 1:

Case T-Var

Assumption:

$$\frac{}{x : A \vdash x : A}$$

Recomposing via TC-Var:

$$\frac{}{x \Leftarrow A \blacktriangleright x : A; \emptyset}$$

with $\Xi = \cdot$.

Case T-Const

Assumption:

$$\frac{c \text{ has base type } C}{\cdot \vdash c : C}$$

By TS-Const:

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

By Lemma E.42 we have that $c \Leftarrow C \blacktriangleright \cdot; \emptyset$ with $\Xi = \cdot$, as required.

Case T-App

Assumption:

$$\frac{\mathcal{P}(f) = \overrightarrow{A} \to B \qquad (\Gamma_i \vdash V_i : A_i)_{i \in 1..n}}{\Gamma_1 + \cdots + \Gamma_n \vdash f(\overrightarrow{V}) : B}$$

By (repeated) use of Lemma E.41, we have that there exist some Γ'_i such that $\Gamma_i \leq \Gamma'_i$ and $V_i \Leftarrow A_i \blacktriangleright \Gamma'_i; \emptyset$ for $i \in 1..n$.

By repeated use of Lemma E.45, we have that there $\Gamma'_1 + \ldots + \Gamma'_n \blacktriangleright \Theta; \Phi$ for some Θ, Φ and that there exists some usable solution Ξ of $\Gamma \leq \Xi(\Theta)$.

Thus by TS-App we can show

$$\frac{\mathcal{P}(f) = \overrightarrow{A} \to B \qquad (V_i \Leftarrow A_i \blacktriangleright \Gamma'_i; \emptyset)_{i \in 1..n} \qquad \Gamma'_1 + \ldots + \Gamma'_n \blacktriangleright \Theta; \Phi}{f(\overrightarrow{V}) \Rightarrow B \blacktriangleright \Theta; \Phi}$$

and by Lemma E.42 we have that $f(\overrightarrow{V}) \Leftarrow B \blacktriangleright \Theta; \Phi$ as required.

Case T-Let

Assumption:

$$\frac{\Gamma_1 \vdash M : \lfloor T \rfloor \qquad \Gamma_2, x : \lfloor T \rfloor \vdash N : B}{\Gamma_1 \triangleright \Gamma_2 \vdash \mathbf{let}\ x\colon T = M\ \mathbf{in}\ N : B}$$

By the IH we have that:

- There exist some Θ_1, Φ_1 and usable solution Ξ_1 of Φ_1 such that $M \Leftarrow A \blacktriangleright \Theta_1; \Phi_1$ where $\Gamma_1 \preccurlyeq \Xi_1(\Theta_1)$
- There exist some Θ_2, Φ_2 and usable solution Ξ_2 of Φ_2 such that $N \Leftarrow B \blacktriangleright \Theta_2, x : \lfloor T' \rfloor; \Phi_2$ where $\Gamma_2, x : \lfloor T \rfloor \preccurlyeq \Xi_2(\Theta_2)$

By Lemma E.44, we have that $\Theta_1 \fatsemi \Theta_2 \blacktriangleright \Theta; \Phi_3$ and a usable solution $\Xi \supseteq \Xi_1 \cup \Xi_2$ of Φ_3 such that $\Gamma_1 \triangleright \Gamma_2 \preccurlyeq \Xi(\Theta)$.

By Lemma E.51, we have that $\text{check}(\Theta_2, x, \lfloor T \rfloor) = \Phi_4$ and Ξ is a usable solution of Φ_4.

Since $\Xi \supseteq \Xi_1 \cup \Xi_2$ and pattern variables in these subderivations are only introduced fresh (Lemma E.64), we have that Ξ is also a usable solution of Φ_1 and Φ_2.

Therefore, we have that Ξ is a usable solution of $\Phi_1 \cup \Phi_2 \cup \Phi_3 \cup \Phi_4$.

Recomposing using TC-Let:

$$\frac{\begin{array}{cc} M \Leftarrow \lfloor T \rfloor \blacktriangleright \Theta_1; \Phi_1 & N \Leftarrow A \blacktriangleright \Theta_2; \Phi_2 \\ \text{check}(\Theta_2, x, \lfloor T \rfloor) = \Phi_4 & \Theta_1 \fatsemi \Theta_2 \blacktriangleright \Theta; \Phi_3 \end{array}}{\mathbf{let}\ x\colon T = M\ \mathbf{in}\ N \Leftarrow A \blacktriangleright \Theta; \Phi_1 \cup \cdots \cup \Phi_4}$$

where $\Xi(\Theta) \preccurlyeq \Gamma_1 \triangleright \Gamma_2$ and Ξ is a usable solution of $\Phi_1 \cup \Phi_2 \cup \Phi_3 \cup \Phi_4$, as required.

Case T-Spawn

$$\frac{\Gamma \vdash M : \mathbf{1}}{\lceil \Gamma \rceil \vdash \mathbf{spawn}\ M : \mathbf{1}}$$

By the IH $M \Leftarrow \mathbf{1} \blacktriangleright \Theta;\ \Phi$ for some Θ, Φ, and a usable solution Ξ such that $\Gamma \leq \Xi(\Phi)$. Thus by TS-Spawn:

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

where $\Gamma \leq \Xi(\Theta)$ and therefore $\lceil \Gamma \rceil \leq \lceil \Xi(\Theta) \rceil$.

Finally, by Lemma E.42, we have that $\mathbf{spawn}\ M \Leftarrow \mathbf{1} \blacktriangleright \lceil \Theta \rceil;\ \Phi$ with usable solution Ξ of Φ as required.

Case T-New

Assumption:

$$\frac{}{\cdot \vdash \mathbf{new} : ?\mathbb{1}^{\bullet}}$$

By TS-New we have that $\mathbf{new} \Rightarrow ?\mathbb{1}^{\bullet} \blacktriangleright \cdot;\ \emptyset$ and by Lemma E.42 it follows that $\mathbf{new} \Leftarrow ?\mathbb{1}^{\bullet} \blacktriangleright \cdot;\ \emptyset$; we can set solution $\Xi = \cdot$, as required.

Case T-Send

Assumption:

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

By the IH we have that:

- $V \Leftarrow !\mathtt{m}^{\circ} \blacktriangleright \Theta_{target};\ \Phi_{target}$ for some Θ_{target}, Φ_{target} and some usable solution Ξ_{target} of Φ_{target} such that $\Gamma_{target} \leq \Xi(\Theta_{target})$.
- $(W_i \Leftarrow \lceil T_i \rceil \blacktriangleright \Theta_i;\ \Phi_i)_{i \in 1..n}$ for Θ_i, Φ_i and usable solutions Ξ_i of Φ_i such that $\Gamma'_i \leq \Xi_i(\Theta_i)$

By repeated use of Lemma E.45 we have that $\Theta_{target} + \Theta_1 + \ldots + \Theta_n \blacktriangleright \Theta; \Phi_{env}$, with some usable solution Ξ_{env} of Φ_{env} such that $\Gamma + \Gamma'_1 + \ldots + \Gamma'_n \leq \Xi_{env}(\Theta)$.

Since pattern variables are always chosen fresh (Lemma E.64) we have that $\Xi_{target} \cup \Xi_{env} \cup \bigcup_{i \in 1..n} \Xi'_i$ is a solution of $\Phi_{target} \cup \Phi_{env} \cup \bigcup_{i \in 1..n} \Phi'_i$.

Thus we can show by TS-Send and Lemma E.42:

$$\dfrac{\dfrac{\begin{array}{cc} \mathcal{P}(\mathtt{m}) = \overrightarrow{T} & V \Leftarrow !\mathtt{m}^{\circ} \blacktriangleright \Theta_{target};\ \Phi_{target} \\ (W_i \Leftarrow \lceil T_i \rceil \blacktriangleright \Theta_i;\ \Phi_i)_{i \in 1..n} & \Theta_{target} + \Theta_1 + \ldots + \Theta_n \blacktriangleright \Theta; \Phi_{env} \end{array}}{V\,!\,\mathtt{m}[\overrightarrow{W}] \Rightarrow \mathbf{1} \blacktriangleright \Theta;\ \Phi_{target} \cup \Phi_1 \cup \cdots \cup \Phi_n \cup \Phi_{env}}}{V\,!\,\mathtt{m}[\overrightarrow{W}] \Leftarrow \mathbf{1} \blacktriangleright \Theta;\ \Phi_{target} \cup \Phi_1 \cup \cdots \cup \Phi_n \cup \Phi_{env}}$$

where Ξ is a usable solution of $\Phi_{target} \cup \Phi_1 \cup \cdots \cup \Phi_n \cup \Phi_{env}$ and $\Gamma \leq \Xi(\Theta)$, as required.

Case T-Guard

Assumption:

$$\frac{\Gamma_1 \vdash V : ?F^{\bullet} \qquad \Gamma_2 \vdash \overrightarrow{G} : A :: F \qquad E \sqsubseteq F \qquad \models F}{\Gamma_1 + \Gamma_2 \vdash \mathbf{guard}\ V : E\ \{\overrightarrow{G}\} : A}$$

By Lemma E.41 we have that $V \Leftarrow ?F^{\bullet} \blacktriangleright \Gamma_1'; \emptyset$ where $\Gamma_1 \leq \Gamma_1'$.

By the IH we have that $\{E\}\ \overrightarrow{G} \Leftarrow A \blacktriangleright \Theta; \Phi; F$ where Ξ is a usable solution of Φ and $\Gamma_2 \leq \Psi$.

By Corollary E.46 we have that $\Theta' + \Psi \blacktriangleright \Theta; \Phi_2$ with $\Gamma_1 + \Gamma_2 \leq \Xi(\Theta)$ and where Ξ is a solution of Φ_2.

Recomposing:

$$\frac{V \Leftarrow ?F \blacktriangleright \Theta'; \emptyset \qquad \begin{array}{c}\{E\}\ \overrightarrow{G} \Leftarrow A \blacktriangleright \Psi; \emptyset; F \\ \Theta' + \Psi \blacktriangleright \Theta; \Phi\end{array}}{\mathbf{guard}\ V : E\ \{\overrightarrow{G}\} \Leftarrow A \blacktriangleright \Theta; \Phi \cup \{E <: F\}}$$

where Ξ is a usable solution of $\Phi_1 \cup \Phi_2$ and $E \sqsubseteq \Xi(\gamma)$, as required.

Case T-Subs

$$\frac{\Gamma \leq \Gamma' \qquad A \leq B \qquad \Gamma' \vdash M : A}{\Gamma \vdash M : B}$$

By the IH, we have that there exist Θ, Φ and some usable solution Ξ of Φ such that $\Gamma' \leq \Xi(\Theta)$ and $M \Leftarrow A \blacktriangleright \Theta; \Phi$.

By Lemma E.62 we have that $M \Leftarrow B \blacktriangleright \Theta'; \Phi'$ where Ξ is a usable solution of Φ' and $\Xi(\Theta) \leq \Xi(\Theta')$.

Recalling that $\Gamma \leq \Gamma'$, and $\Gamma' \leq \Xi(\Theta)$, and noting that $\Xi(\Theta) \leq \Xi(\Theta')$ and that $\Xi(\Theta) \leq \Xi(\Theta')$, by the transitivity of subtyping we have that $\Gamma \leq \Xi(\Theta')$.

Therefore we have that:

- $M \Leftarrow B \blacktriangleright \Theta'; \Phi'$
- Ξ is a usable solution of Φ'
- $\Gamma \leq \Xi(\Theta')$

as required.

Premise 2:

Case TG-GuardSeq

$$\frac{(\Gamma \vdash G_i : A :: E_i)_{i \in I}}{\Gamma \vdash \overrightarrow{G} : A :: E_1 \oplus \ldots \oplus E_n}$$

By repeated use of the IH (2) we have that $\{E_i\}\ G_i \Leftarrow A \blacktriangleright \Psi_i; \Phi_i; \gamma_i$ for some Ψ_i, Ξ_i, γ_i such that $\Gamma \leq \Xi_i(\Psi_i)$ and $E \sqsubseteq \Xi_i(\gamma_i)$ for each $i \in I$.

By Corollary E.49 we have that $\Psi_1 \sqcap \ldots \sqcap \Psi_n \blacktriangleright \Psi_{env}; \Phi_{env}$ and some solution Ξ_{env} of Φ_{env} such that $\Gamma \leq (\Xi_1 \cup \cdots \cup \Xi_n \cup \Xi_{env})(\Psi_{env})$.

Recomposing by TCG-Guards:

$$\frac{(\{E_i\}\ G_i \Leftarrow A \blacktriangleright \Psi_i; \Phi_i; \gamma_i)_{i \in 1..n} \qquad \gamma = \gamma_1 \oplus \cdots \oplus \gamma_n \qquad \Psi_1 \sqcap \ldots \sqcap \Psi_n \blacktriangleright \Psi_{env}; \Phi_{env}}{\{E\}\ \overrightarrow{G} \Leftarrow \Psi_{env} \blacktriangleright \Phi_{env} \cup \Phi_1 \cup \cdots \cup \Phi_n; \gamma;}$$

Since pattern variables are generated fresh, we have that the pattern variables for each Ξ_i are disjoint. Therefore, we have that:

- $\Xi = \bigcup_{i \in 1..n} \Xi_i \cup \Xi_{env}$ is a usable solution of $\Phi = \bigcup_{i \in 1..n} \Phi_n \cup \Phi_{env}$
- $\Gamma \leq \Xi(\Psi_{env})$
- $E \sqsubseteq \Xi(\gamma_1 \oplus \cdots \oplus \gamma_n)$

as required.

Premise 3:

Case TG-Fail

Assumption:

$$\frac{}{\Gamma \vdash \textbf{fail} : A :: \mathbb{0}}$$

By TCG-Fail:

$$\frac{}{\{\mathbb{0}\}\ \textbf{fail} \Leftarrow A \blacktriangleright \top;\ \emptyset;\ \mathbb{0}}$$

Where $\mathbb{0} \sqsubseteq \mathbb{0}$ and $\Gamma \leq \top$ as required.

Case TG-Free

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \textbf{free} \mapsto M : A :: \mathbb{1}}$$

By the IH (1) we have that there exist Θ, Φ and usable solution Ξ of Φ such that $M \Leftarrow A \blacktriangleright \Theta; \Phi$ with $\Gamma \leq \Xi(\Theta)$.

Recomposing by TCG-Free:

$$\frac{M \Leftarrow A \blacktriangleright \Theta;\ \Phi}{\{\mathbb{1}\}\ \textbf{free} \mapsto M \Leftarrow A \blacktriangleright \Theta;\ \Phi;\ \mathbb{1}}$$

with $\Gamma \leq \Xi(\Theta)$ and $\mathbb{1} \sqsubseteq \mathbb{1}$ as required.

Case TG-Recv

Assumption:

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

We also assume that there is some E such that $E \vDash_{\mathrm{lit}} \mathtt{m} \odot F$.

Let $\Gamma' = \Gamma, y : ?F^{\bullet}, \overrightarrow{x} : \overrightarrow{\lceil T \rceil}$.

By the IH we have that there exist Θ, Φ and usable solution Ξ of Φ s.t. $M \Leftarrow A \blacktriangleright \Theta, y : ?\gamma^{\bullet}; \Phi$ where $\mathrm{dom}(\Xi) = \mathrm{pv}(\Theta) \cup \mathrm{pv}(\Phi)$ and $\Gamma' \preccurlyeq \Xi(\Theta, y : ?\gamma^{\bullet})$.

We next need to show that if $\mathrm{base}(\overrightarrow{T}) \vee \mathrm{base}(\Gamma)$ implies that $\mathrm{base}(\overrightarrow{T}) \vee \mathrm{base}(\Theta)$. It suffices to show that $\mathrm{base}(\Gamma)$ implies $\mathrm{base}(\Theta)$. Since $\Gamma \preccurlyeq \Xi(\Theta)$, by the definition of strict environment subtyping it follows that if $\mathrm{base}(\Gamma)$ and $\Gamma \preccurlyeq \Xi(\Theta)$, then $\Gamma = \Theta$.

Next, since $\Gamma, y : ?F^{\bullet}, \overrightarrow{x} : \lceil \overrightarrow{T} \rceil \preccurlyeq \Xi(\Theta'), y : ?\gamma^{\bullet}$ it follows that $\Gamma, \overrightarrow{x} : \lceil \overrightarrow{T} \rceil \preccurlyeq \Xi(\Theta')$ and thus by Corollary E.52 we have that $\mathrm{check}(\Theta, \overrightarrow{x}, \overrightarrow{\lceil T \rceil}) = \Phi_2$ where Ξ is a usable solution of Φ_2.

Next, since $\Gamma, y : ?F^{\bullet} \preccurlyeq \Xi(\Theta), y : ?\gamma^{\bullet}$ it follows by the definition of subtyping that $F \sqsubseteq \Xi(\gamma)$.

We have one final proof obligation: showing that Ξ solves $(E \mathbin{/} \mathtt{m}) <: \gamma$.

Since $E \vDash_{\mathrm{lit}} \mathtt{m} \odot F$ we have that $F \simeq E \mathbin{/} \mathtt{m}$ and therefore both $F \sqsubseteq (E \mathbin{/} \mathtt{m})$ and $(E \mathbin{/} \mathtt{m}) \sqsubseteq F$.

Since $?F \leq ?\Xi(\gamma)$ we have that $F \sqsubseteq \Xi(\gamma)$. Thus by transitivity we have that $E \mathbin{/} \mathtt{m} \sqsubseteq F \sqsubseteq \Xi(\gamma)$ and therefore that Ξ solves $(E \mathbin{/} \mathtt{m}) <: \gamma$ as necessary.

Thus, recomposing, we have:

$$\frac{\begin{array}{c} M \Leftarrow B \blacktriangleright \Theta', y : ?\gamma^{\bullet}; \Phi_1 \\ \mathcal{P}(\mathtt{m}) = \overrightarrow{T} \qquad \Theta = \Theta' - \overrightarrow{x} \qquad \mathrm{base}(\overrightarrow{T}) \vee \mathrm{base}(\Theta) \qquad \mathrm{check}(\Theta', \overrightarrow{x}, \overrightarrow{\lceil T \rceil}) = \Phi_2 \end{array}}{\{E\}\ \textbf{receive}\ \mathtt{m}[\overrightarrow{x}]\ \textbf{from}\ y \mapsto M \Leftarrow B \blacktriangleright \Theta; \Phi_1 \cup \Phi_2 \cup \{E \mathbin{/} \mathtt{m} <: \gamma\}; \mathtt{m} \odot (E \mathbin{/} \mathtt{m})}$$

as required.

$\square$