Speak Now

Safe Actor Programming with Multiparty Session Types

DRAFT: October 2025

SIMON FOWLER, University of Glasgow, United Kingdom
RAYMOND HU, Queen Mary University of London, UK, United Kingdom

Actor languages such as Erlang and Elixir are widely used for implementing scalable and reliable distributed applications, but the informally-specified nature of actor communication patterns leaves systems vulnerable to costly errors such as communication mismatches and deadlocks. *Multiparty session types* (MPSTs) rule out communication errors early in the development process, but until now, the nature of actor communication has made it difficult for actor languages to benefit from session types.

This paper introduces Maty, the first actor language design supporting both *static* multiparty session typing and the full power of actors taking part in *multiple sessions*. Our main insight is to enforce session typing through a flow-sensitive type-and-effect system, combined with an event-driven programming style and first-class message handlers. Using MPSTs allows us to guarantee communication safety: a process will never send or receive an unexpected message, nor will it ever get stuck waiting for a message that will never arrive.

We extend Maty to support Erlang-style supervision and cascading failure, and show that this preserves Maty's strong metatheory. We implement Maty in Scala using an API generation approach, and evaluate our implementation on a series of microbenchmarks, a factory scenario, and a chat server.

1 INTRODUCTION

Modern infrastructure depends on distributed software. Unfortunately, writing distributed software is difficult: developers must reason about a host of issues such as deadlocks, failures, and adherence to complex communication protocols. Actor languages such as Erlang and Elixir, and frameworks like Akka, are popular for writing scalable, resilient systems; Erlang in particular powers the servers of WhatsApp, which has billions of users worldwide. Actor languages support lightweight processes that communicate through asynchronous explicit message passing rather than shared memory, and support robust failure recovery strategies like *supervision hierarchies*.

Nevertheless, actor languages are not a silver bullet: it is still possible—*easy*, even—to introduce subtle bugs that can lead to errors that are difficult to detect, debug, and fix. Examples include waiting for a message that will never arrive, sending a message that cannot be handled, or sending an incorrect payload. *Multiparty session types* (MPSTs) [7, 30] are *types for protocols* and allow us to reason about structured interactions between communicating participants. If each participant typechecks against its session type, then the system is statically guaranteed to correctly implement the associated protocol, in turn catching communication errors before a program is run.

MPSTs therefore offer a tantalising promise for actor languages: by combining the fault-tolerance and ease-of-distribution of actor languages with the correctness guarantees given by MPSTs, users can fearlessly write robust and scalable distributed code, confident in the absence of protocol errors. Unfortunately, there is a spanner in the works: MPSTs have been primarily studied for *channel-based* languages, which have a significantly different communication model, and current session-typing approaches for actor languages are severely limited in expressiveness. Other behavioural type systems for actors struggle to capture *structured* interactions and *handle failure* effectively.

In this paper we present Maty, the first actor language that supports statically-checked multiparty session types and failure handling, combining the error prevention mechanism of session types and the scalability and fault tolerance of actor languages. Our key insight is to adopt an *event-driven programming style* and enforce session typing through a *flow-sensitive effect system*.

Authors' addresses: Simon Fowler, University of Glasgow, United Kingdom; Raymond Hu, Queen Mary University of London, UK, United Kingdom.

1.1 Actor Languages

Actor languages and frameworks are inspired by the *actor model* [2, 29], where an actor reacts to incoming messages by spawning new actors, sending a finite number of messages to other actors, and changing the way it reacts to future messages. Consider the following Akka implementation of an ID server, which generates a fresh number for every client request:

```
def idServer(count: Int): Behavior[IDRequest] = {
  Behaviors.receive { (context, message) =>
    message.replyTo ! IDResponse(count)
    idServer(count + 1)
  }
}
```

The `idServer` function records the current request count as its state, and responds to an incoming `IDRequest` by sending the current `count` before recursing with an incremented request counter.

It is straightforward to specify the client-server *protocol* for this example as a session *type* between these two roles, but there are key problems implementing and verifying even this simple example in standard MPST frameworks. First, actor programming is inherently *reactive*: computation is driven by the reception of a new message, and actors must be able to respond to requests from a *statically-unknown* number of clients. Second, each response depends on some common *state*. Classical MPSTs are instead based on session π-calculus, which is effectively a model of *proactive multithreading* as opposed to reactive event handling. A standard MPST server process relies on *replication* to spawn a separate (π-calculus) process to handle each client session concurrently. For reference, common notations/patterns include:

$$\mathsf{Server} = a(x).(P_{\mathsf{thread}}(x) \mid \mathsf{Server}) \qquad \text{or} \qquad \mathsf{Server} = !\, a(x).P_{\mathsf{thread}}(x)$$

```
def idServer(count: Int, locked: Boolean):
    Behavior[IDServerRequest] = {
  Behaviors.receive { (context, message) =>
    message match {
      case IDRequest(replyTo) =>
        if (locked) {
            replyTo ! Unavailable()
            idServer(count, locked)
        } else {
            replyTo ! IDResponse(count)
            idServer(count + 1, locked)
        }
      case LockRequest(replyTo) =>
        if (locked) {
            replyTo ! Unavailable()
            idServer(count, locked)
        } else {
            replyTo ! Locked(context.self)
            idServer(count, true)
        }
      case Unlock() =>
        idServer(count, false)
    }
  }
}
```

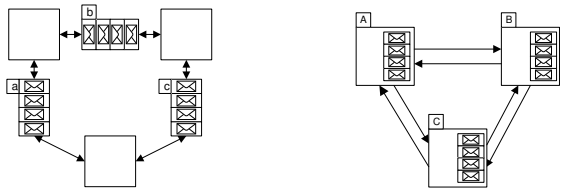
Fig. 1. ID server extended with locking

This model has no direct support for coordinating a *dynamically variable* number of such separate client-handler processes/sessions, and—crucially—key safety properties of standard MPSTs such as deadlock-freedom **only hold when each process engages in a single session and each session can be conducted fully independently from the others** (i.e., an embarrassingly parallel situation). Introducing any method to synchronise shared state between these processes, be it through an intricate web of additional internal sessions or some out-of-band (i.e., non-session-typed) method, means deadlock-freedom is no longer guaranteed.

Besides safety concerns, the π-calculus based programming model makes it difficult to express important patterns such as a *single* process waiting to reactively receive from senders across multiple sessions, since inputs are normally modelled as direct, blocking operations.

A locking ID server. Figure 1 shows a simple extension of our ID server, where a participant can choose to *lock* the server to prevent it from generating fresh IDs until the lock is released.

In this example, replies depend on whether the ID server is locked. Upon receiving an `IDRequest` message, if the server is locked, then it will respond with `Unavailable`; otherwise, it will reply as before. If an unlocked server receives a `LockRequest` message, it responds with `Locked` and sets the `locked` flag. A subsequent `Unlock` message resets the `locked` flag.



(a) Channels (b) Actors

Fig. 2. Channel- and actor-based languages [24]

This small extension to the example reveals some intricacies: once a client has received a lock, it is in a *different state of the protocol* to the remaining clients. First, there is no straightforward way of guaranteeing that the client ever sends an `Unlock` message, nor that the `Unlock` message was sent by the same actor that acquired the lock. Second, the server must *always* be able to handle an `Unlock` message, *even when it is already unlocked*—permitting an invalid state. Both of these issues can be straightforwardly solved using session types in Maty.

1.2 Channels vs. Actors

Session types were originally developed for channel-based languages like Go and Concurrent ML (Figure 2a). Channel-based languages languages support anonymous processes that communicate over *channel endpoints*, supporting either *synchronous* or *asynchronous* communication. In actor languages (Figure 2b) such as Erlang or Elixir, *named* processes send messages directly to each others' mailboxes. The difference in communication models has significant consequences for distribution and typing. We can easily give a channel endpoint precise types, e.g., Chan(Int) or a *session type* such as !Int.!Int.?Bool.end to state that the channel should be used to send two integers and receive a Boolean. However, efficiently implementing channels requires us to store buffered data at the same location that it is processed, but difficulties arise when sending channel endpoints as part of a message (known as *distributed delegation*). Furthermore, implementing even basic channel idioms such as choosing between multiple channels requires complex distributed algorithms [12]. In short, channel-based languages are *easy to type* but *difficult to distribute*.

In contrast, actor languages are much easier to distribute, since every message will always be stored at the process that will handle it. But typing an actor is harder, requiring large variant types, and behavioural typing is difficult since we can only *send* to process IDs and *receive* from mailboxes. Thus, actors are *easy to distribute* but *difficult to type*.

1.3 Key Principles

For session types to be useful for real-world actor programs, we argue that a programming model and session type discipline must satisfy the following *key principles*:

(KP1) Reactivity Following the actor model, frameworks like Akka, and Erlang behaviours like `gen_server`, computation should be triggered by incoming messages.

(KP2) No Explicit Channels Channel-based languages impose a significantly different programming style, so the programming model should *not* expose explicit channels to a developer.

(KP3) Multiple Sessions Actors must be able to simultaneously take part in an unbounded and statically-unknown number of sessions, in order to support server applications. It must be possible for different participants to be at different states of a protocol.

(KP4) Interaction Between Sessions Much like our ID server example, interactions in one session should be able to affect the behaviour of an actor in other sessions.

(KP5) Failure Handling and Recovery The programming model and type discipline should support failure recovery via supervision hierarchies.

No previous work that applies session types to actor languages satisfies the key principles above. Mostrous and Vasconcelos [43] investigated session typing for Core Erlang by emulating session-typed channels using unique references and selective receive. Their approach was unimplemented, not reactive, exposed a channel-based discipline, and does not support failure, violating **KP1, 2, 5**. Francalanza and Tabone [25] implemented a binary session typing system for Elixir, but their approach is limited to typing interactions between isolated pairs of processes and is therefore severely limited in expressiveness, violating **KP1, 3, 4, 5**. Harvey et al. [28] used multiparty session types in an actor language to support safe runtime adaptation, but each actor can only take part in a *single session* at a time. It is therefore difficult to write server applications and so the language *does not support general-purpose actor programming*, violating **KP1, 3, 4**.

Neykova and Yoshida [45] and Fowler [21] implement programming frameworks closer to following our key principles: each actor is programmed in a reactive style and can be involved in multiple sessions, but both works use *dynamic verification of actors using session types as a notation for generating runtime monitors*. They do not consider any formalism, session type system, nor metatheoretical guarantees, and so there is a significant gap between their conceptual framework and a concrete static programming language design.

In contrast, Maty supports our key principles by reacting to incoming messages rather than having an explicit receive operation (**KP1**); enforcing session typing through a flow-sensitive effect system rather than explicit channel handles (**KP2**); using the reactive design to support interleaved handling of messages from different sessions (**KP3**); supporting interaction between sessions using state, self-messages, and an explicit session switching construct (**KP4**); and supporting graceful session failure and failure recovery via supervision hierarchies (**KP5**).

1.4 Contributions

Concretely, we make three specific contributions:

(1) We introduce Maty, the first actor language design with full support for multiparty session types (§3). We show that Maty enjoys a strong metatheory including type preservation, progress, and global progress; in practice this means that Maty programs are free of communication mismatches and deadlocks (§4).
(2) We show how to extend Maty with support for Erlang-style failure handling and process supervision (§5), and prove that this maintains Maty's strong metatheory.
(3) We detail our implementation of Maty using an API generation approach in Scala (§6), and demonstrate our implementation on series of benchmarks, a real-world case study from the factory domain, and a chat server application.

Section 7 discusses related work, and Section 8 concludes.

2 A TOUR OF MATY

In this section we introduce Maty by example, first by considering how to write our ID server, and then by considering a larger online shop example.

2.1 The Basics: ID Server

Session types. Figure 3 shows the session types for the ID server example. The *global type* describes the interactions between the ID server and a client. For simplicity, we assume a standard encoding of mutually recursive types and use mutually recursive definitions in our examples. The client starts by sending one of IDRequest, LockRequest, or Quit to the server. On receiving IDRequest, the server replies with IDResponse if it is unlocked, or Unavailable if it is locked; in both cases, the protocol then repeats. On receiving LockRequest, the server replies with Locked (if it locks

Global Type for ID Server

```
IDServer ≜
  Client → Server : {
    IDRequest() .
      Server → Client : {
        IDResponse(Int) . IDServer,
        Unavailable() . IDServer
      },
    LockRequest() .
      Server → Client : {
        Locked() . AwaitUnlock,
        Unavailable() . IDServer
      },
    Quit() . end
  }
AwaitUnlock ≜
  Client → Server : Unlock() . IDServer
```

Local Type for Server Role

```
ServerTy ≜
  Client &{
    IDRequest() .
      Client ⊕{
        IDResponse(Int) . ServerTy,
        Unavailable() . ServerTy
      },
    LockRequest() .
      Client ⊕{
        Locked() . ServerLockTy
        Unavailable() . ServerTy
      },
    Quit() . end
  }
ServerLockTy ≜
  Client & Unlock() . ServerTy
```

Fig. 3. Session types for the ID server example.

successfully), and the client must then send Unlock before repeating. If already locked, the server responds with Unavailable. The protocol ends when the server receives a Quit message.

A global type can be *projected* to *local types* that describe the protocol from the perspective of each participant. The local type on the right details the protocol from the server's viewpoint: the & operator denotes offering a choice, and the ⊕ operator denotes making a selection. The (omitted) ClientTy type is similar, but implements the *dual* actions: where the server offers a choice, the client makes a selection, and vice-versa. We define a *protocol P* as a mapping from role names to local session types. In our example we define IDServerProtocol ≜ {Client : ClientTy, Server : ServerTy}.

Programming model. The Maty programming model is as follows:

- Maty is faithful to the actor model, which has a single thread of execution per actor. This allows access to shared state *without* needing concurrency control mechanisms like mutexes.
- An actor registers with an *access point* to register to take part in a session.
- Once a session is established, the actor can send messages according to its session type. The actor maintains some *actor-level state* and its active thread must either return an updated state (if it has completed its part in the protocol), or suspend by installing a *message handler* (if it is ready to receive a message). Suspension acts as a *yield point* to the event loop, and occurs at **precisely the same point as in mainstream actor languages.**
- The event loop can then invoke other installed handlers for any messages in its mailbox—**this is the key mechanism that allows Maty to support multiple sessions**.

Implementing the ID server. Figure 4 shows an implementation of the ID server in Maty; we allow ourselves the use of mutually-recursive definitions, taking advantage of the usual encoding into anonymous recursive functions. Although we use an effect system that annotates function arrows, we omit effect annotations where they are not necessary.

The server maintains actor-level state of type (Int × Bool), containing the current ID and a flag recording whether the server is locked. The idServer function takes an *access point* [26] and initial state as an argument, and registers for the Server role. An access point can be thought of as a "matchmaking service": actors *register* to play a role in a session, and the access point establishes a session once actors have registered for each role. The **register** construct takes three arguments: an access point, a role, and a callback to be invoked when the session is established. *Once the callback is invoked, the actor can perform session communication actions for the given role*: in this case, the

```
idServer : (AP(IDServerProtocol) × (Int × Bool))
  → (Int × Bool)
idServer = λ(ap, state). registerAgain ap; state

registerAgain : (AP(IDServerProtocol)) → Unit
registerAgain = λap.
  register ap Server
    (λst. registerAgain ap; suspend requestHandler st)

unlockHandler : Handler(ServerLockTy, (Int × Bool))
unlockHandler =
  handler Client st {
    Unlock() ↦
      let (currentID, locked) = st in
      suspend requestHandler (currentID, false)
  }

main : Unit
main =
  let idServerAP = newAP[IDServerProtocol] in
  spawn (idServer (idServerAP, (0, false)));
  spawn (client idServerAP)
```

```
requestHandler : Handler(ServerTy, (Int × Bool))
requestHandler =
  handler Client st {
    IDRequest() ↦
      let (currentID, locked) = st in
      if locked then
        Client ! Unavailable();
        suspend requestHandler st
      else
        Client ! IDResponse (currentID);
        suspend requestHandler (currentID + 1, locked),
    LockRequest() ↦
      let (currentID, locked) = st in
      if locked then
        Client ! Unavailable();
        suspend requestHandler st
      else
        Client ! Locked();
        suspend unlockHandler (currentID, true),
    Quit() ↦ st
  }
```

Fig. 4. Maty implementation of ID Server

actor can communicate according to the ServerTy type, namely receiving the initial item request from a client. The callback first recursively registers to be involved in future sessions, and then *suspends* awaiting a message from a client, by installing requestHandler.

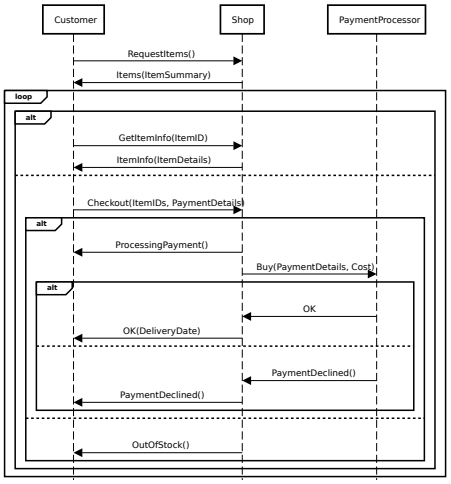
A *message handler* (or simply *handler*) is a first-class construct that describes how an actor handles an incoming message: each handler takes the role to receive from; a variable to which to bind the current actor state; and a series of branches that detail how each message should be handled. An actor *installs* a message handler for the current session by invoking the **suspend** construct, which reverts the actor back to being idle with an updated state, and indicates that the given handler should be invoked when a message is received from the Client.

Tying the example together. The requestHandler has type Handler(ServerTy, (Int × Bool)): handlers are parameterised by an input session type and the type of the actor's state. The handler has three branches, one for each possible incoming message. Maty uses a flow-sensitive effect system [3, 20, 27] to enforce session typing using pre- and post-conditions on expressions. In the IDRequest branch, the pre-condition Client ⊕{IDResponse(Int) . ServerTy, Unavailable() . ServerTy} means that the actor can *only send IDResponse or Unavailable messages*; all other communication actions are rejected statically. After either message is sent, the session type advances to ServerTy, allowing the handler to suspend recursively. The LockRequest branch works similarly; since **suspend** aborts the current evaluation context, both branches can be given type (Int × Bool) with post-condition end to match the Quit branch. The unlockHandler handles Unlock by updating the state, reinstalling requestHandler, and suspending. Implementing a client is similar (details omitted). Finally, `main` sets up the access point (associating Client with ClientTy and Server with ServerTy), then spawns idServer with a pair of arguments *idServerAP* (to allow the server to register for sessions) and (0, **false**), its initial state. The `main` function also spawns the client, passing the same access point.

2.2 A Larger Example: A Shop

Our ID server example demonstrated many of the important parts of Maty, but only considers interactions between *two* roles. Let us now consider a larger example of an online shop, depicted

Scenario Description



- A **Shop** can serve many **Customer**s at once.
- The **Customer** begins by requesting a list of items from the **Shop**, which sends back a list of pairs of an item's identifier and name.
- The **Customer** can then repeatedly either request full details (including description and cost) of an item, or proceed to checkout.
- To check out, the **Customer** sends their payment details and a list of item IDs to the **Shop**.
- If any items are out of stock, then the **Shop** notifies the customer who can then try again. Otherwise, the **Shop** notifies the **Customer** that it is processing the payment, and forwards the payment details and total cost to the **Payment Processor**.
- The **Payment Processor** responds to the **Shop** with whether the payment was successful.
- The **Shop** relays the result to the **Customer**, with a delivery date if the purchase was successful.

Local Types for Shop role

```
ShopTy ≜
  Customer & requestItems() .
  Customer ⊕ items([(ItemID × ItemName)]) .
  ReceiveCommand

PaymentResponse ≜
  PaymentProcessor &{
    ok() .
      Customer ⊕ ok(DeliveryDate) .
      ReceiveCommand,
    paymentDeclined() .
      Customer ⊕ paymentDeclined() .
      ReceiveCommand
  }

ReceiveCommand ≜
  Customer &{
    getItemInfo(ItemID) .
      Customer ⊕ itemInfo(Description) .
      ReceiveCommand,
    checkout(([ItemID] × PaymentDetails)) .
      Customer ⊕{
        paymentProcessing() .
          PaymentProcessor ⊕
            buy((PaymentDetails × Price)) .
            PaymentResponse,
        outOfStock() .
          Customer ⊕ outOfStock() .
          ReceiveCommand
      }
  }
```

Fig. 5. Online Shop Scenario

in Figure 5, that we will use as a running example throughout the rest of the paper. In short, the scenario involves multiple clients interacting with a single shop, and where the shop connects with an external payment processor. Figure 5 also shows the local types for the Shop role; we omit the ClientTy and PPTy types for the Client and PaymentProcessor respectively, but they follow a similar pattern. The global type closely follows the sequence diagram.

Shop message handlers. Figure 6 shows the shop's message handlers. After spawning, the shop suspends with itemReqHandler, awaiting a requestItems message. On receipt, it retrieves the current stock from its state, sends a summary to the customer, and installs custReqHandler.

The custReqHandler handles the getItemInfo and checkout messages. For getItemInfo, the shop sends item details and suspends recursively. For checkout, it checks availability: if all items are in stock, it notifies the customer, updates the stock, sends buy to the payment processor, and installs paymentHandler; otherwise, it sends outOfStock and reinstalls custReqHandler.

```
itemReqHandler : Handler(ShopTy, [Item])
itemReqHandler ≜
  handler Customer stock {
    requestItems() ↦
      Customer ! itemSummary(summary(stock));
      suspend custReqHandler stock
  }

paymentHandler : [ItemID] →
   Handler(PaymentResponse, [Item])
paymentHandler ≜ λitemIDs.
  handler PaymentProcessor stock {
    ok() ↦
      Customer ! ok(deliveryDate(itemIDs));
      suspend custReqHandler stock
    paymentDeclined() ↦
      Customer ! paymentDeclined();
      let newStock = increaseStock(itemIDs, stock) in
      suspend custReqHandler newStock
  }
```

```
custReqHandler : Handler(ReceiveCommand, [Item])
custReqHandler ≜
  handler Customer stock {
    getItemInfo(itemID) ↦
      Customer ! itemInfo(lookupItem(itemID, stock));
      suspend custReqHandler stock
    checkout((itemIDs, details)) ↦
      if inStock(itemIDs, stock) then
        Customer ! paymentProcessing();
        let total = cost(itemIDs, stock) in
        let newStock = decreaseStock(itemIDs, stock) in
        PaymentProcessor ! buy((details, total));
        suspend (paymentHandler itemIDs) newStock
      else
        Customer ! outOfStock();
        suspend custReqHandler stock
  }
```

Fig. 6. Implementation of Shop message handlers in Maty

The paymentHandler waits for the processor's reply: if it receives ok, it sends the delivery date; if it instead receives paymentDeclined, it restores the previous stock. Both branches reinstall custReqHandler to handle future requests.

Tying the example together. Finally, we can show how to establish a session using the Shop actors. Let CustomerProtocol = {Shop : ShopTy, Client : ClientTy, PaymentProcessor : PPTy}.

```
main ≜
let custAP = newAP[CustomerProtocol] in
spawn (shop (custAP, initialStock));
spawn (paymentProcessor custAP);
spawn (customer custAP)
```

```
shop ≜ λ(custAP, stock). registerAgain custAP; stock

registerAgain ≜ λcustAP.
  register custAP Shop
    (λst. registerAgain custAP; suspend itemReqHandler st)
```

The shop definition takes the access point and then proceeds to *register* to take part in a session to interact with customers. After each session has been established, the session type for the shop states that it needs to receive a message from a client, so the shop suspends with itemReqHandler.

3 MATY: A CORE ACTOR LANGUAGE WITH MULTIPARTY SESSION TYPES

In this section we introduce Maty, giving its syntax, typing rules, and semantics.

3.1 Syntax

Figure 8 shows the syntax of Maty. We let p, q range over roles, and x, y, z, f range over variables. We stratify the calculus into values V, W and computations M, N in the style of *fine-grain call-by-value* [39], with different typing judgements for each.

Session types. Although global types are convenient for describing protocols, we instead follow Scalas and Yoshida [52] and base our formalism around local types (*projection* of global types onto roles is standard [30, 50]; the local types resulting from projecting a global type satisfy the properties that we will see in §4 [52]). *Selection* session types $\mathtt{p} \oplus \{\ell_i(A_i).\, S_i\}_{i \in I}$ indicate that a process can choose to send a message with label ℓ_j and payload type A_j to role p, and continue as session type S_j (assuming $j \in I$). *Branching* session types $\mathtt{p} \,\&\, \{\ell_i(A_i).\, S_i\}_{i \in I}$ indicate that a process

Syntax of terms

Roles	p, q		
Variables	x, y, z, f		
Values	V, W	::=	$x \mid \lambda x.M \mid \mathbf{rec}\, f(x).M \mid c \mid (V, W) \mid \mathbf{handler}\ \mathsf{p}\ st\ \{\overrightarrow{H}\}$
Handler clauses	H	::=	$\ell(x) \mapsto M$
Computations	M, N	::=	$\mathbf{let}\ x = M\ \mathbf{in}\ N \mid \mathbf{return}\ V \mid V\ W$
		|	$\mathbf{if}\ V\ \mathbf{then}\ M\ \mathbf{else}\ N \mid \mathbf{let}\ (x, y) = V\ \mathbf{in}\ M$
		|	$\mathbf{spawn}\ M \mid \mathsf{p}\,!\,\ell(V) \mid \mathbf{suspend}\ V\ W$
		|	$\mathbf{newAP}[P] \mid \mathbf{register}\ V\ \mathsf{p}\ W$

Syntax of types and type environments

Output session types	$S^!$	::=	$\mathsf{p} \oplus \{\ell_i(A_i).S_i\}_{i \in I}$
Input session types	$S^?$	::=	$\mathsf{p}\,\&\{\ell_i(A_i).S_i\}_{i \in I}$
Session types	S, T	::=	$S^! \mid S^? \mid \mu X.S \mid X \mid \mathsf{end}$
Protocols	P	::=	$\{\mathsf{p}_i : S_i\}_{i \in I}$
Types	A, B, C	::=	$D \mid A \xrightarrow[C]{S,T} B \mid (A \times B) \mid \mathsf{AP}((\mathsf{p}_i : S_i)_{i \in I}) \mid \mathsf{Handler}(S^?, C)$
Base types	D	::=	$\mathsf{Unit} \mid \mathsf{Bool} \mid \mathsf{Int} \mid \cdots$
Type environments	Γ	::=	$\cdot \mid \Gamma, x : A$

Fig. 7. Syntax of terms, types, and type environments

must *receive* a message. We let $S^!$ range over selection (or *output*) session types, and let $S^?$ range over branching (or *input*) session types. Session type $\mu X.S$ indicates a recursive session type that binds variable X in S; we take an equi-recursive view of session types and identify each recursive session type with its unfolding. Finally, end denotes a session type that has finished.

Protocols. A *protocol* P is a collection of roles and their associated session types. Protocols are used when defining access points, and in Section 4 when describing behavioural properties.

Types. Base types D are standard. Since our type system enforces session typing by pre- and post-conditions, a function type $A \xrightarrow[C]{S,T} B$ states that the function takes an argument of type A where the current session type is S, and produces a result of type B with resulting session type T, to be run on an actor with state of type C. An access point has type $\mathsf{AP}((\mathsf{p}_i : S_i)_{i \in 1..n})$, mapping each role to a local type. Finally, a message handler has type $\mathsf{Handler}(S^?, A)$ where $S^?$ is an *input* session type and A is the type of the actor state.

3.2 Typing Rules

Values. Fig. 8 gives the typing rules for Maty. The value typing judgement $\Gamma \vdash V : A$ states that value V has type A under environment Γ. Unlike many session type systems, we do not need linear types since session typing is enforced by effect typing (following Harvey et al. [28]). Typing rules for variables and constants are standard (we assume constants include the unit value () of type Unit), and typing rules for anonymous functions and anonymous recursive functions are adapted to include pre- and postconditions. Message handlers specify how to handle incoming messages: TV-Handler states that a message handler $\mathbf{handler}\ \mathsf{p}\ st\ \{\ell_i(x_i) \mapsto M_i\}_i$ is typable with type $\mathsf{Handler}(\mathsf{p}\,\&\{\ell_i(A_i)\,.\,S_i\}_i, C)$ if each continuation M_i is typable with session precondition S_i where the environment is extended with x_i of type A_i and st of type C, and all branches have the postcondition end.

Computations. The computation typing judgement has the form $\Gamma \mid C \mid S \triangleright M : A \triangleleft T$, read as "under type environment Γ and evaluating in an actor with state of type C, given session precondition S, term M has type A and postcondition T". A let-binding $\mathbf{let}\ x = M\ \mathbf{in}\ N$ evaluates M and binds its result to x in N, with the session postcondition from typing M used as the precondition

Value typing $\boxed{\Gamma \vdash V : A}$

$$\text{TV-Var}\ \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \text{TV-Const}\ \frac{c \text{ has base type } D}{\Gamma \vdash c : D}$$

$$\text{TV-Lam}\ \frac{\Gamma, x : A \mid C \mid S \triangleright M : B \triangleleft T}{\Gamma \vdash \lambda x.M : A \xrightarrow[C]{S,T} B} \qquad \text{TV-Rec}\ \frac{\Gamma, x : A, f : A \xrightarrow[C]{S,T} B \mid C \mid S \triangleright M : B \triangleleft T}{\Gamma \vdash \mathbf{rec}\ f(x).M : A \xrightarrow[C]{S,T} B}$$

$$\text{TV-Pair}\ \frac{\Gamma \vdash V : A \qquad \Gamma \vdash W : B}{\Gamma \vdash (V, W) : (A \times B)}$$

$$\text{TV-Handler}\ \frac{(\Gamma, x_i : A_i, st : C \mid C \mid S_i \triangleright M_i : C \triangleleft \mathsf{end})_{i \in I}}{\Gamma \vdash \mathbf{handler}\ \mathrm{p}\ st\ \{\ell_i(x_i) \mapsto M_i\}_{i \in I} : \mathsf{Handler}(\mathrm{p}\,\&\{\ell_i(A_i).S_i\}_{i \in I}, C)}$$

Computation typing $\boxed{\Gamma \mid C \mid S \triangleright M : A \triangleleft T}$

$$\text{T-Let}\ \frac{\Gamma \mid C \mid S_1 \triangleright M : A \triangleleft S_2 \qquad \Gamma, x : A \mid C \mid S_2 \triangleright N : B \triangleleft S_3}{\Gamma \mid C \mid S_1 \triangleright \mathbf{let}\ x = M\ \mathbf{in}\ N : B \triangleleft S_3} \qquad \text{T-Return}\ \frac{\Gamma \vdash V : A}{\Gamma \mid C \mid S \triangleright \mathbf{return}\ V : A \triangleleft S}$$

$$\text{T-App}\ \frac{\Gamma \vdash V : A \xrightarrow[C]{S,T} B \qquad \Gamma \vdash W : A}{\Gamma \mid C \mid S \triangleright V\ W : B \triangleleft T}$$

$$\text{T-LetPair}\ \frac{\Gamma \vdash V : (A_1 \times A_2) \qquad \Gamma, x : A_1, y : A_2 \mid C \mid S_1 \triangleright M : B \triangleleft S_2}{\Gamma \mid C \mid S_1 \triangleright \mathbf{let}\ (x, y) = V\ \mathbf{in}\ M : B \triangleleft S_2}$$

$$\text{T-If}\ \frac{\Gamma \vdash V : \mathsf{Bool} \qquad \Gamma \mid C \mid S_1 \triangleright M : A \triangleleft S_2 \qquad \Gamma \mid C \mid S_1 \triangleright N : A \triangleleft S_2}{\Gamma \mid C \mid S_1 \triangleright \mathbf{if}\ V\ \mathbf{then}\ M\ \mathbf{else}\ N : A \triangleleft S_2}$$

$$\text{T-Spawn}\ \frac{\Gamma \mid A \mid \mathsf{end} \triangleright M : A \triangleleft \mathsf{end}}{\Gamma \mid C \mid S \triangleright \mathbf{spawn}\ M : \mathsf{Unit} \triangleleft S}$$

$$\text{T-Send}\ \frac{j \in I \qquad \Gamma \vdash V : A_j}{\Gamma \mid C \mid \mathrm{p} \oplus \{\ell_i(A_i).S_i\}_{i \in I} \triangleright \mathrm{p}\,!\,\ell_j(V) : \mathsf{Unit} \triangleleft S_j} \qquad \text{T-Suspend}\ \frac{\Gamma \vdash V : \mathsf{Handler}(S^?, C) \qquad \Gamma \vdash W : C}{\Gamma \mid C \mid S^? \triangleright \mathbf{suspend}\ V\ W : A \triangleleft S'}$$

$$\text{T-NewAP}\ \frac{\mathsf{comp}((\mathrm{p}_i : T_i)_{i \in I})}{\Gamma \mid C \mid S \triangleright \mathbf{newAP}[(\mathrm{p}_i : T_i)_{i \in I}] : \mathsf{AP}((\mathrm{p}_i : T_i)_{i \in I}) \triangleleft S} \qquad \text{T-Register}\ \frac{j \in I \qquad \Gamma \vdash V : \mathsf{AP}((\mathrm{p}_i : T_i)_{i \in I}) \qquad \Gamma \vdash W : A \xrightarrow[A]{T_j, \mathsf{end}} A}{\Gamma \mid A \mid S \triangleright \mathbf{register}\ V\ \mathrm{p}_j\ W : \mathsf{Unit} \triangleleft S}$$

Fig. 8. Maty Static Semantics

when typing N (T-Let); note that this is the only evaluation context in the system. The **return** V expression is a trivial computation returning value V and has type A if V also has type A (T-Return). A function application $V\ W$ is typable by T-App provided that the precondition in the function type matches the current precondition, and advances the postcondition to that of the function type. Rule T-LetPair types a pair deconstruction by binding both pair elements in the continuation M. Rule T-If types a conditional if its condition is of type Bool and both continuations have the same return type and postcondition; this design is in keeping with analogous session type systems [25, 28], but by treating **suspend** as a control operator (with an arbitrary return type and postcondition) we can maintain expressiveness by allowing each branch to finish at a different session type.

The **spawn** M construct spawns a new actor that evaluates term M; rule T-Spawn states that if the spawned actor supports state type C, then M must also have type C to return the initial state. It must also have pre- and postconditions end because the spawned computation is not yet in a session and so cannot communicate. Rule T-Send types a send computation $\mathrm{p}\,!\,\ell(V)$ if ℓ is contained within the selection session precondition, and if V has the corresponding type; the postcondition is the session continuation for the specified branch. There is *no **receive** construct*, since receiving messages is handled by the event loop. Instead, when an actor wishes to receive a message, it must *suspend* itself with updated state W and install a message handler using **suspend** $V\ W$. The T-Suspend rule states that **suspend** $V\ W$ is typable if the handler is compatible with the current

Runtime syntax

Actor names	a, b		
Session names	s		
AP names	p		
Init. tokens	ι		
Runtime names	α	::=	$a \mid s \mid p \mid \iota$
Values	U, V, W	::=	$\cdots \mid p$
Type env.	Γ	::=	$\cdots$
		$\mid$	$\Gamma, p : \mathsf{AP}((\mathsf{p}_i : S_i)_i)$
Reduction labels	l	::=	$s \mid \tau$

Configurations	$C, \mathcal{D}$	::=	$(\nu\alpha)C \mid C \parallel \mathcal{D}$
		$\mid$	$\langle a, \mathcal{T}, \sigma, \rho\rangle \mid p(\chi) \mid s \triangleright \delta$
Message queues	δ	::=	$\epsilon \mid (\mathsf{p}, \mathsf{q}, \ell(V)) \cdot \delta$
Stored handlers	σ	::=	$\epsilon \mid \sigma, s[\mathsf{p}] \mapsto V$
Initialisation states	ρ	::=	$\epsilon \mid \rho, \iota \mapsto V$
Thread states	$\mathcal{T}$	::=	$\mathbf{idle}(V) \mid (M)^{s[\mathsf{p}]} \mid M$
Access point states	χ	::=	$(\mathsf{p}_i \mapsto \widetilde{\iota_i})_i$
Evaluation contexts	$\mathcal{E}$	::=	$[\,] \mid \mathbf{let}\ x = \mathcal{E}\ \mathbf{in}\ M$
Thread contexts	$\mathcal{M}$	::=	$\mathcal{E} \mid (\mathcal{E})^{s[\mathsf{p}]}$
Top-level contexts	$\mathcal{Q}$	::=	$[\,] \mid ([\,])^{s[\mathsf{p}]}$

Structural congruence (configurations) $\boxed{C \equiv \mathcal{D}}$

$$C \parallel \mathcal{D} \equiv \mathcal{D} \parallel C \qquad C \parallel (\mathcal{D} \parallel \mathcal{D}') \equiv (C \parallel \mathcal{D}) \parallel \mathcal{D}' \qquad (\nu\alpha_1)(\nu\alpha_2)C \equiv (\nu\alpha_2)(\nu\alpha_1)C \qquad (\nu s)(s \triangleright \epsilon) \parallel C \equiv C$$

$$\frac{\alpha \notin \mathsf{fn}(C)}{C \parallel (\nu\alpha)\mathcal{D} \equiv (\nu\alpha)(C \parallel \mathcal{D})} \qquad \frac{\mathsf{p}_1 \neq \mathsf{p}_2 \vee \mathsf{q}_1 \neq \mathsf{q}_2}{s \triangleright \sigma_1 \cdot (\mathsf{p}_1, \mathsf{q}_1, \ell_1(V_1)) \cdot (\mathsf{p}_2, \mathsf{q}_2, \ell_2(V_2)) \cdot \sigma_2 \equiv s \triangleright \sigma_1 \cdot (\mathsf{p}_2, \mathsf{q}_2, \ell_2(V_2)) \cdot (\mathsf{p}_1, \mathsf{q}_1, \ell_1(V_1)) \cdot \sigma_2}$$

Fig. 9. Operational semantics (1)

session type precondition and state type; since the computation does not return, it can be given an arbitrary return type and postcondition.

Sessions are initiated using *access points*: we create an access point for a session with roles and types $(\mathsf{p}_i : S_i)_i$ using **newAP**$[(\mathsf{p}_i : S_i)_i]$, which must be annotated with the set of roles and local types to be involved in the session (T-NewAP). The rule ensures that the protocol supported by the access point is *compliant*; will describe this further in §4, but at a high level, if a protocol is compliant then it is free of communication mismatches and deadlocks.

An actor can *register* to take part in a session as role p on access point V using **register** V p W; function W is a callback to be invoked once the session is established. Rule T-Register ensures that the access point must contain a session type T associated with role p, and since the initiation callback will be evaluated when the session is established, M must be typable under session type T. Since neither **newAP** nor **register** perform any communication, the session types are unaltered.

3.3 Operational semantics

Figure 9 introduces runtime syntax (i.e., syntax that is introduced during reduction), along with structural congruence.

Runtime syntax. To model the concurrent behaviour of Maty processes, we require additional runtime syntax. Runtime names are identifiers for runtime entities: actor names a identify actors; session names s identify established sessions; access points p identify access points; and *initialisation tokens* ι associate registration entries in an access point with registered initialisation continuations.

We model communication and concurrency through a language of *configurations* (reminiscent of π-calculus processes). A *name restriction* $(\nu\alpha)C$ binds runtime name α in configuration C, and the right-associative parallel composition $C \parallel \mathcal{D}$ denotes configurations C and $\mathcal{D}$ running in parallel.

An actor is represented as a 4-tuple $\langle a, \mathcal{T}, \sigma, \rho\rangle$, where $\mathcal{T}$ is a thread that can either be idle with state V (**idle**(V)); a term M that is not involved in a session; or $(M)^{s[\mathsf{p}]}$ denoting that the actor is evaluating term M playing role p in session s. An actor is *active* if its thread is M or $(M)^{s[\mathsf{p}]}$ (for some s, p, and M), and *idle* otherwise. A handler state σ maps endpoints to handlers, which are invoked when an incoming message is received and the actor is idle. The initialisation state ρ maps initialisation tokens to callbacks to be invoked whenever a session is established. Our reduction rules

Configuration reduction $\boxed{C \xrightarrow{l} \mathcal{D}}$

E-Send

$$\frac{}{\langle a, (\mathcal{E}[\mathsf{q}\,!\,\ell(V)])^{s[\mathsf{p}]}, \sigma, \rho\rangle \parallel s \triangleright \delta \xrightarrow{s} \langle a, (\mathcal{E}[\mathbf{return}\ ()])^{s[\mathsf{p}]}, \sigma, \rho\rangle \parallel s \triangleright \delta \cdot (\mathsf{p}, \mathsf{q}, \ell(V))}$$

E-React

$$\frac{(\ell(x) \mapsto M) \in \overrightarrow{H}}{\langle a, \mathbf{idle}(W), \sigma[s[\mathsf{p}] \mapsto \mathbf{handler}\ \mathsf{q}\ st\ \{\overrightarrow{H}\}], \rho\rangle \parallel s \triangleright (\mathsf{q}, \mathsf{p}, \ell(V)) \cdot \delta \xrightarrow{s} \langle a, (M\{V/x, W/st\})^{s[\mathsf{p}]}, \sigma, \rho\rangle \parallel s \triangleright \delta}$$

E-Suspend

$$\frac{}{\langle a, (\mathcal{E}[\mathbf{suspend}\ V\ W])^{s[\mathsf{p}]}, \sigma, \rho\rangle \xrightarrow{\tau} \langle a, \mathbf{idle}(W), \sigma[s[\mathsf{p}] \mapsto V], \rho\rangle}$$

E-Spawn

$$\frac{}{\langle a, \mathcal{M}[\mathbf{spawn}\ M], \sigma, \rho\rangle \xrightarrow{\tau} (\nu b)(\langle a, \mathcal{M}[\mathbf{return}\ ()], \sigma, \rho\rangle \parallel \langle b, M, \epsilon, \epsilon\rangle)}$$

E-Reset

$$\frac{}{\langle a, \mathcal{Q}[\mathbf{return}\ V], \sigma, \rho\rangle \xrightarrow{\tau} \langle a, \mathbf{idle}(V), \sigma, \rho\rangle}$$

E-NewAP

$$\frac{p \text{ fresh}}{\langle a, \mathcal{M}[\mathbf{newAP}[(\mathsf{p}_i : S_i)_{i\in I}]], \sigma, \rho\rangle \xrightarrow{\tau} (\nu p)(\langle a, \mathcal{M}[\mathbf{return}\ p], \sigma, \rho\rangle \parallel p((\mathsf{p}_i \mapsto \emptyset)_{i\in I}))}$$

E-Register

$$\frac{\iota \text{ fresh}}{\langle a, \mathcal{M}[\mathbf{register}\ p\ \mathsf{p}\ V], \sigma, \rho\rangle \parallel p(\chi[\mathsf{p} \mapsto \widetilde{\iota'}]) \xrightarrow{\tau} (\nu\iota)(\langle a, \mathcal{M}[\mathbf{return}\ ()], \sigma, \rho[\iota \mapsto V]\rangle \parallel p(\chi[\mathsf{p} \mapsto \widetilde{\iota'} \cup \{\iota\}]))}$$

E-Init

$$\frac{s \text{ fresh}}{(\nu\iota_{\mathsf{p}_i})_{i\in 1..n}(p((\mathsf{p}_i \mapsto \widetilde{\iota'_{\mathsf{p}_i}} \cup \{\iota_{\mathsf{p}_i}\})_{i\in 1..n}) \parallel \langle a_i, \mathbf{idle}(W_i), \sigma_i, \rho_i[\iota_{\mathsf{p}_i} \mapsto V_i]\rangle_{i\in 1..n}) \xrightarrow{\tau} (\nu s)(p((\mathsf{p}_i \mapsto \widetilde{\iota'_{\mathsf{p}_i}})_{i\in 1..n}) \parallel s \triangleright \epsilon \parallel \langle a_i, (V_i\ W_i)^{s[\mathsf{p}_i]}, \sigma_i, \rho_i\rangle_{i\in 1..n})}$$

E-Par

$$\frac{C \xrightarrow{l} C'}{C \parallel \mathcal{D} \xrightarrow{l} C' \parallel \mathcal{D}}$$

E-Lift

$$\frac{M \longrightarrow_{\mathsf{M}} N}{\langle a, \mathcal{M}[M], \sigma, \rho\rangle \xrightarrow{\tau} \langle a, \mathcal{M}[N], \sigma, \rho\rangle}$$

E-Nu

$$\frac{C \xrightarrow{l} \mathcal{D}}{(\nu\alpha)C \xrightarrow{l-\alpha} (\nu\alpha)\mathcal{D}}$$

E-Struct

$$\frac{C \equiv C' \qquad C' \xrightarrow{l} \mathcal{D}' \qquad \mathcal{D}' \equiv \mathcal{D}}{C \xrightarrow{l} \mathcal{D}}$$

where $l - \alpha = \tau$ if $l = \alpha$, and l otherwise

Fig. 10. Operational semantics (2)

(Figure 10) make use of indexing notation as syntactic sugar for parallel composition: for example, $\langle a_i, \mathcal{T}_i, \sigma_i, \rho_i\rangle_{i\in 1..n}$ is syntactic sugar for the configuration $\langle a_1, \mathcal{T}_1, \sigma_1, \rho_1\rangle \parallel \cdots \parallel \langle a_n, \mathcal{T}_n, \sigma_n, \rho_n\rangle$.

An access point $p(\chi)$ has name p and state χ, where the state maps roles to sets of initialisation tokens for actors that have registered to take part in the session. Finally, each session s is associated with a queue $s \triangleright \delta$, where δ is a list of entries $(\mathsf{p}, \mathsf{q}, \ell(V))$ denoting a message $\ell(V)$ sent from p to q.

Initial configurations. A program M is run by placing it in an *initial configuration* $(\nu a)(\langle a, M, \epsilon, \epsilon\rangle)$.

Structural congruence and term reduction. Structural congruence is the smallest congruence relation defined by the axioms in Figure 9. As with the π-calculus, parallel composition is associative and commutative, and we have the usual scope extrusion rule; we write $\mathsf{fn}(C)$ to refer to the set of free names in a configuration C. We also include a structural congruence rule on queues that allows us to reorder unrelated messages; notably this rule maintains message ordering between pairs of participants. Consequently, the session-level queue representation is isomorphic to a set of queues between each pair of roles. Term reduction $M \longrightarrow_{\mathsf{M}} N$ is standard β-reduction (omitted).

Communication and concurrency. It is convenient for our metatheory to annotate each communication reduction with the name of the session in which the communication occurs, although we sometimes omit the label where it is not relevant. Rule E-Send describes an actor playing role p in session s sending a message $\ell(V)$ to role q: the message is appended to the session queue and the operation reduces to **return** $()$. The E-React rule captures the event-driven nature of the system: if an actor is idle with state V, and has a stored handler for $s[\mathsf{p}]$, and there exists a matching message in the session queue, then the message is dequeued and the message handler is

evaluated with the message payload and state. If an actor is currently evaluating a computation in the context of a session $s[\mathsf{p}]$, rule E-Suspend evaluates **suspend** V W by installing handler V for $s[\mathsf{p}]$ and returning the actor to the **idle**(W) state. Rule E-Spawn spawns a fresh actor with empty handler and initialisation state, and E-Reset returns an actor to the **idle**(V) state once it has finished evaluating to an updated state V.

Session initialisation. Rule E-NewAp creates an access point with a fresh name p and empty mappings for each role. Rule E-Register evaluates **register** p p V by creating an initialisation token ι, storing a mapping from ι to the callback V in the requesting actor's initialisation state, and appending ι to the participant set for p in p. Finally, E-Init establishes a session when idle participants are registered for all roles: the rule discards all initialisation tokens, creates a session name restriction and empty session queue, and invokes all initialisation callbacks.

Example 3.1. Consider a simple Ping-Pong example. We can describe the protocol as:

$$\mathsf{PingPong} = \left\{ \begin{array}{l} \mathsf{Pinger} : \mathsf{Ponger} \oplus \mathsf{Ping}(\mathsf{Unit})\,.\,\mathsf{Ponger}\,\&\,\mathsf{Pong}(\mathsf{Unit})\,.\,\mathsf{end}, \\ \mathsf{Ponger} : \mathsf{Pinger}\,\&\,\mathsf{Ping}(\mathsf{Unit})\,.\,\mathsf{Pinger} \oplus \mathsf{Pong}(\mathsf{Unit})\,.\,\mathsf{end} \end{array} \right\}.$$

The main function and the initialisation functions for the Pinger and Ponger are described as:

$$\mathsf{main} \triangleq \textbf{let } ap = \textbf{newAP}[\mathsf{PingPong}] \textbf{ in spawn } \mathsf{pinger}\ ap;\ \textbf{spawn } \mathsf{ponger}\ ap$$

$\mathsf{ponger} \triangleq \lambda ap\,.\,\textbf{register } ap\ \mathsf{Ponger}\ \mathsf{pongerCallback}$
$\mathsf{pongerCallback} \triangleq \lambda().$
$\quad\textbf{suspend } (\textbf{handler } \mathsf{Pinger}\ st\ \{\mathsf{Ping} \mapsto \mathsf{Pinger}\,!\,\mathsf{Pong}(())\})\ ()$

$\mathsf{pinger} \triangleq \lambda ap\,.\,\textbf{register } ap\ \mathsf{Pinger}\ \mathsf{pingerCallback}$
$\mathsf{pingerCallback} \triangleq \lambda().$
$\quad\mathsf{Ponger}\,!\,\mathsf{Ping}(());$
$\quad\textbf{suspend } (\textbf{handler } \mathsf{Ponger}\ st\ \{\mathsf{Pong} \mapsto ()\})\ ()$

With these defined, we place the main function in an initial configuration, which creates a new access point p (E-NewAP) and spawns the Pinger and Ponger actors (E-Spawn):

$$(\nu a)(\langle a, \mathsf{main}, \epsilon, \epsilon\rangle) \longrightarrow^{+} (\nu ping)(\nu pong)(\nu p)(\nu a)\left(\begin{array}{l} \langle a, \textbf{idle}(()), \epsilon, \epsilon\rangle \parallel \langle ping, \mathsf{pinger}, \epsilon, \epsilon\rangle \parallel \langle pong, \mathsf{ponger}, \epsilon, \epsilon\rangle \\ \parallel p(\mathsf{Pinger} \mapsto \emptyset, \mathsf{Ponger} \mapsto \emptyset) \end{array} \right)$$

At this point, both of the actors can register with the access point (E-Register). By registering, the access points generate *initialisation tokens* ι_1, ι_2, which are stored both in the access point and also as keys in the actors' initialisation states. The actors then revert to being idle (E-Reset).

$$\longrightarrow^{+} (\nu \iota_1)(\nu \iota_2)(\nu ping)(\nu pong)(\nu p)(\nu a)\left(\begin{array}{l} \langle a, \textbf{idle}(()), \epsilon, \epsilon\rangle \\ \parallel \langle ping, \textbf{idle}(()), \epsilon, \iota_1 \mapsto \mathsf{pingerCallback}\rangle \parallel \langle pong, \textbf{idle}(()), \epsilon, \iota_2 \mapsto \mathsf{pongerCallback}\rangle \\ \parallel p(\mathsf{Pinger} \mapsto \{\iota_1\}, \mathsf{Ponger} \mapsto \{\iota_2\}) \end{array} \right)$$

Since the access point now has idle registered actors for each role, it establishes a session and removes the initialisation tokens (E-Init). Both actors evaluate their initialisation callbacks in the context of the newly-created session:

$$\longrightarrow^{+} (\nu s)(\nu ping)(\nu pong)(\nu p)(\nu a)\left(\begin{array}{l} \langle a, \textbf{idle}(()), \epsilon, \epsilon\rangle \\ \parallel \langle ping, (\mathsf{pingerCallback}\ ())^{s[\mathsf{Pinger}]}, \epsilon, \epsilon\rangle \parallel \langle pong, (\mathsf{pongerCallback}\ ())^{s[\mathsf{Ponger}]}, \epsilon, \epsilon\rangle \\ \parallel p(\mathsf{Pinger} \mapsto \emptyset, \mathsf{Ponger} \mapsto \emptyset) \parallel s \triangleright \epsilon \end{array} \right)$$

Following the behaviour in the callbacks, the Ponger suspends awaiting a message (E-Suspend), and the Pinger sends a message to the Ponger, which is stored in the session queue (E-Send).

$$\longrightarrow^{+} (\nu s)(\nu ping)(\nu pong)(\nu p)(\nu a)\left(\begin{array}{l} \langle a, \textbf{idle}(()), \epsilon, \epsilon\rangle \parallel \langle ping, (\textbf{suspend } (\textbf{handler } \mathsf{Ponger}\ st\ \{\mathsf{Pong} \mapsto ()\})\ ())^{s[\mathsf{Pinger}]}, \epsilon, \epsilon\rangle \\ \parallel \langle pong, \textbf{idle}(()), s[\mathsf{Ponger}] \mapsto \textbf{handler } \mathsf{Pinger}\ st\ \{\mathsf{Ping} \mapsto \mathsf{Pinger}\,!\,\mathsf{Pong}(())\}, \epsilon\rangle \\ \parallel p(\mathsf{Pinger} \mapsto \emptyset, \mathsf{Ponger} \mapsto \emptyset) \parallel s \triangleright (\mathsf{Pinger}, \mathsf{Ponger}, \mathsf{Ping}(())) \end{array} \right)$$

The Pinger can now suspend, awaiting a message from the Ponger (E-Suspend). Since there is a queued message for the idle Ponger, we can re-activate the suspended handler (E-React):

$$\longrightarrow^{+} (\nu s)(\nu ping)(\nu pong)(\nu p)(\nu a)\left(\begin{array}{l} \langle a, \textbf{idle}(()), \epsilon, \epsilon\rangle \\ \parallel \langle ping, \textbf{idle}(()), s[\mathsf{Pinger}] \mapsto \textbf{handler } \mathsf{Ponger}\ st\ \{\mathsf{Pong} \mapsto ()\}, \epsilon\rangle \\ \parallel \langle pong, (\mathsf{Pinger}\,!\,\mathsf{Pong}(()))^{s[\mathsf{Ponger}]}, \epsilon, \epsilon\rangle \\ \parallel p(\mathsf{Pinger} \mapsto \emptyset, \mathsf{Ponger} \mapsto \emptyset) \parallel s \triangleright \epsilon \end{array} \right)$$

Finally, the Ponger can send a Pong back to the Pinger, which activates the stored handler:

Runtime types, environments, and labels

Polarised initialisation tokens	$\iota^{\pm}$	::=	$\iota^{+} \mid \iota^{-}$
Queue types	Q	::=	$\epsilon \mid (\mathsf{p}, \mathsf{q}, \ell(A)) \cdot Q$
Runtime type environments	Δ	::=	$\cdot \mid \Delta, a \mid \Delta, p \mid \Delta, \iota^{\pm} : S \mid \Delta, s[\mathsf{p}] : S \mid \Delta, s : Q$
Labels	γ	::=	$s : \mathsf{p} \uparrow \mathsf{q}::\ell \mid s : \mathsf{p} \downarrow \mathsf{q}::\ell \mid \mathsf{end}(s, \mathsf{p})$

Structural congruence (queue types) $\boxed{Q \equiv Q'}$

$$\frac{\mathsf{p}_1 \neq \mathsf{p}_2 \vee \mathsf{q}_1 \neq \mathsf{q}_2}{Q_1 \cdot (\mathsf{p}_1, \mathsf{q}_1, \ell_1(A_1)) \cdot (\mathsf{p}_2, \mathsf{q}_2, \ell_2(A_2)) \cdot Q_2 \equiv Q_1 \cdot (\mathsf{p}_2, \mathsf{q}_2, \ell_2(A_2)) \cdot (\mathsf{p}_1, \mathsf{q}_1, \ell_1(A_1)) \cdot Q_2}$$

Runtime type environment reduction $\boxed{\Delta \xrightarrow{\gamma} \Delta'}$

$$\textsc{Lbl-Send}\quad \Delta, s[\mathsf{p}] : \mathsf{q} \oplus \{\ell_i(A_i).S_i\}_{i \in I}, s : Q \xrightarrow{s:\mathsf{p}\uparrow\mathsf{q}::\ell_j} \Delta, s[\mathsf{p}] : S_j, s : Q \cdot (\mathsf{p}, \mathsf{q}, \ell_j(A_j)) \quad (\text{if } j \in I)$$

$$\textsc{Lbl-Recv}\quad \Delta, s[\mathsf{p}] : \mathsf{q} \,\&\, \{\ell_i(A_i).S_i\}_{i \in I}, s : (\mathsf{q}, \mathsf{p}, \ell_j(A_j)) \cdot Q \xrightarrow{s:\mathsf{q}\downarrow\mathsf{p}::\ell_j} \Delta, s[\mathsf{p}] : S_j, s : Q \quad (\text{if } j \in I)$$

$$\textsc{Lbl-End}\quad \Delta, s[\mathsf{p}] : \mathsf{end} \xrightarrow{\mathsf{end}(s,\mathsf{p})} \Delta$$

$$\textsc{Lbl-Rec}\quad \Delta, s[\mathsf{p}] : \mu X.S \xrightarrow{\gamma} \Delta' \quad (\text{if } \Delta, s[\mathsf{p}] : S\{\mu X.S/X\} \xrightarrow{\gamma} \Delta')$$

Fig. 11. Labelled transition system on runtime type environments

$$\longrightarrow^{+} (vs)(vping)(vpong)(vp)(va)\left(\begin{array}{l}\langle a, \mathbf{idle}(()), \epsilon, \epsilon\rangle \parallel \langle ping, (())^{s[\mathsf{Pinger}]}, \epsilon, \epsilon\rangle \parallel \langle pong, (())^{s[\mathsf{Ponger}]}, \epsilon, \epsilon\rangle \\ \parallel p(\mathsf{Pinger} \mapsto \emptyset, \mathsf{Ponger} \mapsto \emptyset) \parallel s \triangleright \epsilon\end{array}\right)$$

Both actors have now finished the session and therefore revert to being idle (E-Reset).

4 METATHEORY

In order to prove metatheoretical properties about Maty, we define an extrinsic [49] type system for Maty configurations. **Note that our configuration type system is purely metatheoretical and used only to establish inductive invariants required for our proofs; we do *not* need to implement it in a typechecker and we do *not* require runtime type checking**.

Following Scalas and Yoshida [52] we begin by showing a type semantics for sets of local types. Using this semantics we can ensure that collections of local types are *compliant*, meaning that communicated messages are always compatible and that communication is deadlock-free, and use this to prove type preservation, progress, and global progress for Maty configurations.

Relations. We write $\mathcal{R}^{?}$, $\mathcal{R}^{+}$, and $\mathcal{R}^{*}$ for the reflexive, transitive, and reflexive-transitive closures of a relation $\mathcal{R}$ respectively. We write $\mathcal{R}_1\mathcal{R}_2$ for the composition of relations $\mathcal{R}_1$ and $\mathcal{R}_2$.

Runtime types and environments. Runtime environments are used to type configurations and to define behavioural properties on sets of local types. Unlike type environments Γ, runtime type environments Δ are *linear* to ensure safe use of session endpoints, and also to ensure that there is precisely one instance of each actor and access point. Runtime type environments can contain actor names a; access point names p; *polarised* initialisation tokens $\iota^{\pm} : S$ (since each initialisation token is used twice: once in the access point and one inside an actor's initialisation state); session endpoints $s[\mathsf{p}] : S$; and finally session queue types $s : Q$. Queue types mirror the structure of queue entries and consist of a series of triples $(\mathsf{p}, \mathsf{q}, \ell(A))$. We include structural congruence on queue types to match structural congruence on queues, and extend this to runtime environments.

Labelled transition system on environments. Figure 11 shows the LTS on runtime type environments. The Lbl-Send reduction gives the behaviour of an output session type interacting with a queue: supposing we send a message with some label ℓ_j from p to q, we advance the session type for p to the continuation S_j and add the message to the end of the queue. The Lbl-Recv rule handles receiving and works similarly, instead *consuming* the message from the queue. Rule

Lbl-End allows us to discard a session endpoint from the environment if it does not support any further communication, and Lbl-Rec allows reduction of recursive session types by considering their unrolling. We write $\Delta \Longrightarrow \Delta'$ if $\Delta \equiv \xrightarrow{\gamma} \equiv \Delta'$ for some synchronisation label γ, and conversely write $\Delta \not\Longrightarrow$ if there exists no Δ' such that $\Delta \Longrightarrow \Delta'$.

Protocol Properties. In order to prove type preservation and progress properties on Maty configurations, we need to ensure each protocol in the system is *compliant*, meaning that it is safe and deadlock-free. *Safety* is the minimum we can expect from a protocol in order for us to prove type preservation: a safe runtime type environment ensures that communication does not introduce type errors. Intuitively, safety ensures that a message received from a queue is of the expected type, thereby ruling out communication mismatches; safety properties must also hold under unfoldings of recursive session types and safety must be preserved by environment reduction. Deadlock-freedom on runtime type environments requires that every message that is sent in a protocol can eventually be received, and that a participant will never wait for a message that will never arrive.

Definition 4.1 (Compliance). A runtime environment Δ is *compliant*, written $\mathsf{comp}(\Delta)$, if it is *safe* and *deadlock-free*:

Safe An environment Δ is *safe*, written $\mathsf{safe}(\Delta)$, if:

- $\Delta = \Delta', s[\mathsf{p}] : \mathsf{q} \,\&\, \{\ell_i(A_i).S_i\}_{i\in I}, s : Q$ with $Q \equiv (\mathsf{q}, \mathsf{p}, \ell_j(B_j)) \cdot Q'$ implies $j \in I$ and $B_j = A_j$; and
- $\Delta = \Delta', s[\mathsf{p}] : \mu X.S$ implies $\mathsf{safe}(\Delta', s[\mathsf{p}] : S\{\mu X.S/X\})$; and
- $\mathsf{safe}(\Delta)$ and $\Delta \Longrightarrow \Delta'$ implies $\mathsf{safe}(\Delta')$.

Deadlock-free An environment Δ is *deadlock-free*, written $\mathsf{df}(\Delta)$, if $\Delta \Longrightarrow^* \Delta' \not\Longrightarrow$ implies $\Delta' = s : \epsilon$.

A *protocol* $\{\mathsf{p}_i : S_i\}_{i\in 1..n}$ is compliant if $\mathsf{comp}(s[\mathsf{p}_1] : S_1, \ldots, s[\mathsf{p}_n] : S_n)$ for an arbitrary s.

Checking compliance for an *asynchronous* protocol is undecidable in general [52], but various sound and tractable mechanisms can ensure it in practice. For example, syntactic projections from global types produce safe and deadlock-free sets of local types [52]. Furthermore, multiparty compatibility [18] allows safety to be verified by bounded model checking; this is the core approach implemented in Scribble [34], used by our implementation.

We have therefore designed our type system to be agnostic to any specific implementation method for validating compliance, as common in recent MPST language design papers (e.g., [28, 38]).

4.1 Configuration typing

Figure 12 shows the typing rules for Maty configurations. The configuration typing judgement $\Gamma;\Delta \vdash C$ can be read, "under type environment Γ and runtime type environment Δ, configuration C is well typed". We have three rules for name restrictions: read bottom-up, T-APName adds p to both the type and runtime environments, and rule T-InitName adds tokens of both polarities to the runtime type environment. Rule T-SessionName is key to the generalised multiparty session typing approach introduced by Scalas and Yoshida [52]: to type a name restriction $(\nu s)C$, the type environment Δ' consists of a set of session endpoints $\{s[\mathsf{p}_i]\}_i$ with session types S_{p_i}, along with a session queue $s : Q$. Environment Δ' must be compliant. The condition $s \notin \mathsf{snames}(\Delta)$ ensures that no other endpoint or queue with session name s may be present in the initial environment.

Rule T-Par types two parallel subconfigurations under disjoint runtime environments. Rule T-AP types an access point: it requires that the access point reference is included in Γ and through the auxiliary judgement $\{(\mathsf{p}_i : S_i)_i\} \;\Delta \vdash \chi$ ensures that each initialisation token in the access point has a compatible type. We also require that the protocol supported by the access point is compliant.

Rule T-Actor types an actor $\langle a, \mathcal{T}, \sigma, \rho\rangle$ using three auxiliary judgements. The thread state typing judgement $\Gamma;\Delta \mid C \vdash \mathcal{T}$ ensures that an active thread either performs all pending communication actions, or it suspends. The handler typing judgement $\Gamma;\Delta \mid C \vdash \sigma$ ensures that the stored

Configuration typing rules $\boxed{\Gamma;\Delta \vdash C}$

$$\text{T-APName}\ \frac{\Gamma, p : \mathsf{AP}((\mathrm{p}_i : S_i)_{i\in I});\Delta, p \vdash C}{\Gamma;\Delta \vdash (\nu p)C} \qquad \text{T-InitName}\ \frac{\Gamma;\Delta, \iota^+ : S, \iota^- : S \vdash C}{\Gamma;\Delta \vdash (\nu \iota)C}$$

$$\text{T-SessionName}\ \frac{\Delta' = \{s[\mathrm{p}_i] : S_{\mathrm{p}_i}\}_{i\in 1..n}, s : Q \quad \mathsf{comp}(\Delta') \quad s \notin \mathsf{snames}(\Delta) \quad \Gamma;\Delta, \Delta' \vdash C}{\Gamma;\Delta \vdash (\nu s)C} \qquad \text{T-ActorName}\ \frac{\Gamma;\Delta, a \vdash C}{\Gamma;\Delta \vdash (\nu a)C}$$

$$\text{T-Par}\ \frac{\Gamma;\Delta_1 \vdash C \qquad \Gamma;\Delta_2 \vdash \mathcal{D}}{\Gamma;\Delta_1, \Delta_2 \vdash C \parallel \mathcal{D}} \qquad \text{T-AP}\ \frac{p : \mathsf{AP}((\mathrm{p}_i : S_i)_{i\in I}) \in \Gamma \quad \{\{\mathrm{p}_i : S_i\}_{i\in I}\}\ \Delta \vdash \chi \quad \mathsf{comp}(\{\mathrm{p}_i : S_i\}_{i\in I})}{\Gamma;\Delta, p \vdash p(\chi)}$$

$$\text{T-Actor}\ \frac{\Gamma;\Delta_1 \mid A \vdash \mathcal{T} \quad \Gamma;\Delta_2 \mid A \vdash \sigma \qquad \Gamma;\Delta_3 \mid A \vdash \rho}{\Gamma;\Delta_1, \Delta_2, \Delta_3, a \vdash \langle a, \mathcal{T}, \sigma, \rho\rangle}$$

$$\text{T-EmptyQueue}\ \frac{}{\Gamma;s : \epsilon \vdash s \triangleright \epsilon} \qquad \text{T-ConsQueue}\ \frac{\Gamma \vdash V : A \qquad \Gamma;s : Q \vdash s \triangleright \sigma}{\Gamma;s : ((\mathrm{p}, \mathrm{q}, \ell(A)) \cdot Q) \vdash s \triangleright (\mathrm{p}, \mathrm{q}, \ell(V)) \cdot \sigma}$$

Access point typing $\boxed{\{(\mathrm{p}_i : S_i)_i\}\ \Delta \vdash \chi}$

$$\text{TA-Empty}\ \frac{}{\{(\mathrm{p}_i : S_i)_{i\in 1..n}\}\ \cdot \vdash \cdot} \qquad \text{TA-Entry}\ \frac{j \in I \qquad \{(\mathrm{p}_i : S_i)_{i\in I}\}\ \Delta \vdash \chi}{\{(\mathrm{p}_i : S_i)_{i\in I}\}\ \Delta, \widetilde{\iota^- : S_j} \vdash \chi[\mathrm{p}_j \mapsto \widetilde{\iota}]}$$

Thread state typing $\boxed{\Gamma;\Delta \mid A \vdash \mathcal{T}}$

$$\text{TT-Idle}\ \frac{\Gamma \vdash V : A}{\Gamma;\cdot \mid A \vdash \mathbf{idle}(V)} \qquad \text{TT-Sess}\ \frac{\Gamma \mid A \mid S \triangleright M : A \triangleleft \mathsf{end}}{\Gamma;s[\mathrm{p}] : S \mid A \vdash (M)^{s[\mathrm{p}]}} \qquad \text{TT-NoSess}\ \frac{\Gamma \mid A \mid \mathsf{end} \triangleright M : A \triangleleft \mathsf{end}}{\Gamma;\cdot \mid A \vdash M}$$

Handler state typing $\boxed{\Gamma;\Delta \mid A \vdash \sigma}$

$$\text{TH-Empty}\ \frac{}{\Gamma;\cdot \mid A \vdash \epsilon} \qquad \text{TH-Handler}\ \frac{\Gamma \vdash V : \mathsf{Handler}(S^?, A) \qquad \Gamma;\Delta \mid A \vdash \sigma}{\Gamma;\Delta, s[\mathrm{p}] : S^? \mid A \vdash \sigma[s[\mathrm{p}] \mapsto V]}$$

Initialisation state typing $\boxed{\Gamma;\Delta \mid A \vdash \rho}$

$$\text{TI-Empty}\ \frac{}{\Gamma;\cdot \mid A \vdash \epsilon} \qquad \text{TI-Callback}\ \frac{\Gamma \vdash V : A \xrightarrow[A]{S,\mathsf{end}} A \qquad \Gamma;\Delta \mid A \vdash \rho}{\Gamma;\Delta, \iota^+ : S \mid A \vdash \rho[\iota \mapsto V]}$$

Meta-level definitions

$$\mathsf{snames}(\Delta) \quad = \quad \{s \mid s : Q \in \Delta \vee \exists \mathrm{p}.(s[\mathrm{p}] \in \mathsf{dom}(\Delta))\}$$

Fig. 12. Typing of Configurations

handlers match the types in the runtime environments, and the initialisation state typing judgement $\Gamma;\Delta \mid C \vdash \rho$ ensures that all initialisation callbacks match the session type of the initialisation token. Finally, T-EmptyQueue and T-ConsQueue ensure that queued messages match the queue type.

4.2 Properties

With configuration typing defined, we can begin to describe the properties enjoyed by Maty.

4.2.1 Preservation. Typing is preserved by reduction; consequently we know that communication actions must match those specified by the session type. Full proofs can be found in Appendix B.

Theorem 4.2 (Preservation). *Typability is preserved by structural congruence and reduction.*

(≡) If $\Gamma;\Delta \vdash C$ *and* $C \equiv \mathcal{D}$ *then there exists some* $\Delta' \equiv \Delta$ *such that* $\Gamma;\Delta' \vdash \mathcal{D}$.

(→) If $\Gamma;\Delta \vdash C$ *with* $\mathit{safe}(\Delta)$ *and* $C \rightarrow \mathcal{D}$, *then there exists some* Δ' *such that* $\Delta \Longrightarrow^? \Delta'$ *where* $\mathit{safe}(\Delta')$ *and* $\Gamma;\Delta' \vdash \mathcal{D}$.

Remark 4.3 (Session Fidelity). Traditionally, *session fidelity* is presented as a property that all communication in a system conforms to its associated session type, i.e., that if a process performs a communication action then there is a corresponding (meta-theoretical) type reduction [15, 30].

Fidelity is often an implicit corollary of type preservation in works on functional session types (e.g., [23, 26, 28, 41]). Alternatively, session fidelity in [52] (and derived works) refer to session fidelity as the property that at least one typing context reduction can be reflected by a process. We follow the former definition and account for preservation through our preservation theorem.

4.2.2 Progress. In general, just because two protocols are individually deadlock-free *does not* mean that the system as a whole is deadlock-free, due to the possibility of inter-session deadlocks. For example, consider the following two trivially deadlock-free protocols:

$$\{\mathsf{p} : \mathsf{q}\,\&\{\ell_1(\mathsf{Unit})\,.\,\mathsf{end}\},\ \ \mathsf{q} : \mathsf{p}\oplus\{\ell_1(\mathsf{Unit})\,.\,\mathsf{end}\}\} \qquad \{\mathsf{r} : \mathsf{s}\,\&\{\ell_2(\mathsf{Unit})\,.\,\mathsf{end}\},\ \ \mathsf{s} : \mathsf{r}\oplus\{\ell_2(\mathsf{Unit})\,.\,\mathsf{end}\}\}$$

Even with an asynchronous semantics, a standard multiparty process calculus would admit the following deadlocking process, since each send is blocked by a receive:

$$s_1[\mathsf{p}][\mathsf{q}]\&\ell_1(x)\,.\,s_2[\mathsf{r}][\mathsf{s}]\oplus\ell_2(y)\,.\,\mathbf{0}\ \parallel\ s_2[\mathsf{s}][\mathsf{r}]\&\ell_2(a)\,.\,s_1[\mathsf{q}][\mathsf{p}]\oplus\ell_1(b)\,.\,\mathbf{0}$$

There are various approaches to ruling out inter-session deadlocks: some approaches restrict each subprocess to only play a single role in a single session (e.g., [52]); this would rule out the above example but is too restrictive for our setting. Other approaches (e.g., [15]) overlay additional interaction type systems to rule out inter-process deadlocks, again at the cost of expressiveness and type system complexity. Finally, logically-inspired approaches to multiparty session typing (e.g., [9]) treat sessions as monolithic processes $(\nu s)(P_1 \parallel \cdots \parallel P_n)$ that mean that such cycles cannot arise. Programming with such processes requires "multi-fork" style session initiations that combine channel- and process creation, and therefore are inapplicable to our programming model.

Maty *does not* suffer from inter-process deadlocks because of our event-driven programming style where although an actor is involved in many sessions at a time, only one is *active* at once, and code within handlers does not block. Since an actor yields and installs a handler whenever it needs to receive a message, the actor can then schedule any handler that has a waiting message.

To show this intuition formally, we start by classifying a *canonical form* for configurations.

Definition 4.4 (Canonical form). A configuration C is in *canonical form* if it can be written:

$$(\nu\tilde{\imath})(\nu p_{i\in 1..l})(\nu s_{j\in 1..m})(\nu a_{k\in 1..n})(p_i(\chi_i)_{i\in 1..l} \parallel (s_j \triangleright \delta_j)_{j\in 1..m} \parallel \langle a_k, \mathcal{T}_k, \sigma_k, \rho_k\rangle_{k\in 1..n})$$

Every well typed configuration can be written in canonical form; the result follows from the structural congruence rules and Theorem 4.2.

Compliance requires the session *types* in every session to satisfy progress. Due to our event-driven design, the property transfers to *configurations*: a non-reducing closed configuration cannot be blocked on any session communication and so cannot contain any sessions.

Progress states that since (by compliance) all protocols are deadlock-free, a configuration can either reduce, or it contains no sessions and no further sessions can be established.

Theorem 4.5 (Progress). *If $\cdot;\cdot \vdash C$, then either there exists some $\mathcal{D}$ such that $C \longrightarrow \mathcal{D}$, or C is structurally congruent to the following canonical form:*

$$(\nu\tilde{\imath})(\nu p_{i\in 1..m})(\nu a_{j\in 1..n})(p_1(\chi_1)_{i\in 1..m} \parallel \langle a_j, \mathbf{idle}(V_j), \epsilon, \rho_j\rangle_{j\in 1..n})$$

4.2.3 Global Progress. Assuming that actors only run *terminating* threads—a common assumption in event-driven systems—we actually obtain the stronger property of *global progress*, which ensures that communication can eventually happen in every active session.

We begin by defining configuration contexts $\mathcal{G} ::= [\,] \mid (\nu\alpha)\mathcal{G} \mid \mathcal{G} \parallel C \mid C \parallel \mathcal{G}$ that allow us to focus on a subconfiguration. We say that $\langle a, M, \sigma, \rho\rangle$ is an *actor subconfiguration* of a configuration C if $C = \mathcal{G}[\langle a, M, \sigma, \rho\rangle]$ for some configuration context $\mathcal{G}$.

Definition 4.6 (Active environment / session). We say that a runtime type environment Δ is *active*, written active(Δ), if it contains at least one entry of the form $s[\mathrm{p}] : S$ where $S \neq \mathsf{end}$.

Given a derivation $\Gamma;\Delta \vdash C$, let us write activeSessions(C) for the set of names of sessions in C typable under active environments. We now classify *thread-terminating* actors: actors that will eventually either suspend with a handler or fully evaluate to a value. A *thread reduction* for an actor a is a configuration reduction that affects the active thread of a.

Definition 4.7 (Thread Reduction). A reduction $C \longrightarrow \mathcal{D}$ where $C = \mathcal{G}_1[\langle a, \mathcal{M}[M], \sigma, \rho\rangle]$ and $\mathcal{D} = \mathcal{G}_2[\langle a, \mathcal{M}[N], \sigma', \rho'\rangle]$ is a *thread reduction for a* if $\langle a, \mathcal{M}[M], \sigma, \rho\rangle$ is a subconfiguration of the fired redex of C, and $\langle a, \mathcal{M}[N], \sigma', \rho'\rangle$ is a subconfiguration of its contractum.

A *maximal thread reduction* $C \longrightarrow^* \mathcal{D}$ for a is a sequence of thread reductions for a where there exist no further thread reductions for a from $\mathcal{D}$.

Definition 4.8 (Thread-Terminating). An actor subconfiguration $\langle a, \mathcal{T}, \sigma, \rho\rangle$ of a configuration $C = \mathcal{G}[\langle a, \mathcal{T}, \sigma, \rho\rangle]$ is *thread-terminating* if either $\mathcal{T} = \mathbf{idle}(V)$ for some V, or $\mathcal{T} = \mathcal{M}[M]$ such that there exists no infinite thread reduction for a from C.

We deliberately design our metatheory to be agnostic to the precise method used to ensure termination. Concretely, to ensure that actors are always thread-terminating, we could for example use straightforward type system restrictions like requiring all callbacks and handlers to be total or use primitive recursion. We could also use effect-based analyses (e.g. those used for ensuring safe database programming [40]). We conjecture we could also adapt type systems designed to enforce *fair termination* [14, 48]; we discuss this further in Section 7. The additional power of exceptions described in Section 5 would also allow the smooth integration of run-time termination analyses. All of the example callbacks and handlers discussed in the paper would preserve thread-termination.

Next, we show that reduction in one actor will never inhibit reduction in another. The result follows because all communication is asynchronous and (in part due to Theorem 4.5), given a well-typed configuration, all constructs occurring in an active thread can always reduce immediately.

Lemma 4.9 (Independence of Thread Reductions). *If $\cdot;\cdot \vdash C$ where $C = \mathcal{G}_1[\langle a, \mathcal{M}[M], \sigma, \rho\rangle]$ and $C \longrightarrow \mathcal{G}_2[\langle a, \mathcal{M}[N], \sigma', \rho'\rangle]$ is a thread reduction for a, then for every $\mathcal{D}$ and $\mathcal{G}_3$ such that $C \longrightarrow \mathcal{D}$ and $\mathcal{D} = \mathcal{G}_3[\langle a, \mathcal{M}[M], \sigma, \rho\rangle]$ it follows that $\mathcal{D} \longrightarrow \mathcal{G}_4[\langle a, \mathcal{M}[N], \sigma', \rho'\rangle]$ for some $\mathcal{G}_4$.*

Definition 4.10 (Idle Configuration). An actor subconfiguration $\langle a, \mathcal{T}, \sigma, \rho\rangle$ of a configuration C is *idle* if $\mathcal{T} = \mathbf{idle}(V)$ for some V. Configuration C is idle if all of its actor subconfigurations are idle.

It follows by typing and from Lemma 4.9 that every thread-terminating actor subconfiguration of a configuration C eventually evaluates to either $\mathbf{return}\ V$ or $\mathbf{suspend}\ V\ W$ and that (via E-Suspend or E-Return) C further evaluates to an idle configuration.

Corollary 4.11. *If $\cdot;\cdot \vdash C$ and C is thread-terminating, then $C \longrightarrow^* \mathcal{D}$ where $\mathcal{D}$ is idle.*

Finally, due to session typing and compliance, every active session in a well-typed idle configuration can reduce. The result follows as a special case of Theorem 4.5.

Lemma 4.12. *If $\cdot;\cdot \vdash C$ where C is idle, then for every $s \in$ activeSessions(C), $C \equiv (\nu s)\mathcal{D}$ and $\mathcal{D} \xrightarrow{s}$.*

Since (by Theorem 4.2) we can always use the structural congruence rules to hoist a session name restriction to the topmost level, global progress follows as an immediate corollary.

Corollary 4.13 (Global Progress). *If $\cdot;\cdot \vdash C$ where C is thread-terminating, then for every $s \in$ activeSessions(C), $C \equiv (\nu s)\mathcal{D}$ for some $\mathcal{D}$, and $\mathcal{D} \xrightarrow{\tau}{}^{*} \xrightarrow{s}$.*

Syntax

Types	A, B	::=	$\cdots \mid \mathsf{Pid}$	Monitored processes	ω	::=	$\widetilde{(a, V)}$
Values	V, W	::=	$\cdots \mid a$	Configurations	$\mathcal{C}, \mathcal{D}$	::=	$\cdots \mid \langle a, \mathcal{T}, \sigma, \rho, \omega\rangle$
Computations	M, N	::=	$\cdots \mid \mathbf{suspend}\ U\ V\ W$				$\mid\ \lightning a \mid \lightning s[\mathsf{p}] \mid \lightning \iota$
			$\mid\ \mathbf{monitor}\ V\ W \mid \mathbf{raise}$				

Modified typing rules for computations $\boxed{\Gamma \mid C \mid S \triangleright M : A \triangleleft T}$

$$\text{T-Spawn}\ \frac{\Gamma \mid A \mid \mathsf{end} \triangleright M : A \triangleleft \mathsf{end}}{\Gamma \mid C \mid S \triangleright \mathbf{spawn}\ M : \mathsf{Pid} \triangleleft S} \qquad \text{T-Suspend}\ \frac{\Gamma \vdash U : \mathsf{Handler}(S^?, C) \quad \Gamma \vdash V : C \quad \Gamma \vdash W : C \xrightarrow[C]{\mathsf{end},\mathsf{end}} C}{\Gamma \mid C \mid S^? \triangleright \mathbf{suspend}\ U\ V\ W : A \triangleleft T}$$

$$\text{T-Monitor}\ \frac{\Gamma \vdash V : \mathsf{Pid} \quad \Gamma \vdash W : C \xrightarrow[C]{\mathsf{end},\mathsf{end}} C}{\Gamma \mid C \mid S \triangleright \mathbf{monitor}\ V\ W : \mathsf{Unit} \triangleleft S} \qquad \text{T-Raise}\ \frac{}{\Gamma \mid C \mid S \triangleright \mathbf{raise} : A \triangleleft T}$$

Modified configuration reduction rules $\boxed{\mathcal{C} \xrightarrow{\iota} \mathcal{D}}$

$$\text{E-React}\quad \langle a, \mathbf{idle}(W), \sigma[s[\mathsf{p}] \mapsto (\mathbf{handler}\ \mathsf{q}\ st\ \{\overrightarrow{H}\}, U)], \rho, \omega\rangle \parallel s \triangleright (\mathsf{q}, \mathsf{p}, \ell(V)) \cdot \delta \xrightarrow{s} \langle a, (M\{V/x, W/st\})^{s[\mathsf{p}]}, \sigma, \rho, \omega\rangle \parallel s \triangleright \delta \quad \text{if } (\ell(V) \mapsto M) \in \overrightarrow{H}$$

$$\text{E-Spawn}\quad \langle a, \mathcal{M}[\mathbf{spawn}\ M], \sigma, \rho, \omega\rangle \xrightarrow{\tau} (\nu b)(\langle a, \mathcal{M}[\mathbf{return}\ b], \sigma, \rho, \omega\rangle \parallel \langle b, M, \epsilon, \epsilon, \emptyset\rangle)$$

$$\text{E-Suspend}\quad \langle a, (\mathcal{E}[\mathbf{suspend}\ U\ V\ W])^{s[\mathsf{p}]}, \sigma, \rho, \omega\rangle \xrightarrow{\tau} \langle a, \mathbf{idle}(V), \sigma[s[\mathsf{p}] \mapsto (U, W)], \rho, \omega\rangle$$

$$\text{E-Monitor}\quad \langle a, \mathcal{M}[\mathbf{monitor}\ b\ V], \sigma, \rho, \omega\rangle \xrightarrow{\tau} \langle a, \mathcal{M}[\mathbf{return}\ ()], \sigma, \rho, \omega \cup \{(b, V)\}\rangle$$

$$\text{E-InvokeM}\quad \langle a, \mathbf{idle}(W), \sigma, \rho, \omega \cup \{(b, V)\}\rangle \parallel \lightning b \xrightarrow{\tau} \langle a, (V\ W), \sigma, \rho, \omega\rangle \parallel \lightning b$$

$$\text{E-Raise}\quad \langle a, \mathcal{E}[\mathbf{raise}], \sigma, \rho, \omega\rangle \xrightarrow{\tau} \lightning a \parallel \lightning \sigma \parallel \lightning \rho$$

$$\text{E-RaiseS}\quad \langle a, (\mathcal{E}[\mathbf{raise}])^{s[\mathsf{p}]}, \sigma, \rho, \omega\rangle \xrightarrow{\tau} \lightning a \parallel \lightning s[\mathsf{p}] \parallel \lightning \sigma \parallel \lightning \rho$$

$$\text{E-CancelMsg}\quad s \triangleright (\mathsf{p}, \mathsf{q}, \ell(V)) \cdot \delta \parallel \lightning s[\mathsf{q}] \xrightarrow{\tau} s \triangleright \delta \parallel \lightning s[\mathsf{q}]$$

$$\text{E-CancelAP}\quad (\nu \iota)(p(\chi[\mathsf{p} \mapsto \widetilde{\iota'} \cup \{\iota\}]) \parallel \lightning \iota) \xrightarrow{\tau} p(\chi[\mathsf{p} \mapsto \widetilde{\iota'}])$$

$$\text{E-CancelH}\quad \langle a, \mathbf{idle}(W), \sigma[s[\mathsf{p}] \mapsto (\mathbf{handler}\ \mathsf{q}\ st\ \{\overrightarrow{H}\}, V)], \rho, \omega\rangle \parallel s \triangleright \delta \parallel \lightning s[\mathsf{q}] \xrightarrow{\tau} \langle a, (V\ W), \sigma, \rho, \omega\rangle \parallel s \triangleright \delta \parallel \lightning s[\mathsf{q}] \parallel \lightning s[\mathsf{p}] \quad \text{if } \mathsf{messages}(\mathsf{q}, \mathsf{p}, \delta) = \emptyset$$

where $\mathsf{messages}(\mathsf{p}, \mathsf{q}, \delta) = \{\ell(V) \mid (\mathsf{r}, \mathsf{s}, \ell(V)) \in \delta \wedge \mathsf{p} = \mathsf{r} \wedge \mathsf{q} = \mathsf{s}\}$

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

$$(\nu s)(\lightning s[\mathsf{p}_i]_{i \in 1..n} \parallel s \triangleright \epsilon) \parallel \mathcal{C} \equiv \mathcal{C}$$
$$(\nu a)(\lightning a) \parallel \mathcal{C} \equiv \mathcal{C}$$

Syntactic sugar

$$\lightning \sigma \triangleq \lightning s_1[\mathsf{p}_1] \parallel \cdots \parallel \lightning s_n[\mathsf{p}_n] \quad (\text{where } \mathsf{dom}(\sigma) = \{s_i[\mathsf{p}_i]\}_{i \in 1..n})$$
$$\lightning \rho \triangleq \lightning \iota_1 \parallel \cdots \parallel \lightning \iota_n \quad (\text{where } \mathsf{dom}(\rho) = \{\iota_i\}_{i \in 1..n})$$

Fig. 13. Maty$_{\lightning}$: Modified syntax and reduction rules

5 FAILURE HANDLING AND SUPERVISION

A major factor in the success of actor languages is their support for the *let-it-crash* philosophy: actors encountering errors should crash and be restarted by a supervisor actor. So far, we have not accounted for failure. A crashed actor cannot send messages, so we need a mechanism to prevent sessions from getting 'stuck'. Our solution builds on *affine sessions* [23, 28, 38, 44]: the core idea is that a role can be marked as cancelled, preventing further participation. Trying to receiving from a cancelled participant when there are no pending messages in the queue raises an exception, triggering a crash and propagating the failure.

Figure 13 shows the additional syntax, typing rules, and reduction rules needed for supervision and cascading failure; we call this extension Maty$_{\lightning}$. We make actors *addressable*, so **spawn** returns a process identifier (PID) of type Pid. The **monitor** $V\ W$ construct installs a callback function W to be evaluated should the actor referred to by V crash. The **raise** construct signifies a user-level error has occurred, for example **if** fileExists(*path*)) **then** processFile(*path*) **else raise**. Raising an exception causes an actor to crash and cancels all the sessions in which it is involved. The **raise** construct, like **suspend**, can be given an arbitrary return type and post-condition since it does not

return a value to a calling context. We also modify the **suspend** construct to take an additional callback to be run if the sender fails and the message is never sent; a sensible piece of syntactic sugar would be **suspend** $V\ W \triangleq$ **suspend** $V\ W\ (\lambda st.\ \textbf{raise})$ to propagate the failure.

We can make our shop actor robust by using a shopSup actor that restarts it upon failure:

$$\text{shopSup} \triangleq \lambda custAP.\textbf{monitor}\ (\textbf{spawn}\ \text{shop}\ (custAP, \text{initialStock}))\ (\text{shopSup}\ custAP)$$

The shopSup actor spawns a shop actor and monitors the resulting PID. Any failure of the shop actor will be detected by the shopSup which will restart the actor and monitor it again. The restarted shop actor will re-register with the access points and can then take part in subsequent sessions.

Configurations. To capture the additional runtime behaviour we need to extend the language of configurations. The actor configuration becomes $\langle a, \mathcal{T}, \sigma, \rho, \omega \rangle$, where ω pairs monitored PIDs with callbacks to be evaluated should the actor crash. We also introduce three kinds of "zapper thread", $\lightning a$, $\lightning s[\mathrm{p}]$, $\lightning \iota$ to indicate the cancellation of an actor, role, or initialisation token respectively.

Reduction rules by example. Consider the supervised Shop example after the Customer has sent a Checkout request and is awaiting a response. Instead of suspending to handle the request, the Shop raises an exception. This scenario can be represented by the following configuration, where *shop*, *cust*, and *pp* are actors playing the Shop, Customer, and PaymentProcessor in session s, and *sup* is monitoring *shop*:

$$(\nu sup)(\nu shop)(\nu cust)(\nu pp)(\nu s)\left(\begin{array}{l}\langle shop, (\textbf{raise})^{s[\mathrm{Shop}]}, \epsilon, \epsilon, \epsilon\rangle \\ \parallel \langle cust, \textbf{idle}(()), s[\mathrm{Customer}] \mapsto (\text{checkoutHandler}, \textbf{raise}), \epsilon, \epsilon\rangle \\ \parallel \langle pp, \textbf{idle}(()), s[\mathrm{PaymentProcessor}] \mapsto (\text{buyHandler}, \textbf{raise}), \epsilon, \epsilon\rangle \\ \parallel s \triangleright (\mathrm{Customer}, \mathrm{Shop}, \text{checkout}(([123], 510))) \\ \parallel \langle sup, \textbf{idle}(()), \epsilon, \epsilon, (shop, \text{shopSup}\ cAP)\rangle\end{array}\right)$$

For brevity we shorten Shop, Customer, and PaymentProcessor to S, C, and PP respectively. We let configuration context $\mathcal{G} = (\nu sup)(\nu shop)(\nu cust)(\nu pp)(\nu s)([\,] \parallel \langle sup, \textbf{idle}(()), \epsilon, \epsilon, (shop, \text{shopSup}\ cAP)\rangle)$.

Since the *shop* actor is playing role $s[\mathrm{S}]$ and raising an exception, by E-RaiseS the actor is replaced with zapper threads $\lightning shop$ and $\lightning s[\mathrm{S}]$.

$$\mathcal{G}\left[\begin{array}{l}\langle shop, (\textbf{raise})^{s[\mathrm{S}]}, \epsilon, \epsilon, \epsilon\rangle \\ \parallel \langle cust, \textbf{idle}(()), s[\mathrm{C}] \mapsto (\text{checkoutHandler}, \textbf{raise}), \epsilon, \epsilon\rangle \\ \parallel \langle pp, \textbf{idle}(()), s[\mathrm{PP}] \mapsto (\text{buyHandler}, \textbf{raise}), \epsilon, \epsilon\rangle \\ \parallel s \triangleright (\mathrm{C}, \mathrm{S}, \text{checkout}(([123], 510)))\end{array}\right] \longrightarrow \mathcal{G}\left[\begin{array}{l}\lightning shop \parallel \lightning s[\mathrm{S}] \\ \parallel \langle cust, \textbf{idle}(()), s[\mathrm{C}] \mapsto (\text{checkoutHandler}, \textbf{raise}), \epsilon, \epsilon\rangle \\ \parallel \langle pp, \textbf{idle}(()), s[\mathrm{PP}] \mapsto (\text{buyHandler}, \textbf{raise}), \epsilon, \epsilon\rangle \\ \parallel s \triangleright (\mathrm{C}, \mathrm{S}, \text{checkout}(([123], 510)))\end{array}\right]$$

Next, since $s[\mathrm{S}]$ has been cancelled, the checkout message can never be received and so is removed from the queue (E-CancelMsg). Similarly since both C and PP are waiting for messages from cancelled role S, they both evaluate their failure computations, **raise** (E-CancelH). In turn this results in the cancellation of the *cust* and *pp* actors, and the $s[\mathrm{C}]$ and $s[\mathrm{PP}]$ endpoints (E-RaiseS).

$$\longrightarrow^{+} \mathcal{G}\left[\begin{array}{l}\lightning shop \parallel \lightning s[\mathrm{S}] \\ \parallel \langle cust, \textbf{idle}(()), (\textbf{raise})^{s[\mathrm{C}]}, \epsilon, \epsilon\rangle \\ \parallel \langle pp, \textbf{idle}(()), (\textbf{raise})^{s[\mathrm{PP}]}, \epsilon, \epsilon\rangle \\ \parallel s \triangleright \epsilon\end{array}\right] \longrightarrow^{+} \mathcal{G}\left[\ \lightning shop \parallel \lightning s[\mathrm{S}] \parallel \lightning cust \parallel \lightning s[\mathrm{C}] \parallel \lightning pp \parallel \lightning s[\mathrm{PP}] \parallel s \triangleright \epsilon\ \right]$$

At this point the session has failed and can be garbage collected, leaving the supervisor actor and the zapper thread for *shop*. Since the supervisor was monitoring *shop*, which has crashed, the monitor callback is invoked (E-InvokeM) which finally re-spawns and monitors the Shop actor.

$$\longrightarrow (\nu shop)(\nu sup)\left(\begin{array}{l}\lightning shop \\ \parallel \langle sup, \text{shopSup}\ cAP\ (), \epsilon, \epsilon, \epsilon\rangle\end{array}\right) \longrightarrow^{+} (\nu shop')(\nu sup)\left(\begin{array}{l}\langle shop', \text{shop}\ (cAP, \text{initialStock}), \epsilon, \epsilon, \epsilon\rangle \\ \parallel \langle sup, \textbf{idle}(()), \epsilon, \epsilon, (shop', \text{shopSup}\ cAP)\rangle\end{array}\right)$$

5.1 Metatheory

All metatheoretical results continue to hold. Figure 14 shows the necessary modifications to the configuration typing rules and type LTS. We extend runtime type environments to *cancellation-aware* environments Φ that include an additional entry of the form $s[\mathrm{p}] : \lightning$, denoting that endpoint $s[\mathrm{p}]$ has been cancelled. We also need to extend the type LTS to account for failure propagation;

Runtime syntax

Cancellation-aware runtime envs. $\Phi ::= \cdot \mid \Phi, p \mid \Phi, \iota^{\pm} : S \mid \Phi, s[\mathrm{p}] : S \mid \Phi, s[\mathrm{p}] : \lightning \mid \Phi, s : Q$

Labels $\gamma ::= \cdots \mid \lightning\, s[\mathrm{p}] \mid s : \mathrm{p} \lightning \mathrm{q} :: \ell \mid s : \mathrm{p} \lightning \mathrm{q}$

Modified typing rules for configurations $\boxed{\Gamma;\Phi \vdash C}$ $\boxed{\Gamma;\Phi \mid \sigma \vdash}$

$$\textsc{T-ActorName}\ \frac{\Gamma, a : \mathsf{Pid}; \Phi, a \vdash C}{\Gamma;\Phi \vdash (\nu a) C} \qquad \textsc{T-ZapActor}\ \frac{}{\Gamma; a \vdash \lightning\, a} \qquad \textsc{T-ZapRole}\ \frac{}{\Gamma; s[\mathrm{p}] : \lightning \vdash \lightning\, s[\mathrm{p}]} \qquad \textsc{T-ZapTok}\ \frac{}{\Gamma; \iota^{+} : S \vdash \lightning\, \iota}$$

$$\textsc{T-Actor}\ \frac{\Gamma;\Phi_1 \mid C \vdash \mathcal{T} \qquad \Gamma;\Phi_2 \mid C \vdash \sigma \qquad \Gamma;\Phi_3 \mid C \vdash \rho \qquad \forall (b, V) \in \omega.\ \Gamma \vdash b : \mathsf{Pid} \wedge \Gamma \vdash V : C \xrightarrow[C]{\text{end,end}} C}{\Gamma;\Phi_1, \Phi_2, \Phi_3, a \vdash \langle a, \mathcal{T}, \sigma, \rho, \omega \rangle}$$

$$\textsc{TH-Handler}\ \frac{\Gamma \vdash V : \mathsf{Handler}(S^{?}, C) \qquad \Gamma \vdash W : C \xrightarrow[C]{\text{end,end}} C \qquad \Gamma;\Phi \mid C \vdash \sigma}{\Gamma;\Phi, s[\mathrm{p}] : S^{?} \mid C \vdash \sigma[s[\mathrm{p}] \mapsto (V, W)]}$$

Additional LTS rules $\boxed{\Phi \xrightarrow{\gamma} \Phi'}$ $\boxed{\Phi \overset{s[\mathrm{p}]}{\rightsquigarrow\!\triangleright} \Phi}$

$$\textsc{Lbl-ZapMsg}\quad \Phi, s[\mathrm{q}] : \lightning, s : (\mathrm{p}, \mathrm{q}, \ell(A)) \cdot Q \xrightarrow{s:\mathrm{p}\lightning\mathrm{q}::\ell} \Phi, s[\mathrm{q}] : \lightning, s : Q$$

$$\textsc{Lbl-ZapRecv}\quad \Phi, s[\mathrm{p}]{:}\mathrm{q} \,\&\, \{\ell_i(A_i).S_i\}_{i\in I}, s[\mathrm{q}]{:}\lightning, s{:}Q \xrightarrow{s:\mathrm{p}\lightning\mathrm{q}} \Phi, s[\mathrm{p}]{:}\lightning, s[\mathrm{q}]{:}\lightning, s{:}Q \quad (\text{if } \mathsf{messages}(\mathrm{q}, \mathrm{p}, Q) = \emptyset)$$

$$\textsc{Lbl-Zap}\quad \Phi, s[\mathrm{p}] : S \overset{s[\mathrm{p}]}{\rightsquigarrow\!\triangleright} \Phi, s[\mathrm{p}] : \lightning$$

Fig. 14. $\mathsf{Maty}_{\lightning}$: Modified configuration typing rules and type LTS

we take a similar approach to Barwell et al. [6]. Rule Lbl-Zap accounts for the possibility that in any given reduction step, a role may be cancelled (for example, as a result of E-RaiseS), but it is a separate relation since it is unnecessary for determining behavioural properties of types.

5.1.1 Preservation. We need a slightly modified preservation theorem in order to account for cancelled roles; specifically we write $\Rrightarrow$ for the relation $\Longrightarrow^{?}\rightsquigarrow\!\triangleright^{*}$. The safety property is unchanged for cancellation-aware environments.

Theorem 5.1 (Preservation ($\longrightarrow$, $\mathsf{Maty}_{\lightning}$)). *If $\Gamma;\Phi \vdash C$ with safe(Φ) and $C \longrightarrow \mathcal{D}$, then there exists some Φ' such that $\Phi \Rrightarrow \Phi'$ and safe(Φ') and $\Gamma;\Phi' \vdash \mathcal{D}$.*

5.1.2 Progress. $\mathsf{Maty}_{\lightning}$ enjoys progress since E-CancelMsg discards messages that cannot be received, and E-CancelMsg invokes the failure continuation whenever a message will never be sent due to a failure. Monitoring is orthogonal. The one change is that zapper threads for actors may remain if the actor name is free in an existing monitoring or initialisation callback. We require a slightly-adjusted deadlock-freedom property and canonical form to account for session failure.

Definition 5.2 (Deadlock-freedom and compliance ($\mathsf{Maty}_{\lightning}$)). A runtime environment Φ is *deadlock-free*, written $\mathsf{df}_{\lightning}(\Phi)$, if $\Phi \Longrightarrow^{*} \Phi' \not\Longrightarrow$ implies that either $\Phi' = s : \epsilon$ or $\Phi' = (s[\mathrm{p}_i] : \lightning)_{i\in I}, s : \epsilon$.

A runtime environment Φ is *compliant*, written $\mathsf{comp}_{\lightning}(\Phi)$, if safe($\Phi$) and $\mathsf{df}_{\lightning}(\Phi)$.

Definition 5.3. A $\mathsf{Maty}_{\lightning}$ configuration C is in *canonical form* if it can be written:

$$(\nu \tilde{\iota})(\nu p_{i\in 1..l})(\nu s_{j\in 1..m})(\nu a_{k\in 1..n})(p_i(\chi_i)_{i\in 1..l} \parallel (s_j \triangleright \delta_j)_{j\in 1..m} \parallel \langle a_k, \mathcal{T}_k, \sigma_k, \rho_k, \omega_k \rangle_{k\in 1..n'-1} \parallel \widetilde{\lightning\, \alpha})$$

with $(\lightning\, a_k)_{k\in n'..n}$ contained in $\widetilde{\lightning\, \alpha}$.

Theorem 5.4 (Progress ($\mathsf{Maty}_{\lightning}$)). *If $\cdot;\cdot \vdash C$, then either there exists some $\mathcal{D}$ such that $C \longrightarrow \mathcal{D}$, or C is structurally congruent to the following canonical form:*

$$(\nu \tilde{\iota})(\nu p_{i\in 1..m})(\nu a_{j\in 1..n})(p_1(\chi_1)_{i\in 1..m} \parallel \langle a_j, \mathbf{idle}(U_j), \epsilon, \rho_j, \omega_j \rangle_{j\in 1..n'-1} \parallel (\lightning\, a_j)_{j\in n'..n})$$

5.1.3 Global Progress. A modified version of global progress holds: in every active session, after a number of reductions each session will either be cancelled or perform a communication action.

Theorem 5.5 (Global Progress (Maty$_{\not\downarrow}$)). *If $\cdot;\cdot \vdash C$ where C is thread-terminating, then for every $s \in \mathit{activeSessions}(C)$, then there exist $\mathcal{D}$ and $\mathcal{D}_1$ such that $C \equiv (\nu s)\mathcal{D}$ where $\mathcal{D} \xrightarrow{\tau}{}^{*} \mathcal{D}_1$ and either $\mathcal{D}_1 \xrightarrow{s}$, or $\mathcal{D}_1 \equiv \mathcal{D}_2$ for some $\mathcal{D}_2$ where $s \notin \mathit{activeSessions}(\mathcal{D}_2)$.*

5.2 Discussion

The semantics of **raise** follows the Erlang "let it crash" methodology that favours crashing upon errors over defensive programming. However, cancellation is flexible enough to support other failure-handling strategies: we can for example implement a **leave** V construct, that allows an actor to exit a session and update its state to V *without* terminating, using the following reduction rule:

$$\langle a, (\mathcal{E}[\textbf{leave}\ V])^{s[\mathrm{p}]}, \sigma, \rho, \omega\rangle \longrightarrow \langle a, \textbf{idle}(V), \sigma, \rho, \omega\rangle \parallel \not\downarrow s[\mathrm{p}]$$

Maty's combination of event-driven concurrency and cancellation also makes handling timeouts straightforward. We could for example extend the **suspend** $U\ V\ W$ construct to **suspend** $U\ V$ t W, where t is some deadline and invoke the failure-handling computation if the deadline is missed. The failure-handling callback could then e.g. either retry or raise an exception. Indeed, Hou et al. [31] show how session cancellation can be used to enable flexible timed session types and we expect that their results could be incorporated into our design.

6 IMPLEMENTATION AND EVALUATION

6.1 Implementation

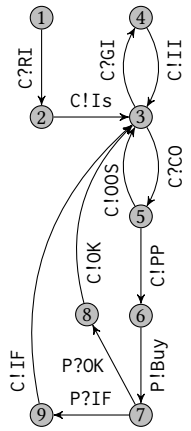
Based on our formal design, we have implemented a toolchain for Maty-style event-driven actor programming in Scala. It adopts the state machine based API generation approach of Scribble [33]:

(1) The user specifies *global types* in the Scribble protocol description language [56].
(2) Our toolchain internally uses Scribble to validate global types according to the MPST-based safety conditions, *project* them to local types for each role, and construct a representation of each local type based on *communicating finite state machines* (CFSM) [8].
(3) From each CFSM, the toolchain generates a typed, *protocol-and-role-specific* API for the user to implement that role as an event-driven Maty actor in native Scala.

Typed APIs for Maty actor programming. Consider the Shop role in our running example (Fig. 5). Fig. 15 shows the CFSM for Shop (with abbreviated message labels) and a summary of the main generated types and operations (omitting the type annotations for the `sid` and `pay` parameters). The toolchain generates Scala types for each CFSM state: non-blocking states (sends or suspends) are coloured blue, whereas blocking states (inputs) are red.

Non-blocking state types provide methods for outputs and suspend actions, with types specific to each state. The return type corresponds to the successor state type, enabling chaining of session actions: e.g., state type `S2` has method `Customer_sendItems` for the transition `C!Is`. The successor state type `S3Suspend` includes a `suspend` method to install a handler for the input event of state 3, and to yield control back to the event loop. The `Done.type` type ensures that each handler must either complete the protocol or perform a `suspend`. Input state types are traits implemented by case classes generated for each input message. The event loop calls the user-specified handler with the corresponding case class upon each input event, with each case class carrying an instance of the successor state type. For example, `S3` (state 3) is implemented by case classes `GetItemInfo` and `Checkout` for its input transitions, which respectively carry instances of successor states `S4` and `S5`.

The API guides the user through the protocol to construct a Maty actor with compatible handlers for every possible input event. For example, Fig. 16 handles state `S1` and could be safely supplied to the `suspend` method of `S1Suspend` immediately following a new session initiation. It *further* handles `S3` (so could also be supplied to `S3Suspend`), where the shop receives either `GetItemInfo` or `Checkout`.



State	State types	**Methods** (send, suspend) or **Input cases** (extends state type trait)
1	S1Suspend	`suspend[D](d: D, f: (D, S1) => Done.type): Done.type`
	S1	`case class RequestItems(sid, pay, succ: S2) extends S1`
2	S2	`Customer_sendItems(pay: ItemList): S3Suspend`
3	S3Suspend	`suspend[D](d: D, f: (D, S3) => Done.type): Done.type`
	S3	`case class GetItemInfo(sid, pay, succ: S4) extends S3`
		`case class Checkout(sid, pay, succ: S5) extends S3`
4	S4	`Customer_sendItemInfo(pay): S3Suspend`
5	S5	`Customer_sendProcessingPayment(): S6`
		`Customer_sendOutOfStock(): S3Suspend`
6	S6	`PaymentProcessor_sendBuy(pay): S7Suspend`
7	S7Suspend	`suspend[D](d: D, f: (D, S7) => Done.type): Done.type`
	S7	`case class OK(sid, pay, succ: S8) extends S7`
		`case class InsufficientFunds(sid, pay, succ: S9) extends S7`
8	S8	`Customer_sendOK(pay): S3Suspend`
9	S9	`Customer_sendInsufficientFunds(pay): S3Suspend`

Fig. 15. (left) CFSM for the Shop role in the Customer-Shop-PaymentProcessor protocol, and (right) summary of state types and methods in the toolchain-generated Scala API for this role.

```
// d can be used for internal, _session-specific_ actor data
def custReqHandler[T: S1orS3](d: DataS, s: T): Done.type = {
  s match {
    case c: S1 => c match {
      // pay is message payload; succ is successor state
      case RequestItems(sid, pay, succ) =>
        succ.Customer_sendItems(d.summary())
            .suspend(d, custReqHandler[S3]) }
    case c: S3 => c match {
      case GetItemInfo(sid, pay, succ) =>
        succ.Customer_sendItemInfo(d.lookupItem(pay))
            .suspend(d, custReqHandler[S3])
      case Checkout(sid, pay, succ) =>
        if (d.inStock(pay)) {
          succ.Customer_sendProcessingPayment()
              .PaymentProcessor_sendBuy(d.total(pay))
              .suspend(d, paymentResponseHandler)
    // ...continuing on from the left column
    } else {
      val sus = succ.Customer_sendOutOfStock()
      // d.staff: LOption[R1] -- this is a..
      // .."frozen" instance of state type R1
      d.staff match {
        // R1 is the Restock protocol state type
        case x: LSome[R1] =>
          ibecome(d, x, restockHndlr)
        case _: LNone =>
          // Error handling
          throw new RuntimeException
      }
      sus.suspend(d, custReqHandler[S3])
    }
}}}
```

Fig. 16. Example handler code from a Maty actor implemented in Scala using the toolchain-generated API

The runtime for our APIs executes sessions over TCP and uses the Java NIO library to run the actor event loops. It supports *fully distributed* sessions between remote Maty actors.

Switching between sessions. As well as supporting the core features and failure handling capabilities of Maty, our implementation also includes the ability to proactively *switch* between sessions. Figure 16 shows how this functionality can be used to switch into a long-running `Restock` session when more stock is needed. For this purpose, the API allows the user to "freeze" unused state type instances as a type `LOption[S]` and resume them later by an inline `ibecome`. It allows the callback for a session switching behaviour to be performed inline with the currently active handler.

Discussion. Following our formal model, our generated APIs support a conventional style of actor programming where non-blocking operations are programmed in direct-style, in contrast to approaches that invert both input *and* output actions [54, 57] through the event loop.

Static Scala typing ensures that handlers safely handle all possible input events at every stage (by exhaustive matching of case classes), and that state types offer only the permitted operations at each state (by method typing). Our API design requires *linear* usage of state type objects (e.g., `s` and `succ`) and frozen session instances. Following other works [11, 33, 47, 51, 55], we check linearity in a hybrid fashion: the `Done` return types in Fig. 15 statically require `suspend` to be invoked at least

Table 1. Selected case studies, examples from Savina, and key features of their Maty programs.

	MPST(s)			Maty actor programs								
	⊕/&	μ	C/P	mSA	mRA	PP	dSp	dTo	mAP	dAP	be	self
Shop (Fig. 6)	✓	✓		✓	✓	✓			✓			
ShopRestock (Fig. 16)	✓	✓		✓	✓	✓			✓		✓	
Robot [22]	✓			✓		✓	✓	(✓)				
Chat [21]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
Ping-self [36]	✓	✓	✓	✓	✓				✓		✓	✓
Ping [36]	✓	✓										
Fib [35]				✓	✓	✓	✓	✓	✓	✓	✓	
Dining-self [36]	✓	✓	✓	✓	✓	✓	✓	(✓)	✓		✓	✓
Dining [36]	✓	✓		✓	✓	✓	✓	(✓)	✓			
Sieve [36]	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	

⊕/& = Branch type(s) μ = Recursive type(s) C/P = Concurrent/Parallel types mSA = Multiple sessions/actor
mRA = Multiple roles/actor PP = Parameterised number of actors dSp = Dynamic actor spawning
dTo = Dynamic topology mAP = Multiple APs dAP = Dynamic AP creation be = `ibecome` self = Self communication

once, but our APIs rule out multiple uses *dynamically*. We exploit our formal support for failure handling (Sec. 5) to treat dynamic linearity errors as failures and retain safety and progress.

In summary, our toolchain enables Scala programming of Maty actors that support concurrent handling of multiple heterogeneously-typed sessions, and ensures their safe execution. A statically well-typed actor will *never* select an unavailable branch or send/receive an incompatible payload type, and an actor system will *never* become stuck due to mismatching I/O actions. As in the theory, the system without `ibecome` will enjoy global progress provided every handler is terminating (e.g., by avoiding general recursion/infinite iteration). Although we make no formal claims about the system *with* `ibecome`, we conjecture that it will also enjoy progress up-to re-invocation of frozen sessions stored in the actor's state.

6.2 Evaluation

Table 1 summarises selected examples from the Savina [36] benchmark suite (lower) and larger case studies (upper); Appendix A contains sequence diagrams for the larger examples. Notably, key design features of Maty, e.g. support for handling multiple sessions per actor (mSA) and implementing multiple protocols/roles within actors (mRA), are crucial to expressing many concurrency patterns. For example, the Shop actor in both Shop examples plays the distinct Shop roles in the main Shop protocol and Shop-Staff protocol simultaneously, and handles these sessions concurrently.

The "-self" versions of Ping and Dining are versions faithful to the original Akka programs that involve internal coordination using `self ! msg` operations, but our APIs can express equivalent behaviour more simply without needing self-communication.

The (✓) distinguishes simpler forms of dynamic topologies (dTo) due to a parameterised number of clients dynamically connecting to a central server, from richer structures such as the parent-children tree topology dynamically created in Fib and the user-driven dynamic connections between clients and chat rooms in Chat; note both the latter involve dynamic access point creation (dAP).

Robot coordination. In this scenario, a real-world factory use case from Actyx AG [1] that was originally described by Fowler et al. [22], multiple Robots access a Warehouse with a single door. Only one Robot is allowed in the warehouse at a time. Concretely, each Robot actor establishes a separate session with the Door and Warehouse actors. Maty's event-driven model allows the Door and Warehouse to each be implemented as a *single* actor that can safely handle the concurrent interleavings of events across *any number* (PP, dSP) of separate Robot sessions (mSA).

Chat server. This use case [21] involves an arbitrary number of Clients (PP) using a Registry to create new chat Rooms, and to dynamically join and leave any existing Room. We model each Client, the Registry and each Room as separate actors. Rooms are created by spawning new Room actors (dSp) with fresh access points (dAP, mAP), and we allow any Client to establish sessions

with the Registry or any Room asynchronously (dTo). We decompose the Client-Registry and the Client-Room interactions into separate protocols (C/P, mAP), and use `ibecome` (be) in the Room actor to broadcast chat messages to all Clients currently in that Room.

7 RELATED WORK

Several works have investigated event-driven session typing. Zhou et al. [57] introduce a multiparty session type discipline that supports statically-checked refinement types, implemented in F★; to avoid needing to reason about linearity, users implement callbacks for each send and receive action. This approach is used by Miu et al. [42] for session-typed web applications, and by Thiemann [54] in Agda [46]. In contrast, our approach only yields control to the event loop on actor *receives*, as in idiomatic actor programming. Hu et al. [32] and Kouzapas et al. [37] introduced a binary session π-calculus with primitives used to implement an event loop; our work instead encodes an event loop directly in the semantics. Viering et al. [55] use event-driven programming in a framework for fault-tolerant session-typed distributed programming. Their model involves inversion of control on *output* as well as input events. They establish a version of global progress for a system of subsessions spawned in a tree hierarchy. By contrast, we establish our global progress property for every session in the system. These works all focus on process calculi rather than language design.

Ciccone et al. [14] developed *fair termination* for *synchronous* multiparty sessions, a strong property that subsumes our global progress: it implies every *role* fairly terminates, whereas our coarser-grained property is per *session*. Padovani and Zavattaro [48] developed fair termination for asynchronous *binary* sessions and show that fair termination implies orphan message freedom [13]. Our system ensures orphan message freedom for terminated multiparty sessions (as in [17, 19]), but we do not aim to restrict Maty to terminating sessions. We may be able to strengthen our formal results by adapting the fairness conditions discussed in [14, 48] (developed for session π-calculi) to our event-driven actor setting; however, features such as combining session creation and parallel composition into one term (based on linear logic) are more restrictive than in our model.

Mostrous and Vasconcelos [43] were first to investigate session typing for actors, using Erlang's unique reference generation and selective receive to impose a channel-based communication model. Their approach remains unimplemented and only supports binary session types. Francalanza and Tabone [25] implement binary session typing in Elixir using pre- and post-conditions on module-level functions, but their approach can only reason about interactions between *pairs* of participants. Our approach is inspired by the model introduced by Neykova and Yoshida [45] (later implemented in Erlang [21]), but our language design supports *static* checking and is formalised. EnsembleS [28] enforces session typing using a flow-sensitive effect system, focusing on supporting safe *adaptive* systems. However, each EnsembleS actor can only take part in a single session at a time.

Mailbox types [16, 22] capture the expected contents of a mailbox as a commutative regular expression, and ensure that processes do not receive unexpected messages. Mailbox and session types both aim to ensure safe communication but address different problems: session types suit *structured* interactions among known participants, whereas mailbox types are better when participants are unknown and message ordering is unimportant. Mailbox types cannot yet handle failure.

Castellani et al. [10] developed *internal delegation* where channels can be migrated within a multiparty session. Our actor model hides channels; however, internal delegation may provide a way to formally relate our model to session π-calculi via encodings (cf. [37]). Barbanera et al. [4, 5] emphasize simplified, *compositional* models of multiparty sessions. It would be interesting to formally compare their approach, based on parallel composition and forwarding, against our model, where a single-threaded actor can embed handlers for any number of concurrent sessions.

Scalas et al. [53] introduce a behavioural type system with dependent function types, allowing functions to be checked against interaction patterns written in a type-level DSL, enabling verification

of properties such as liveness and termination. Their behavioural type discipline is different to session typing (e.g., supporting parameterised server interactions but not branching). Our session-based approach is designed for structured interactions among known participants, and it is unclear how their actor API would scale to processes handling multiple session-style interactions.

8 CONCLUSION AND FUTURE WORK

This paper introduces Maty, an actor language that rules out communication mismatches and deadlocks using *multiparty session types*. Key to our approach is a novel combination of a flow-sensitive effect system and first-class message handlers. We have extended Maty with the ability to switch between sessions and recover from failures. In future it would be interesting to investigate path-dependent types in our implementation.

DATA AVAILABILITY STATEMENT

We will submit our implementation and examples as an artifact, and will upload the extended version of the paper with full proofs to arXiv upon acceptance.

REFERENCES

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

[2] Gul A. Agha. 1990. *ACTORS - a model of concurrent computation in distributed systems.* MIT Press.

[3] Robert Atkey. 2009. Parameterised notions of computation. *J. Funct. Program.* 19, 3-4 (2009), 335–376.

[4] Franco Barbanera, Viviana Bono, and Mariangiola Dezani-Ciancaglini. 2025. Open compliance in multiparty sessions with partial typing. *J. Log. Algebraic Methods Program.* 144 (2025), 101046. https://doi.org/10.1016/J.JLAMP.2025.101046

[5] Franco Barbanera, Mariangiola Dezani-Ciancaglini, Lorenzo Gheri, and Nobuko Yoshida. 2023. Multicompatibility for Multiparty-Session Composition. In *International Symposium on Principles and Practice of Declarative Programming, PPDP 2023, Lisboa, Portugal, October 22-23, 2023*, Santiago Escobar and Vasco T. Vasconcelos (Eds.). ACM, 2:1–2:15. https://doi.org/10.1145/3610612.3610614

[6] Adam D. Barwell, Alceste Scalas, Nobuko Yoshida, and Fangyi Zhou. 2022. Generalised Multiparty Session Types with Crash-Stop Failures. In *CONCUR (LIPIcs, Vol. 243)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 35:1–35:25.

[7] Lorenzo Bettini, Sara Capecchi, Mariangiola Dezani-Ciancaglini, Elena Giachino, and Betti Venneri. 2008. Session and Union Types for Object Oriented Programming. In *Concurrency, Graphs and Models, Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday (Lecture Notes in Computer Science, Vol. 5065)*, Pierpaolo Degano, Rocco De Nicola, and José Meseguer (Eds.). Springer, 659–680. https://doi.org/10.1007/978-3-540-68679-8_41

[8] Daniel Brand and Pitro Zafiropulo. 1983. On Communicating Finite-State Machines. *J. ACM* 30, 2 (apr 1983), 323?342. https://doi.org/10.1145/322374.322380

[9] Marco Carbone, Sam Lindley, Fabrizio Montesi, Carsten Schürmann, and Philip Wadler. 2016. Coherence Generalises Duality: A Logical Explanation of Multiparty Session Types. In *CONCUR (LIPIcs, Vol. 59)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 33:1–33:15.

[10] Ilaria Castellani, Mariangiola Dezani-Ciancaglini, Paola Giannini, and Ross Horne. 2020. Global types with internal delegation. *Theor. Comput. Sci.* 807 (2020), 128–153. https://doi.org/10.1016/J.TCS.2019.09.027

[11] David Castro-Perez, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng, and Nobuko Yoshida. 2019. Distributed programming using role-parametric session types in go: statically-typed endpoint APIs for dynamically-instantiated communication structures. *Proc. ACM Program. Lang.* 3, POPL (2019), 29:1–29:30. https://doi.org/10.1145/3290342

[12] Avik Chaudhuri. 2009. A Concurrent ML Library in Concurrent Haskell. In *ICFP*. ACM.

[13] Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, Alceste Scalas, and Nobuko Yoshida. 2017. On the Preciseness of Subtyping in Session Types. *Log. Methods Comput. Sci.* 13, 2 (2017). https://doi.org/10.23638/LMCS-13(2:12)2017

[14] Luca Ciccone, Francesco Dagnino, and Luca Padovani. 2024. Fair termination of multiparty sessions. *J. Log. Algebraic Methods Program.* 139 (2024), 100964. https://doi.org/10.1016/J.JLAMP.2024.100964

[15] Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. 2016. Global progress for dynamically interleaved multiparty sessions. *Math. Struct. Comput. Sci.* 26, 2 (2016), 238–302.

[16] 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.

[17] Pierre-Malo Deniélou and Nobuko Yoshida. 2012. Multiparty Session Types Meet Communicating Automata. In *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7211)*, Helmut Seidl (Ed.). Springer, 194–213. https://doi.org/10.1007/978-3-642-28869-2_10

[18] Pierre-Malo Deniélou and Nobuko Yoshida. 2013. Multiparty Compatibility in Communicating Automata: Characterisation and Synthesis of Global Session Types. In *ICALP (2) (Lecture Notes in Computer Science, Vol. 7966)*. Springer, 174–186.

[19] Pierre-Malo Deniélou and Nobuko Yoshida. 2013. Multiparty Compatibility in Communicating Automata: Characterisation and Synthesis of Global Session Types. In *Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 7966)*, Fedor V. Fomin, Rusins Freivalds, Marta Z. Kwiatkowska, and David Peleg (Eds.). Springer, 174–186. https://doi.org/10.1007/978-3-642-39212-2_18

[20] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. 2002. Flow-Sensitive Type Qualifiers. In *PLDI*. ACM, 1–12.

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

[22] Simon Fowler, Duncan Paul Attard, Franciszek Sowul, Simon J. Gay, and Phil Trinder. 2023. Special Delivery: Programming with Mailbox Types. *Proc. ACM Program. Lang.* 7, ICFP (2023), 78–107.

[23] Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. 2019. Exceptional asynchronous session types: session types without tiers. *Proc. ACM Program. Lang.* 3, POPL (2019), 28:1–28:29.

[24] 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.

[25] Adrian Francalanza and Gerard Tabone. 2023. ElixirST: A session-based type system for Elixir modules. *J. Log. Algebraic Methods Program.* 135 (2023), 100891.

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

[27] Colin S. Gordon. 2017. A Generic Approach to Flow-Sensitive Polymorphic Effects. In *ECOOP (LIPIcs, Vol. 74)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 13:1–13:31.

[28] 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.

[29] Carl Hewitt, Peter Boehler Bishop, and Richard Steiger. 1973. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *IJCAI*. William Kaufmann, 235–245.

[30] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty asynchronous session types. In *POPL*. ACM, 273–284.

[31] Ping Hou, Nicolas Lagaillardie, and Nobuko Yoshida. 2024. Fearless Asynchronous Communications with Timed Multiparty Session Protocols. In *ECOOP (LIPIcs, Vol. 313)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 19:1–19:30.

[32] Raymond Hu, Dimitrios Kouzapas, Olivier Pernet, Nobuko Yoshida, and Kohei Honda. 2010. Type-Safe Eventful Sessions in Java. In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6183)*, Theo D'Hondt (Ed.). Springer, 329–353. https://doi.org/10.1007/978-3-642-14107-2_16

[33] Raymond Hu and Nobuko Yoshida. 2016. Hybrid Session Verification Through Endpoint API Generation. In *FASE (Lecture Notes in Computer Science, Vol. 9633)*. Springer, 401–418.

[34] 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.

[35] Shams Imam. [n. d.]. Savina Actor Benchmark Suite. https://github.com/shamsimam/savina. Accessed: 2024-11-13.

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

[37] Dimitrios Kouzapas, Nobuko Yoshida, Raymond Hu, and Kohei Honda. 2016. On asynchronous eventful session semantics. *Math. Struct. Comput. Sci.* 26, 2 (2016), 303–364.

[38] Nicolas Lagaillardie, Rumyana Neykova, and Nobuko Yoshida. 2022. Stay Safe Under Panic: Affine Rust Programming with Multiparty Session Types. In *ECOOP (LIPIcs, Vol. 222)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 4:1–4:29.

[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] Sam Lindley and James Cheney. 2012. Row-based effect types for database integration. In *TLDI*. ACM, 91–102.

[41] Sam Lindley and J. Garrett Morris. 2015. A Semantics for Propositions as Sessions. In *ESOP (Lecture Notes in Computer Science, Vol. 9032)*. Springer, 560–584.

[42] Anson Miu, Francisco Ferreira, Nobuko Yoshida, and Fangyi Zhou. 2021. Communication-safe web programming in TypeScript with routed multiparty session types. In *CC*. ACM, 94–106.

[43] 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.

[44] Dimitris Mostrous and Vasco T. Vasconcelos. 2018. Affine Sessions. *Log. Methods Comput. Sci.* 14, 4 (2018).

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

[46] Ulf Norell. 2008. Dependently Typed Programming in Agda. In *Advanced Functional Programming (Lecture Notes in Computer Science, Vol. 5832)*. Springer, 230–266.

[47] Luca Padovani. 2017. A simple library implementation of binary sessions. *J. Funct. Program.* 27 (2017), e4. https://doi.org/10.1017/S0956796816000289

[48] Luca Padovani and Gianluigi Zavattaro. 2025. Fair Termination of Asynchronous Binary Sessions. In *39th European Conference on Object-Oriented Programming, ECOOP 2025, June 30 to July 2, 2025, Bergen, Norway (LIPIcs, Vol. 333)*, Jonathan Aldrich and Alexandra Silva (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 24:1–24:29. https://doi.org/10.4230/LIPICS.ECOOP.2025.24

[49] John C. Reynolds. 2000. *The Meaning of Types—From Intrinsic to Extrinsic Semantics.* Technical Report RS-00-32. BRICS.

[50] Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. 2017. A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming. In *ECOOP (LIPIcs, Vol. 74)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 24:1–24:31.

[51] Alceste Scalas and Nobuko Yoshida. 2016. Lightweight Session Programming in Scala. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy (LIPIcs, Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 21:1–21:28. https://doi.org/10.4230/LIPICS.ECOOP.2016.21

[52] Alceste Scalas and Nobuko Yoshida. 2019. Less is more: multiparty session types revisited. *Proc. ACM Program. Lang.* 3, POPL (2019), 30:1–30:29.

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

[54] Peter Thiemann. 2023. Intrinsically Typed Sessions with Callbacks (Functional Pearl). *Proc. ACM Program. Lang.* 7, ICFP (2023), 711–739.

[55] Malte Viering, Raymond Hu, Patrick Eugster, and Lukasz Ziarek. 2021. A multiparty session typing discipline for fault-tolerant event-driven distributed programming. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–30.

[56] Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. 2013. The Scribble Protocol Language. In *TGC (Lecture Notes in Computer Science, Vol. 8358)*. Springer, 22–41.

[57] Fangyi Zhou, Francisco Ferreira, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. 2020. Statically verified refinements for multiparty protocols. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 148:1–148:30.

Appendices

APPENDIX CONTENTS

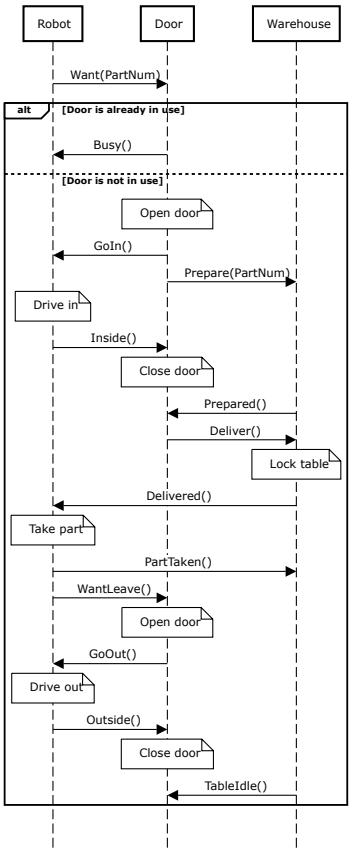
A DETAILS OF CASE STUDY PROTOCOLS

In this section we detail the protocols and sequence diagrams for the two case studies.

A.1 Robots

The robots protocol can be found below, both as a Scribble global type and a sequence diagram. Role `R` stands for Robot, `D` stands for Door, and `W` stands for Warehouse.

```
global protocol Robot(role R, role D, role W) {
  Want(PartNum) from R to D;
  choice at D {
    Busy() from D to R;
    Cancel() from D to W;
  } or {
    GoIn() from D to R;
    Prepare(PartNum) from D to W;
    Inside() from R to D;
    Prepared() from W to D;
    Deliver() from D to W;
    Delivered() from W to R;
    PartTaken() from R to W;
    WantLeave() from R to D;
    GoOut() from D to R;
    Outside() from R to D;
    TableIdle() from W to D;
  }
}
```



Below is the straightforward user code for a Door actor to repeatedly register for an unbounded number of Robot sessions. The Door actor will safely handle all Robot sessions concurrently, coordinated by its encapsulated state (e.g., `isBusy`). The generated `ActorDoor` API provides a `register` method for the formal **register** operation, and `d1Suspend` is a user-defined handler that registers once more after every session initiation.

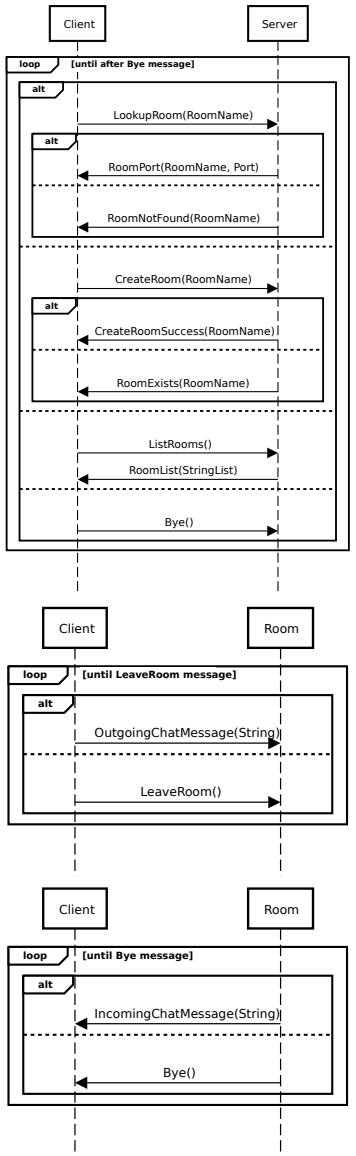
```
1  class Door(pid: Pid, port: Int, apHost: Host, apPort: Int) extends ActorDoor(pid) {
2    private var isBusy = false // Shared state -- n.b. every actor is a single-threaded event loop
3    def spawn(): Unit = { super.spawn(this.port); regForInit(new DataD(...)) }
4    def regForInit(d: DataD) = register(this.port, apHost, apPort, d, d1Suspend)
5    def d1Suspend(d: DataD, s: D1Suspend): Done.type = { regForInit(new DataD(...)); s.suspend(d, d1) }
6    ... // def d1(d: DataD, s: D1): Done.type ... etc.
```

A.2 Chat Server

```
global protocol ChatServer(role C, role S) {
  choice at C {
    LookupRoom(RoomName) from C to S;
    choice at S {
      RoomPort(RoomName, Port) from S to C;
    } or {
      RoomNotFound(RoomName) from S to C;
    }
    do ChatServer(C, S);
  } or {
    CreateRoom(RoomName) from C to S;
    choice at S {
      CreateRoomSuccess(RoomName) from S to C;
    } or {
      RoomExists(RoomName) from S to C;
    }
    do ChatServer(C, S);
  } or {
    ListRooms() from C to S;
    RoomList(StringList) from S to C;
    do ChatServer(C, S);
  } or {
    Bye(String) from C to S;
  }
}

global protocol ChatSessionCtoR(role C, role R) {
  choice at C {
    OutgoingChatMessage(String) from C to R;
    do ChatSessionCtoR(C, R);
  } or {
    LeaveRoom() from C to R;
  }
}

global protocol ChatSessionRtoC(role R, role C){
  choice at R {
    IncomingChatMessage(String) from R to C;
    do ChatSessionRtoC(R, C);
  } or {
    Bye() from R to C;
  }
}
```



B SUPPLEMENT TO SECTION 4

B.1 Omitted Definitions

Term reduction $M \longrightarrow_{\mathsf{M}} N$ is standard β-reduction:

Term reduction rules $\boxed{M \longrightarrow_{\mathsf{M}} N}$

$$\begin{array}{rcll}
\textbf{let } x = \textbf{return } V \textbf{ in } M & \longrightarrow_{\mathsf{M}} & M\{V/x\} & \\
(\lambda x.\, M)\ V & \longrightarrow_{\mathsf{M}} & M\{V/x\} & \\
(\textbf{rec } f(x).M)\ V & \longrightarrow_{\mathsf{M}} & M\{\textbf{rec } f(x).M/f, V/x\} & \\
\textbf{if true then } M \textbf{ else } N & \longrightarrow_{\mathsf{M}} & M & \\
\textbf{if false then } M \textbf{ else } N & \longrightarrow_{\mathsf{M}} & N & \\
\mathcal{E}[M] & \longrightarrow_{\mathsf{M}} & \mathcal{E}[N] & (\text{if } M \longrightarrow_{\mathsf{M}} N)
\end{array}$$

B.2 Preservation

We begin with some unsurprising auxiliary lemmas.

Lemma B.1 (Substitution). *If* $\Gamma, x : B \mid C \mid S \triangleright M : A \triangleleft T$ *and* $\Gamma \vdash V : B$, *then* $\Gamma \mid S \mid C \triangleright M\{V/x\} : A \triangleleft T$.

Proof. By induction on the derivation of $\Gamma, x : A \mid C \mid S \triangleright M : B \triangleleft T$. □

Lemma B.2 (Subterm typability). *Suppose* $\mathbf{D}$ *is a derivation of* $\Gamma \mid C \mid S \triangleright \mathcal{E}[M] : A \triangleleft T$. *Then there exists some subderivation* $\mathbf{D}'$ *of* $\mathbf{D}$ *concluding* $\Gamma \mid C \mid S \triangleright M : B \triangleleft S'$ *for some type* B *and session type* S', *where the position of* $\mathbf{D}'$ *in* $\mathbf{D}$ *corresponds to that of the hole in* $\mathcal{E}$.

Proof. By induction on the structure of $\mathcal{E}$. □

Lemma B.3 (Replacement). *If:*

(1) $\mathbf{D}$ *is a derivation of* $\Gamma \mid C \mid S \triangleright \mathcal{E}[M] : A \triangleleft T$
(2) $\mathbf{D}'$ *is a subderivation of* $\mathbf{D}$ *concluding* $\Gamma \mid C \mid S \triangleright M : B \triangleleft T'$ *where the position of* $\mathbf{D}'$ *in* $\mathbf{D}$ *corresponds to that of the hole in* $\mathcal{E}$
(3) $\Gamma \mid C \mid S' \triangleright N : B \triangleleft T'$

then $\Gamma \mid C \mid S' \triangleright \mathcal{E}[N] : A \triangleleft T$.

Proof. By induction on the structure of $\mathcal{E}$. □

Since type environments are unrestricted, we also obtain a weakening result.

Lemma B.4 (Weakening). *(1) If* $\Gamma \vdash V : B$ *and* $x \notin dom(\Gamma)$, *then* $\Gamma, x : A \vdash V : B$.
(2) If $\Gamma \mid C \mid S \triangleright M : B \triangleleft T$ *and* $x \notin dom(\Gamma)$, *then* $\Gamma, x : A \mid C \mid S \triangleright M : B \triangleleft T$.
(3) If $\Gamma; \Delta \mid C \vdash \sigma$ *and* $x \notin dom(\Gamma)$, *then* $\Gamma, x : A; \Delta \mid C \vdash \sigma$.
(4) If $\Gamma; \Delta \mid C \vdash \rho$ *and* $x \notin dom(\Gamma)$, *then* $\Gamma, x : A; \Delta \mid C \vdash \rho$.
(5) If $\Gamma; \Delta \vdash C$ *and* $x \notin dom(\Gamma)$, *then* $\Gamma, x : A; \Delta \vdash C$.

Proof. By mutual induction on all premises. □

Lemma B.5 (Preservation (terms)). *If* $\Gamma \mid S \mid C \triangleright M : A \triangleleft T$ *and* $M \longrightarrow_{M} N$, *then* $\Gamma \mid S \mid C \triangleright N : A \triangleleft T$.

Proof. A standard induction on the derivation of $M \longrightarrow_{\mathsf{M}} N$, noting that functional reduction does not modify the session type. □

Next, we introduce some MPST-related lemmas that are helpful for proving preservation of configuration reduction. We often make use of these lemmas implicitly.

Lemma B.6. *If safe*(Δ, Δ')*, then safe*(Δ).

Proof sketch. Splitting a context only removes potential reductions. Only by adding reductions could we violate safety. □

Lemma B.7. *If safe*(Δ_1, Δ_2) *and* $\Delta_1 \Longrightarrow \Delta_1'$*, then safe*$(\Delta_1', \Delta_2)$.

Proof. By induction on the derivation of $\Delta_1 \equiv \overset{\pi}{\Longrightarrow} \equiv \Delta_1'$.

It suffices to consider the cases where reduction could potentially make the combined environments unsafe.

In the case of Lbl-Sync-Send, the resulting reduction adds a message $(\mathsf{p}, \mathsf{q}, \ell_i(A_i))$ to a queue Q.

The only way this could violate safety is if there were some entry $s[\mathsf{q}] : \mathsf{p} \,\&\, \{\ell_i(A_i) \,.\, S_i\}_{i \in I}$, and $Q \equiv (\mathsf{p}, \mathsf{q}, \ell_j(A_j)) \cdot Q'$ where $j \in I$, but $(Q \cdot (\mathsf{p}, \mathsf{q}, \ell_k(A_k)) \equiv (\mathsf{p}, \mathsf{q}, \ell_k(A_k)) \cdot Q''$ with $k \notin I$. However, this is impossible since it is not possible to permute this message ahead of the existing message because of the side-conditions on queue equivalence.

A similar argument applies for Lbl-Sync-Recv. □

Lemma B.8. *If* $\Gamma;\Delta, s : Q \vdash s \triangleright \sigma$ *and* $\Gamma \vdash V : A$*, then* $\Gamma;\Delta, s : (Q \cdot (\mathsf{p}, \mathsf{q}, \ell(A))) \vdash s \triangleright \sigma \cdot (\mathsf{p}, \mathsf{q}, \ell(V))$

Proof. A straightforward induction on the derivation of $\Gamma;\Delta, s : Q \vdash s \triangleright \sigma$. □

Lemma B.9. *If safe*(Δ_1) *and safe*(Δ_2) *for environments* Δ_1, Δ_2 *such that snames*$(\Delta_1) \cap$*snames*$(\Delta_2) = \emptyset$ *and where* Δ_1, Δ_2 *is defined, then safe*(Δ_1, Δ_2).

Proof. By inspection of the definition of safe$(-)$ and the environment reduction rules, noting that each are only defined on a single session. □

Lemma B.10. *Given environments* Δ_1, Δ_2 *such that safe*(Δ_1, Δ_2) *and snames*$(\Delta_1) \cap$ *snames*$(\Delta_2) = \emptyset$ *and* $\Delta_1, \Delta_2 \Longrightarrow^? \Delta'$ *such that safe*(Δ')*, either:*

(1) $\Delta' = \Delta$*; or*
(2) $\Delta' = \Delta_1', \Delta_2$ *such that* $\Delta_1 \Longrightarrow \Delta_1'$ *and safe*(Δ_1')*; or*
(3) $\Delta' = \Delta_1, \Delta_2'$ *such that* $\Delta_2 \Longrightarrow \Delta_2'$ *and safe*(Δ_2').

Proof. By inspection of the reduction rules for $\Longrightarrow$, noting that reduction only affects a single session and that the session names in Δ_1, Δ_2 are disjoint. □

Lemma B.11 (Preservation (Equivalence)). *If* $\Gamma;\Delta \vdash C$ *and* $C \equiv \mathcal{D}$ *then there exists some* $\Delta' \equiv \Delta$ *such that* $\Gamma;\Delta' \vdash \mathcal{D}$.

Proof. By induction on the derivation of $C \equiv \mathcal{D}$. The only case that causes the type environment to change is queue message reordering, which can be made typable by mirroring the change in the queue type. □

Lemma B.12 (Preservation (Configuration reduction)). *If* $\Gamma;\Delta \vdash C$ *with safe*(Δ) *and* $C \longrightarrow \mathcal{D}$*, then there exists some* Δ' *such that* $\Delta \Longrightarrow^? \Delta'$ *such that safe*(Δ') *and* $\Gamma;\Delta' \vdash \mathcal{D}$.

Proof. By induction on the derivation of $C \longrightarrow \mathcal{D}$. In each case where $\Delta \Longrightarrow \Delta'$ for some Δ', by the definition of safety it follows that safe(Δ').

Case E-Send.

$$\langle a, (\mathcal{E}[\mathsf{q} \,!\, \ell(V)])^{s[\mathsf{p}]}, \sigma, \rho \rangle \parallel s \triangleright \delta \longrightarrow \langle a, (\mathcal{E}[\mathbf{return}\ ()])^{s[\mathsf{p}]}, \sigma, \rho \rangle \parallel s \triangleright \delta \cdot (\mathsf{p}, \mathsf{q}, \ell(V))$$

Assumption:

$$\dfrac{\dfrac{\dfrac{\Gamma \mid C \mid S \triangleright \mathcal{E}[\mathsf{q}\,!\,\ell(V)] : C \triangleleft \mathsf{end}}{\Gamma; s[\mathsf{p}] : S \mid C \vdash (\mathcal{E}[\mathsf{q}\,!\,\ell(V)])^{s[\mathsf{p}]}} \quad \Gamma;\Delta_2 \mid C \vdash \sigma \quad \Gamma;\Delta_3 \mid C \vdash \rho}{\Gamma;s[\mathsf{p}] : S, \Delta_2, \Delta_3, a \vdash \langle a, (\mathcal{E}[\mathsf{q}\,!\,\ell(V)])^{s[\mathsf{p}]}, \sigma, \rho\rangle} \quad \Gamma;s : Q \vdash s \triangleright \delta}{\Gamma;s[\mathsf{p}] : S, \Delta_2, \Delta_3, s : Q, a \vdash \langle a, (\mathcal{E}[\mathsf{q}\,!\,\ell(V)])^{s[\mathsf{p}]}, \sigma, \rho\rangle \parallel s \triangleright \delta}$$

By Lemma B.2 we have that $\Gamma \mid C \mid \mathsf{q} \oplus \{\ell_i(A_i) : T_i\}_{i\in I} \triangleright \mathsf{q}\,!\,\ell_j(V) : \mathsf{Unit} \triangleleft T_j$ and therefore that $S = \mathsf{q} \oplus \{\ell_i(A_i) : T_i\}_{i\in I}$.

Since $\Gamma \mid C \mid T_j \triangleright \mathbf{return}\ () : \mathsf{Unit} \triangleleft T_j$, we can show by Lemma B.3 we have that $\Gamma \mid C \mid T_j \triangleright \mathcal{E}[\mathbf{return}\ ()] : C \triangleleft \mathsf{end}$.

By Lemma B.8, $\Gamma; s : Q \cdot (\mathsf{p}, \mathsf{q}, \ell_j(A_j)) \vdash s \triangleright \delta \cdot (\mathsf{p}, \mathsf{q}, \ell_j(V))$.

Therefore, recomposing:

$$\dfrac{\dfrac{\dfrac{\Gamma \mid C \mid T_j \triangleright \mathcal{E}[\mathbf{return}\ ()] : C \triangleleft \mathsf{end}}{\Gamma; s[\mathsf{p}] : T_j \mid C \vdash (\mathcal{E}[\mathbf{return}\ ()])^{s[\mathsf{p}]}} \quad \begin{array}{l}\Gamma;\Delta_2 \mid C \vdash \sigma\\ \Gamma;\Delta_3 \mid C \vdash \rho\end{array}}{\Gamma;s[\mathsf{p}] : T_j, \Delta_2, \Delta_3, a \vdash \langle a, (\mathcal{E}[\mathbf{return}\ ()])^{s[\mathsf{p}]}, \sigma, \rho\rangle} \quad \Gamma;s : Q \cdot (\mathsf{p}, \mathsf{q}, \ell_j(A_j)) \vdash s \triangleright \delta \cdot (\mathsf{p}, \mathsf{q}, \ell_j(V))}{\Gamma;s[\mathsf{p}] : T_j, \Delta_2, \Delta_3, s : Q \cdot (\mathsf{p}, \mathsf{q}, \ell_j(B_j)), a \vdash \langle a, (\mathcal{E}[\mathbf{return}\ ()])^{s[\mathsf{p}]}, \sigma, \rho\rangle \parallel s \triangleright \delta \cdot (\mathsf{p}, \mathsf{q}, \ell_j(V))}$$

Finally,

$s[\mathsf{p}] : \mathsf{q} \oplus \{\ell_i(A_i) : T_i\}_{i\in I}, \Delta_2, \Delta_3, s : Q, a \Longrightarrow s[\mathsf{p}] : T_j, \Delta_2, \Delta_3, s : Q \cdot (\mathsf{p}, \mathsf{q}, \ell_j(B_j)), a$ by Lbl-Send as required.

Case E-React.

$$\dfrac{\ell(x) \mapsto M \in \overrightarrow{H}}{\langle a, \mathbf{idle}(U), \sigma[s[\mathsf{p}] \mapsto \mathbf{handler}\ \mathsf{q}\ st\ \{\overrightarrow{H}\}], \rho\rangle \parallel s \triangleright (\mathsf{q}, \mathsf{p}, \ell(V)) \cdot \delta \longrightarrow \langle a, (M\{V/x, U/st\})^{s[\mathsf{p}]}, \sigma, \rho\rangle \parallel s \triangleright \delta}$$

For simplicity (and equivalently) let us refer to ℓ as ℓ_j.

Let **D** be the following derivation:

$$\dfrac{\dfrac{\dfrac{(\Gamma, x_i : B_i, st : C \mid C \mid S_i \triangleright M_i : C \triangleleft \mathsf{end})_{i\in I}}{\Gamma \vdash \mathbf{handler}\ \mathsf{q}\ st\ \{(\ell_i(x_i) \mapsto M_i)_{i\in I}\} : \mathsf{Handler}(S^?, C)} \quad \Gamma;\Delta_2 \mid C \vdash \sigma}{\Gamma;\Delta_2, s[\mathsf{p}] : S^? \mid C \vdash \sigma[s[\mathsf{p}] \mapsto \mathbf{handler}\ \mathsf{q}\ st\ \{\overrightarrow{H}\}]} \quad \dfrac{\dfrac{\Gamma \vdash U : C}{\Gamma; C \mid \cdot \vdash \mathbf{idle}(U)} \quad \Gamma;\Delta_3 \mid C \vdash \rho}{\Gamma;\Delta_2, \Delta_3, s[\mathsf{p}] : S^?, a \vdash \langle a, \mathbf{idle}(U), \sigma[s[\mathsf{p}] \mapsto \mathbf{handler}\ \mathsf{q}\ st\ \{\overrightarrow{H}\}], \rho\rangle}$$

Assumption:

$$\mathbf{D} \quad \dfrac{\Gamma \vdash V : A \qquad \Gamma; s : Q \vdash s \triangleright \delta}{\Gamma; s[\mathsf{p}] : S^?, s : ((\mathsf{q}, \mathsf{p}, \ell_j(A)) \cdot Q) \vdash s \triangleright (\mathsf{q}, \mathsf{p}, \ell_j(V)) \cdot \delta}$$

$$\Gamma;\Delta_2, \Delta_3, s[\mathsf{p}] : S^?, s : ((\mathsf{q}, \mathsf{p}, \ell_j(A)) \cdot Q), a \vdash \langle a, \mathbf{idle}(U), \sigma[s[\mathsf{p}] \mapsto \mathbf{handler}\ \mathsf{q}\ st\ \{\overrightarrow{H}\}], \rho\rangle \parallel s \triangleright (\mathsf{q}, \mathsf{p}, \ell_j(V)) \cdot \delta$$

where $S^? = \mathsf{p}\,\&\{\ell_i(B_i).S_i\}_{i\in I}$.

Since $\mathsf{safe}(\Delta_2, \Delta_3, s[\mathsf{p}] : S^?, s : ((\mathsf{q}, \mathsf{p}, \ell_j(A)) \cdot Q)), a$ we have that $j \in I$ and $A = B_j$.

Similarly since $\ell_j(x_j) \mapsto M \in \overrightarrow{H}$ we have that $\Gamma, x_j : B_j, st : C \mid C \mid S_j \triangleright M : C \triangleleft \mathsf{end}$.

By Lemma B.1, $\Gamma \mid C \mid S_j \triangleright M\{V/x_j, U/st\} : C \triangleleft \mathsf{end}$.

Let **D′** be the following derivation:

$$\dfrac{\dfrac{\Gamma \mid S_j \mid C \triangleright M\{V/x_j, U/st\} : C \triangleleft \mathsf{end}}{\Gamma; s[\mathrm{p}] : S_j \mid C \vdash (M\{V/x_j, U/st\})^{s[\mathrm{p}]}} \qquad \Gamma; \Delta_2 \mid C \vdash \sigma \qquad \Gamma; \Delta_3 \mid C \vdash \rho}{\Gamma; \Delta_2, \Delta_3, s[\mathrm{p}] : S_j, a \vdash \langle a, (M\{V/x_j, U/st\})^{s[\mathrm{p}]}, \sigma, \rho\rangle}$$

Recomposing:

$$\dfrac{\mathbf{D}' \qquad \Gamma; s : Q \vdash s \triangleright \delta}{\Gamma; \Delta_2, \Delta_3, s[\mathrm{p}] : S_j, s : Q, a \vdash \langle a, (M\{V/x_j, U/st\})^{s[\mathrm{p}]}, \sigma, \rho\rangle \parallel s \triangleright \delta}$$

Finally, we note that $\Delta_2, \Delta_3, s[\mathrm{p}] : S^?, s : ((\mathrm{q}, \mathrm{p}, \ell_j(A)) \cdot Q), a \Longrightarrow \Delta_2, \Delta_3, s[\mathrm{p}] : S_j, s : Q, a$ by Lbl-Recv as required.

Case E-Suspend.

$$\langle a, (\mathcal{E}[\mathbf{suspend}\ V\ W])^{s[\mathrm{p}]}, \sigma, \rho\rangle \longrightarrow \langle a, \mathbf{idle}(W), \sigma[s[\mathrm{p}] \mapsto V], \rho\rangle$$

Assumption:

$$\dfrac{\dfrac{\Gamma \mid C \mid S \triangleright \mathcal{E}[\mathbf{suspend}\ V\ W] : C \triangleleft \mathsf{end}}{\Gamma; s[\mathrm{p}] : S \mid C \vdash (\mathcal{E}[\mathbf{suspend}\ V\ W])^{s[\mathrm{p}]}} \qquad \Gamma; \Delta_2 \mid C \vdash \sigma \qquad \Gamma; \Delta_3 \mid C \vdash \rho}{\Gamma; s[\mathrm{p}] : S, \Delta_2, \Delta_3, a \vdash \langle a, (\mathcal{E}[\mathbf{suspend}\ V\ W])^{s[\mathrm{p}]}, \sigma, \rho\rangle}$$

By Lemma B.2 we have that:

$$\dfrac{\Gamma \vdash V : \mathsf{Handler}(S^?, C) \qquad \Gamma \vdash W : C}{\Gamma \mid C \mid S^? \triangleright \mathbf{suspend}\ V\ W : A \triangleleft T}$$

for any arbitrary A, T, and showing that $S = S^?$.

Recomposing:

$$\dfrac{\dfrac{\Gamma \vdash W : C}{\Gamma; \cdot \mid C \vdash \mathbf{idle}(W)} \qquad \dfrac{\Gamma \vdash V : \mathsf{Handler}(S^?, C) \qquad \Gamma; \Delta_2 \mid C \vdash \sigma}{\Gamma; \Delta_2, s[\mathrm{p}] : S^? \mid C \vdash \sigma[s[\mathrm{p}] \mapsto V]} \qquad \Gamma; \Delta_3 \mid C \vdash \rho}{\Gamma; s[\mathrm{p}] : S^?, \Delta_2, \Delta_3, a \vdash \langle a, \mathbf{idle}(W), \sigma[s[\mathrm{p}] \mapsto V], \rho\rangle}$$

as required.

Case E-Spawn.

$$\langle a, \mathcal{M}[\mathbf{spawn}\ M], \sigma, \rho\rangle \longrightarrow (\nu b)(\langle a, \mathcal{M}[\mathbf{return}\ ()], \sigma, \rho\rangle \parallel \langle b, M, \epsilon, \epsilon\rangle)$$

(with b fresh)

There are two subcases based on whether $\mathcal{M} = \mathcal{E}[-]$ or $\mathcal{M} = (\mathcal{E}[-])^{s[\mathrm{p}]}$. Both are similar so we will prove the latter case.

Assumption:

$$\dfrac{\dfrac{\Gamma \mid C \mid S \triangleright \mathcal{E}[\textbf{spawn}\ M] : C \triangleleft \mathsf{end}}{\Gamma; s[\mathsf{p}] : S \mid C \vdash (\mathcal{E}[\textbf{spawn}\ M])^{s[\mathsf{p}]}} \qquad \begin{array}{l}\Gamma;\Delta_2 \mid C \vdash \sigma \\ \Gamma;\Delta_3 \mid C \vdash \rho\end{array}}{\Gamma;\Delta_2, \Delta_3, s[\mathsf{p}] : S, a \vdash \langle a, (\mathcal{E}[\textbf{spawn}\ M])^{s[\mathsf{p}]}, \sigma, \rho\rangle}$$

By Lemma B.2:

$$\dfrac{\Gamma \mid A \mid \mathsf{end} \triangleright M : A \triangleleft \mathsf{end}}{\Gamma \mid C \mid S \triangleright \textbf{spawn}\ M : \mathsf{Unit} \triangleleft S}$$

By Lemma B.3, $\Gamma \mid C \mid S \triangleright \mathcal{E}[\textbf{return}\ ()] : C \triangleleft \mathsf{end}$.
Thus, recomposing:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \mid C \mid S \triangleright \mathcal{E}[\textbf{return}\ ()] : C \triangleleft \mathsf{end}}{\Gamma; s[\mathsf{p}] : S \mid C \vdash (\mathcal{E}[\textbf{return}\ ()])^{s[\mathsf{p}]}} \quad \begin{array}{l}\Gamma;\Delta_2 \mid C \vdash \sigma \\ \Gamma;\Delta_3 \mid C \vdash \rho\end{array}}{\Gamma;\Delta_2, \Delta_3, s[\mathsf{p}] : S, a \vdash \langle a, (\mathcal{E}[\textbf{return}\ ()])^{s[\mathsf{p}]}, \sigma, \rho\rangle} \quad \dfrac{\dfrac{\Gamma \mid A \mid \mathsf{end} \triangleright M : A \triangleleft \mathsf{end}}{\Gamma; \cdot \mid A \vdash M} \quad \begin{array}{l}\Gamma; \cdot \mid A \vdash \epsilon \\ \Gamma; \cdot \mid A \vdash \epsilon\end{array}}{\Gamma; b \vdash \langle b, M, \epsilon, \epsilon\rangle}}{\Gamma;\Delta_2, \Delta_3, s[\mathsf{p}] : S, a, b \vdash \langle a, (\mathcal{E}[\textbf{return}\ ()])^{s[\mathsf{p}]}, \sigma, \rho\rangle \parallel \langle b, M, \epsilon, \epsilon\rangle}}{\Gamma;\Delta_2, \Delta_3, s[\mathsf{p}] : S, a \vdash (\nu b)(\langle a, (\mathcal{E}[\textbf{return}\ ()])^{s[\mathsf{p}]}, \sigma, \rho\rangle \parallel \langle b, M, \epsilon, \epsilon\rangle)}$$

as required.
Case E-Reset.

$$\langle a, \mathcal{Q}[\textbf{return}\ V], \sigma, \rho\rangle \longrightarrow \langle a, \textbf{idle}(V), \sigma, \rho\rangle$$

There are two subcases based on whether $\mathcal{Q} = [-]$ or $\mathcal{Q} = ([-])^{s[\mathsf{p}]}$. We prove the latter case; the former is similar but does not require a context reduction.
Assumption:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash V : C}{\Gamma \mid C \mid \mathsf{end} \triangleright \textbf{return}\ V : C \triangleleft \mathsf{end}}}{\Gamma; s[\mathsf{p}] : \mathsf{end} \mid C \vdash (\textbf{return}\ V)^{s[\mathsf{p}]}} \qquad \begin{array}{l}\Gamma;\Delta_2 \mid C \vdash \sigma \\ \Gamma;\Delta_3 \mid C \vdash \rho\end{array}}{\Gamma;\Delta_2, \Delta_3, s[\mathsf{p}] : \mathsf{end}, a \vdash \langle a, (\textbf{return}\ V)^{s[\mathsf{p}]}, \sigma, \rho\rangle}$$

We can show that $\Delta_2, \Delta_3, s[\mathsf{p}] : \mathsf{end}, a \xRightarrow{\mathsf{end}(s,\mathsf{p})} \Delta_2, \Delta_3, a$, so we can reconstruct:

$$\dfrac{\dfrac{\Gamma \vdash V : C}{\Gamma; \cdot \mid C \vdash \textbf{idle}(V)} \qquad \Gamma;\Delta_2 \mid C \vdash \sigma \qquad \Gamma;\Delta_3 \mid C \vdash \rho}{\Gamma;\Delta_2, \Delta_3, a \vdash \langle a, \textbf{idle}(V), \sigma, \rho\rangle}$$

as required.
Case E-NewAP.

$$\dfrac{p \text{ fresh}}{\langle a, \mathcal{M}[\textbf{newAP}[(\mathsf{p}_i : S_i)_{i\in I}]], \sigma, \rho\rangle \longrightarrow (\nu p)(\langle a, \mathcal{M}[\textbf{return}\ p], \sigma, \rho\rangle \parallel p((\mathsf{p}_i \mapsto \epsilon)_{i\in I}))}$$

As usual we prove the case where $\mathcal{M} = (\mathcal{E}[-])^{s[\mathsf{p}]}$; the case where $\mathcal{M} = (\mathcal{E}[-])$ is similar.

Assumption:

$$\dfrac{\dfrac{\Gamma \mid C \mid T \triangleright \mathcal{E}[\mathbf{newAP}[(\mathrm{p}_i : S_i)_{i\in I}]] : C \triangleleft \mathsf{end}}{\Gamma; s[\mathrm{p}] : T \mid C \vdash (\mathcal{E}[\mathbf{newAP}[(\mathrm{p}_i : S_i)_{i\in I}]])^{s[\mathrm{p}]}} \quad \begin{array}{l}\Gamma;\Delta_2 \mid C \vdash \sigma \\ \Gamma;\Delta_3 \mid C \vdash \rho\end{array}}{\Gamma;\Delta_2,\Delta_3, a \vdash \langle a, (\mathcal{E}[\mathbf{newAP}[(\mathrm{p}_i : S_i)_{i\in I}]])^{s[\mathrm{p}]}, \sigma, \rho\rangle}$$

By Lemma B.2:

$$\dfrac{\mathsf{comp}((\mathrm{p}_i : S_i)_{i\in I})}{\Gamma \mid C \mid T \triangleright \mathbf{newAP}[(\mathrm{p}_i : S_i)_{i\in I}] : \mathsf{AP}((\mathrm{p}_i : S_i)_{i\in I}) \triangleleft T}$$

By Lemma B.3, $\Gamma, p : \mathsf{AP}((\mathrm{p}_i : S_i)_{i\in I}) \mid C \mid T \triangleright \mathcal{E}[\mathbf{return}\ p] : C \triangleleft \mathsf{end}$.
Let $\Gamma' = \Gamma, p : \mathsf{AP}((\mathrm{p}_i : S_i)_{i\in I})$.
By Lemma B.4, since p is fresh we have that $\Gamma'; \Delta_2 \mid C \vdash \sigma$ and $\Gamma'; \Delta_3 \mid C \vdash \rho$.
Recomposing:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma' \mid C \mid T \triangleright \mathcal{E}[\mathbf{return}\ p] : C \triangleleft \mathsf{end}}{\Gamma'; s[\mathrm{p}] : T \mid C \vdash (\mathcal{E}[\mathbf{return}\ p])^{s[\mathrm{p}]}} \quad \begin{array}{l}\Gamma';\Delta_2 \mid C \vdash \sigma \\ \Gamma';\Delta_3 \mid C \vdash \rho\end{array}}{\Gamma';\Delta_2,\Delta_3, s[\mathrm{p}] : T, a \vdash \langle a, (\mathcal{E}[\mathbf{return}\ p])^{s[\mathrm{p}]}, \sigma, \rho\rangle} \quad \dfrac{p : \mathsf{AP}((\mathrm{p}_i : S_i)_{i\in I}) \in \Gamma' \qquad (\cdot \vdash \epsilon : S_i)_{i\in I} \quad \mathsf{comp}((\mathrm{p}_i : S_i)_{i\in I})}{\Gamma'; p \vdash p((\mathrm{p}_i \mapsto \epsilon)_{i\in I})}}{\Gamma';\Delta_2,\Delta_3, s[\mathrm{p}] : T, a, p \vdash \langle a, (\mathcal{E}[\mathbf{return}\ p])^{s[\mathrm{p}]}, \sigma, \rho\rangle \parallel p((\mathrm{p}_i \mapsto \epsilon)_{i\in I})}}{\Gamma;\Delta_2,\Delta_3, s[\mathrm{p}] : T \vdash (\nu p)(\langle a, (\mathcal{E}[\mathbf{return}\ p])^{s[\mathrm{p}]}, \sigma, \rho\rangle \parallel p((\mathrm{p}_i \mapsto \epsilon)_{i\in I}))}$$

as required.
Case E-Register.

$$\dfrac{\iota \text{ fresh}}{\langle a, \mathcal{M}[\mathbf{register}\ p\ \mathrm{p}\ V], \sigma, \rho\rangle \parallel p(\chi[\mathrm{p} \mapsto \widetilde{\iota'}]) \longrightarrow (\nu\iota)(\langle a, \mathcal{M}[\mathbf{return}\ ()], \sigma, \rho[\iota \mapsto V]\rangle \parallel p(\chi[\mathrm{p} \mapsto \widetilde{\iota'} \cup \{\iota\}]))}$$

Again, we prove the case where $\mathcal{M} = (\mathcal{E}[-])^{s[\mathrm{q}]}$ and let $\mathrm{p} = \mathrm{p}_j$ for some j.
Let $\Delta = \Delta_2, \Delta_3, \Delta_4, \widetilde{\iota_j^- : S_j}, s[\mathrm{p}] : T, a, p$.
Let **D** be the following derivation:

$$\dfrac{\dfrac{\Gamma \mid C \mid T \triangleright \mathcal{E}[\mathbf{register}\ p\ \mathrm{p}_j\ V] : C \triangleleft \mathsf{end}}{\Gamma; s[\mathrm{q}] : T \mid C \vdash (\mathcal{E}[\mathbf{register}\ p\ \mathrm{p}_j\ V])^{s[\mathrm{q}]}} \quad \begin{array}{l}\Gamma;\Delta_2 \mid C \vdash \sigma \\ \Gamma;\Delta_3 \mid C \vdash \rho\end{array}}{\Gamma;\Delta_2,\Delta_3, s[\mathrm{q}] : T, a \vdash \langle a, (\mathcal{E}[\mathbf{register}\ p\ \mathrm{p}_j\ V])^{s[\mathrm{q}]}, \sigma, \rho\rangle}$$

Assumption:

$$\dfrac{\dfrac{\dfrac{\{(\mathrm{p}_i : S_i)_{i\in 1..n}\}\ \ \Delta_4 \vdash \chi}{\{(\mathrm{p}_i : S_i)_{i\in 1..n}\}\ \ \Delta_4, \overrightarrow{\iota'^-_j : S_j} \vdash \chi[\mathrm{p}_j \mapsto \widetilde{\iota'}]} \quad \begin{array}{l}p : \mathsf{AP}((\mathrm{p}_i : S_i)_{i\in 1..n}) \in \Gamma \\ \mathsf{comp}((\mathrm{p}_i : S_i)_{i\in 1..n})\end{array}}{\Gamma;\Delta_4, \overrightarrow{\iota'^-_j : S_j}, p \vdash p(\chi[\mathrm{p}_j \mapsto \widetilde{\iota'}])}\ \mathbf{D}}{\Gamma;\Delta \vdash \langle a, (\mathcal{E}[\mathbf{register}\ p\ \mathrm{p}_j\ V])^{s[\mathrm{q}]}, \sigma, \rho\rangle \parallel p(\chi[\mathrm{p}_j \mapsto \widetilde{\iota'}])}$$

By Lemma B.2:

$$\dfrac{\Gamma \vdash p : \mathsf{AP}((\mathrm{p}_i : S_i)_i) \qquad \Gamma \vdash V : C \xrightarrow{S_j, \mathsf{end}}_{C} C}{\Gamma \mid C \mid T \triangleright \mathbf{register}\ p\ \mathrm{p}_j\ V : \mathsf{Unit} \triangleleft T}$$

By Lemma B.3, $\Gamma \mid C \mid T \triangleright \mathcal{E}[\mathbf{return}\ ()] : C \triangleleft \mathsf{end}$.
Now, let $\mathbf{D}'$ be the following derivation:

$$\dfrac{\dfrac{\Gamma \mid C \mid T \triangleright \mathcal{E}[\mathbf{return}\ ()] : C \triangleleft \mathsf{end}}{\Gamma; s[\mathsf{q}] : T \mid C \vdash (\mathcal{E}[\mathbf{return}\ ()])^{s[\mathsf{q}]}} \qquad \dfrac{\Gamma \vdash V : C \xrightarrow[C]{S_j,\mathsf{end}} C \qquad \Gamma; \Delta_3 \mid C \vdash \rho}{\Gamma; \Delta_3, \iota^+ : S_j \mid C \vdash \rho[\iota^+ \mapsto V]} \qquad \Gamma; \Delta_2 \mid C \vdash \sigma}{\Gamma; \Delta_2, \Delta_3, s[\mathsf{q}] : S, \iota^+ : S_j, a \vdash \langle a, (\mathcal{E}[\mathbf{return}\ ()])^{s[\mathsf{q}]}, \sigma, \rho\rangle}$$

Finally, we can recompose:

$$\dfrac{\mathbf{D}\quad \dfrac{\dfrac{\dfrac{\{(\mathsf{p}_i : S_i)_{i\in 1..n}\}\ \Delta_4 \vdash \chi}{\{(\mathsf{p}_i : S_i)_{i\in 1..n}\}\ \Delta_4, \overrightarrow{\iota'^-_j : S_j}, \iota^- : S_j \vdash \chi[\mathsf{p}_j \mapsto \widetilde{\iota'} \cup \{\iota\}]} \quad \begin{array}{l} p : \mathsf{AP}((\mathsf{p}_i : S_i)_{i\in 1..n}) \in \Gamma \\ \mathsf{comp}((\mathsf{p}_i : S_i)_{i\in 1..n}) \end{array}}{\Gamma; \Delta_4, \overrightarrow{\iota'^-_j : S_j}, \iota^- : S_j, p \vdash p(\chi[\mathsf{p}_j \mapsto \widetilde{\iota'} \cup \{\iota\}])}}{\Gamma; \Delta, \iota^+ : S_j, \iota^- : S_j \vdash \langle a, (\mathcal{E}[\mathbf{return}\ ()])^{s[\mathsf{q}]}, \sigma, \rho\rangle \parallel p(\chi[\mathsf{p}_j \mapsto \widetilde{\iota'} \cup \{\iota\}])}}{\Gamma; \Delta \vdash (\nu\iota)(\langle a, (\mathcal{E}[\mathbf{return}\ ()])^{s[\mathsf{q}]}, \sigma, \rho\rangle \parallel p(\chi[\mathsf{p}_j \mapsto \widetilde{\iota'} \cup \{\iota\}]))}$$

as required.
Case E-Init.

$$\dfrac{s \text{ fresh}}{\begin{array}{c}(\nu\iota_{\mathsf{p}_i})_{i\in 1..n}(p((\mathsf{p}_i \mapsto \widetilde{\iota'_{\mathsf{p}_i}} \cup \{\iota_{\mathsf{p}_i}\})_{i\in 1..n}) \parallel \langle a_i, \mathbf{idle}(U_i), \sigma_i, \rho_i[\iota_{\mathsf{p}_i} \mapsto (\lambda st_i.\, M_i)]\rangle_{i\in 1..n}) \xrightarrow{\tau} \\ (\nu s)(p((\mathsf{p}_i \mapsto \widetilde{\iota'_{\mathsf{p}_i}})_{i\in 1..n}) \parallel s \triangleright \epsilon \parallel \langle a_i, (M_i\{U_i/st_i\})^{s[\mathsf{p}_i]}, \sigma_i, \rho_i\rangle_{i\in 1..n})\end{array}}$$

For each actor composed in parallel we have:

$$\dfrac{\dfrac{\Gamma \vdash \lambda st_i.\, M_i : C_i \xrightarrow[C_i]{S_i,\mathsf{end}} C_i \qquad \Gamma; \Delta_{i_3} \mid C_i \vdash \rho}{\Gamma; C_i \mid \Delta_{i_3}, \iota_i^+ : S_i \vdash \rho_i[\iota_{\mathsf{p}_i} \mapsto \lambda st_i.\, M_i]} \qquad \dfrac{\dfrac{\Gamma \vdash U_i : C_i}{\Gamma; \cdot \mid C_i \vdash \mathbf{idle}(U_i)} \qquad \Gamma; \Delta_{i_2} \mid C_i \vdash \sigma_i}{\Gamma; \Delta_{i_2}, \Delta_{i_3}, \iota_i^+ : S_i, a_i \vdash \langle a_i, \mathbf{idle}(U_i), \sigma_i, \rho_i[\iota_{\mathsf{p}_i} \mapsto (\lambda st_i.\, M_i)]\rangle}$$

Let:

- $\Delta_{tok+} = \iota_1^+ : S_1, \ldots, \iota_n^+ : S_n$
- $\Delta_{tok-} = \iota_1^- : S_1, \ldots, \iota_n^- : S_n$
- $\Delta_a = \Delta_{1_2}, \Delta_{1_3}, \ldots, \Delta_{n_2}, \Delta_{n_3}, a_1, \ldots, a_n$
- $\Delta_b = \Delta_a, \Delta_{tok+}$

Then by repeated use of TC-Par we have that $\Gamma; \Delta_a, \Delta_{tok+} \vdash (\langle a, \mathbf{idle}(U_i), \sigma_i, \rho_i[\iota_{\mathsf{p}_i} \mapsto \lambda st_i.\, M_i]\rangle)_{i\in 1..n}$
Assumption (given some Δ):

$$\dfrac{\dfrac{\dfrac{\begin{array}{c} p : \mathsf{AP}((\mathsf{p}_i : S_i)_i) \in \Gamma \\ \{\mathsf{p}_i : S_i\}\ \Delta, \Delta_{tok-} \vdash (\mathsf{p}_i \mapsto \widetilde{\iota'_{\mathsf{p}_i}} \cup \{\iota_{\mathsf{p}_i}\})_{i\in 1..n} \\ \mathsf{comp}((\mathsf{p}_i : S_i)_{i\in 1..n}) \end{array}}{\Gamma; \Delta, \Delta_{tok-} \vdash p((\mathsf{p}_i \mapsto \widetilde{\iota'_{\mathsf{p}_i}} \cup \{\iota_{\mathsf{p}_i}\})_{i\in 1..n})} \qquad \Gamma; \Delta_a, \Delta_{tok+} \vdash (\langle a, \mathbf{idle}(U_i), \sigma_i, \rho_i[\iota_{\mathsf{p}_i} \mapsto \lambda st_i.\, M_i]\rangle)_{i\in 1..n}}{\Gamma; \Delta, \Delta_a, \Delta_{tok+}, \Delta_{tok-} \vdash p((\mathsf{p}_i \mapsto \widetilde{\iota'_{\mathsf{p}_i}} \cup \{\iota_{\mathsf{p}_i}\})_{i\in 1..n}) \parallel (\langle a, \mathbf{idle}(U_i), \sigma_i, \rho_i[\iota_{\mathsf{p}_i} \mapsto \lambda st_i.\, M_i]\rangle)_{i\in 1..n}}}{\Gamma; \Delta, \Delta_a \vdash (\nu\iota_1) \cdots (\nu\iota_n)(p((\mathsf{p}_i \mapsto \widetilde{\iota'_{\mathsf{p}_i}} \cup \{\iota_{\mathsf{p}_i}\})_{i\in 1..n}) \parallel (\langle a, \mathbf{idle}(U_i), \sigma_i, \rho_i[\iota_{\mathsf{p}_i} \mapsto \lambda st_i.\, M_i]\rangle)_{i\in 1..n})}$$

By Lemma B.1 we can show that for each callback function $\lambda st_i.\, M_i$, it is the case that $\Gamma \mid C_i \mid S_i \triangleright M_i\{U_i/st_i\} : C_i \triangleleft \mathsf{end}$.

Through the access point typing rules we can show that we can remove each $\iota_{\mathtt{p}_i}$ from the access point: $\Gamma;\Delta \vdash p((\mathtt{p}_i \mapsto \widetilde{\iota'_{\mathtt{p}_i}})_{i\in 1..n})$.

Similarly, for each actor composed in parallel we can construct:

$$\frac{\Gamma \mid C_i \mid S_i \triangleright M_i\{U_i/st_i\} : C_i \triangleleft \mathsf{end}}{\Gamma; s[\mathtt{p}_i] : S_i \mid C_i \vdash (M_i\{U_i/st_i\})^{s[\mathtt{p}_i]} \qquad \Gamma;\Delta_{i_2} \mid C_i \vdash \sigma_i \qquad \Gamma;\Delta_{i_3} \mid C_i \vdash \rho_i}{\Gamma;\Delta_{i_2},\Delta_{i_3}, s[\mathtt{p}_i] : S_i \vdash \langle a, (M_i\{U_i/st_i\})^{s[\mathtt{p}_i]}, \sigma_i, \rho_i\rangle}$$

Let $\Delta_s = s[\mathtt{p}_1] : S_1, \ldots, s[\mathtt{p}_n] : S_n$

Then by repeated use of TC-Par we have that $\Gamma;\Delta_a, \Delta_s \vdash \langle a, (M_i\{U_i/st_i\})^{s[\mathtt{p}_i]}, \sigma_i, \rho_i\rangle_{i\in 1..n}$.

Recomposing:

$$\frac{\dfrac{\mathsf{comp}(\Delta_s) \quad \Gamma;\Delta \vdash p((\mathtt{p}_i \mapsto \widetilde{\iota'_{\mathtt{p}_i}})_{i\in 1..n})}{\Gamma;\Delta, \Delta_a, \Delta_s, s : \epsilon \vdash p((\mathtt{p}_i \mapsto \widetilde{\iota'_{\mathtt{p}_i}})_{i\in 1..n}) \parallel s \triangleright \epsilon \parallel (\langle a, (M_i\{U_i/st_i\})^{s[\mathtt{p}_i]}, \sigma_i, \rho_i\rangle)_{i\in 1..n}}}{\Gamma;\Delta, \Delta_a \vdash (\nu s)(p((\mathtt{p}_i \mapsto \widetilde{\iota'_{\mathtt{p}_i}})_{i\in 1..n}) \parallel s \triangleright \epsilon \parallel (\langle a, (M_i\{U_i/st_i\})^{s[\mathtt{p}_i]}, \sigma_i, \rho_i\rangle)_{i\in 1..n})}$$

where the premises are $\dfrac{\overline{\Gamma; s : \epsilon \vdash s \triangleright \epsilon} \qquad \Gamma;\Delta_a, \Delta_s \vdash (\langle a, (M_i\{U_i/st_i\})^{s[\mathtt{p}_i]}, \sigma_i, \rho_i\rangle)_{i\in 1..n}}{\Gamma;\Delta_a, \Delta_s, s : \epsilon \vdash s \triangleright \epsilon \parallel (\langle a, (M_i\{U_i/st_i\})^{s[\mathtt{p}_i]}, \sigma_i, \rho_i\rangle)_{i\in 1..n}}$

as required.

Case E-Lift.

Immediate by Lemma B.5.

Case E-Nu.

There are different subcases based on whether α is an access point name, initialisation token name, actor name, or session name. All except session names follow from a straightforward application of the induction hypothesis so we prove the case where $\alpha = s$ for some session name s.

Assumption:

$$\frac{\Delta_s = \{s[\mathtt{p}_i] : S_{\mathtt{p}_i}\}_{i\in 1..n}, s : Q \qquad \mathsf{comp}(\Delta_s) \qquad s \notin \mathsf{snames}(\Delta) \qquad \Gamma;\Delta, \Delta_s \vdash C}{\Gamma;\Delta \vdash (\nu s)C}$$

with $C \longrightarrow C'$.

Since $\mathsf{comp}(\Delta_s)$ we have that $\mathsf{safe}(\Delta_s)$ and $\mathsf{df}(\Delta_s)$.

Since $s \notin \Delta$ and therefore $\mathsf{snames}(\Delta) \cap \mathsf{snames}(\Delta_s) = \emptyset$, by Lemma B.9 we have that $\mathsf{safe}(\Delta, \Delta_s)$.

By the IH we have that there exists some Δ' such that $\Delta, \Delta_s \Longrightarrow^? \Delta'$, where $\mathsf{safe}(\Delta')$ and $\Gamma;\Delta' \vdash C'$.

By Lemma B.10, there are three subcases:

- $\Delta' = \Delta$, which follows trivially.
- $\Delta' = \Delta'', \Delta_s$ where $\Delta \Longrightarrow \Delta''$ with $\mathsf{safe}(\Delta'')$ and we can therefore show:

$$\frac{\Delta_s = \{s[\mathtt{p}_i] : S_{\mathtt{p}_i}\}_{i\in 1..n}, s : Q \qquad \mathsf{comp}(\Delta_s) \qquad s \notin \mathsf{snames}(\Delta'') \qquad \Gamma;\Delta'', \Delta_s \vdash C'}{\Gamma;\Delta'' \vdash (\nu s)C'}$$

  as required.
- $\Delta' = \Delta, \Delta'_s$ where $\Delta_s \Longrightarrow \Delta'_s$ and $\mathsf{safe}(\Delta'_s)$. It follows from the definition of progress that $\mathsf{df}(\Delta'_s)$ and thus $\mathsf{comp}(\Delta'_s)$. We can therefore show:

$$\frac{\Delta'_s = \{s[\mathsf{p}_i] : S'_{\mathsf{p}_i}\}_{i\in 1..m}, s : Q' \qquad \mathsf{comp}(\Delta'_s) \qquad s \notin \mathsf{snames}(\Delta) \qquad \Gamma;\Delta, \Delta'_s \vdash C'}{\Gamma;\Delta \vdash (\nu s)C'}$$

as required.

Case E-Par.

Immediate by the IH and Lemma B.7.

Case E-Struct.

Immediate by the IH and Lemma B.11.

□

Theorem 4.2 (Preservation). *Typability is preserved by structural congruence and reduction.*

(≡) If $\Gamma;\Delta \vdash C$ *and* $C \equiv \mathcal{D}$ *then there exists some* $\Delta' \equiv \Delta$ *such that* $\Gamma;\Delta' \vdash \mathcal{D}$.

(→) If $\Gamma;\Delta \vdash C$ *with* $\mathit{safe}(\Delta)$ *and* $C \rightarrow \mathcal{D}$, *then there exists some* Δ' *such that* $\Delta \Longrightarrow^{?} \Delta'$ *where* $\mathit{safe}(\Delta')$ *and* $\Gamma;\Delta' \vdash \mathcal{D}$.

Proof. Immediate from Lemmas B.11 and B.12. □

B.3 Progress

Let Ψ be a type environment containing only references to access points:

$$\Psi ::= \cdot \mid \Psi, p : \mathsf{AP}((\mathrm{p}_i : S_i)_i)$$

Functional reduction satisfies progress.

Lemma B.13 (Term Progress). *If $\Psi \mid S_1 \triangleright M : A \triangleleft S_2$ then either:*

- *$M = \textbf{\textit{return}}\ V$ for some value V; or*
- *there exists some N such that $M \longrightarrow_M N$; or*
- *M can be written $\mathcal{E}[M']$ where M' is a communication or concurrency construct, i.e.*
  - *$M = \textbf{\textit{spawn}}\ N$ for some N; or*
  - *$M = \mathrm{p} \,!\, \ell(V)$ for some role p and message $\ell(V)$; or*
  - *$M = \textbf{\textit{suspend}}\ V\ W$ or some V, W; or*
  - *$M = \textbf{\textit{newAP}}[(\mathrm{p}_i : T_i)]$ for some collection of participants $(\mathrm{p}_i : T_i)$*
  - *$M = \textbf{\textit{register}}\ V\ \mathrm{p}\ W$ for some values V, W and role p*

Proof. A standard induction on the derivation of $\Psi \mid S_1 \triangleright M : A \triangleleft S_2$; there are β-reduction rules for all STLC terms, leaving only values and communication / concurrency terms. □

The key *thread progress* lemma shows that each actor is either idle, or can reduce; the proof is by inspection of $\mathcal{T}$, noting there are reduction rules for each construct; the runtime typing rules ensure the presence of any necessary queues or access points.

Lemma B.14 (Thread Progress). *Let $C = \mathcal{G}[\langle a, \mathcal{T}, \sigma, \rho\rangle]$. If $\cdot;\cdot \vdash C$ then either $\mathcal{T} = \textbf{\textit{idle}}(V)$ for some value V, or there exist $\mathcal{G}', \mathcal{T}', \sigma', \rho', V'$ such that $C \longrightarrow \mathcal{G}'[\langle a, \mathcal{T}', \sigma', \rho'\rangle]$ is a thread reduction for a.*

Proof. If $\mathcal{T} = \textbf{idle}(V)$ then the theorem is satisfied, so consider the cases where $\mathcal{T} = M$ or $\mathcal{T} = (M)^{s[\mathrm{p}]}$. By Lemma B.13, either M can reduce (and the configuration can reduce via E-Lift), M is a value (and the thread can reduce by E-Reset), or M is a state, communication or concurrency construct. Of these:

- **get** and **set** can reduce by E-Get and E-Set respectively
- **spawn** N can reduce by E-Spawn
- **suspend** V can reduce by E-Suspend
- **newAP**$[(\mathrm{p}_i : S_i)_i]$ can reduce by E-NewAP

Next, consider **register** $p\ \mathrm{p}\ M$. Since we begin with a closed environment, it must be the case that p is ν-bound so by T-APName and T-AP there must exist some subconfiguration $p(\chi)$ of $\mathcal{G}$; the configuration can therefore reduce by E-Register.

Finally, consider $M = \mathrm{q} \,!\, \ell(V)$. It cannot be the case that $\mathcal{T} = \mathrm{q} \,!\, \ell(V)$ since by T-Send the term must have an output session type as a precondition, whereas TT-NoSess assigns a precondition of end. Therefore, it must be the case that $\mathcal{T} = (\mathrm{q} \,!\, \ell(V))^{s[\mathrm{p}]}$ for some s, p. Again since the initial runtime typing environment is empty, it must be the case that s is ν-bound and so by T-SessionName and T-EmptyQueue/T-ConsQueue there must be some session queue $s \triangleright \delta$. The thread must therefore be able to reduce by E-Send. □

Proposition B.15. *If $\Gamma;\Delta \vdash C$ then there exists a $\mathcal{D} \equiv C$ where $\mathcal{D}$ is in canonical form.*

Theorem 4.5 (Progress). *If $\cdot;\cdot \vdash C$, then either there exists some $\mathcal{D}$ such that $C \longrightarrow \mathcal{D}$, or C is structurally congruent to the following canonical form:*

$$(\nu \tilde{\imath})(\nu p_{i \in 1..m})(\nu a_{j \in 1..n})(p_1(\chi_1)_{i \in 1..m} \parallel \langle a_j, \textbf{\textit{idle}}(V_j), \epsilon, \rho_j\rangle_{j \in 1..n})$$

Proof. By Proposition B.15 C can be written in canonical form:

$$(\nu\tilde{\imath})(\nu p_{i\in 1..l})(\nu s_{j\in 1..m})(\nu a_{k\in 1..n})(p_i(\chi_i)_{i\in 1..l} \parallel (s_j \triangleright \delta_j)_{j\in 1..m} \parallel \langle a_k, \mathcal{T}_k, \sigma_k, \rho_k\rangle_{k\in 1..n})$$

By repeated applications of Lemma B.14, either the configuration can reduce or all threads are idle:

$$(\nu\tilde{\imath})(\nu p_{i\in 1..l})(\nu s_{j\in 1..m})(\nu a_{k\in 1..n})(p_i(\chi_i)_{i\in 1..l} \parallel (s_j \triangleright \delta_j)_{j\in 1..m} \parallel \langle a_k, \mathbf{idle}(U_k), \sigma_k, \rho_k\rangle_{k\in 1..n})$$

By the linearity of runtime type environments Δ, each role endpoint $s[\mathrm{p}]$ must be contained in precisely one actor. There are two ways an endpoint can be used: either by TT-Sess in order to run a term in the context of a session, or by TH-Handler to record a receive session type as a handler. Since all threads are idle, it must be the case the only applicable rule is TH-Handler and therefore each role must have an associated stored handler.

Since the types for each session must satisfy progress, the collection of local types must reduce. Since all session endpoints must have a receive session type, the only type reductions possible are through Lbl-Sync-Recv. Since all threads are idle we can pick the top message from any session queue and reduce the actor with the associated stored handler by E-React.

The only way we could not do such a reduction is if there were to be no sessions, leaving us with a configuration of the form:

$$(\nu\tilde{\imath})(\nu p_{i\in 1..m})(\nu a_{j\in 1..n})(p_i(\chi_i)_{i\in 1..m} \parallel \langle a_j, \mathbf{idle}(U_j), \sigma_j, \rho_j\rangle_{j\in 1..n})$$

□

C SUPPLEMENT TO SECTION 5

This appendix details the full formal development and proofs for $\text{Maty}_{\lightning}$ (Section 5).

First, it is useful to show that safety is preserved even if several roles are cancelled; we use this lemma implicitly throughout the preservation proof.

Let us write $\text{roles}(\Delta) = \{\mathtt{p} \mid s[\mathtt{p}] : S \in \Phi\}$ to retrieve the roles from an environment. Let us also define the operation $\text{zap}(\Phi, \widetilde{\mathtt{p}})$ that cancels any role in the given set, i.e., $\text{zap}(s[\mathtt{p}_1] : S_1, s[\mathtt{p}_2] : S_2, a, \{\mathtt{p}_1\}) = s[\mathtt{p}_1] : \lightning, s[\mathtt{p}_2] : S_2, a$.

Lemma C.1. *If safe(Φ) then safe($zap(\Phi, \widetilde{\mathtt{p}})$) for any $\widetilde{\mathtt{p}} \subseteq roles(\Phi)$.*

Proof. Zapping a role does not affect safety; the only way to violate safety is by *adding* further unsafe communication reductions. □

Theorem 5.1 (Preservation ($\longrightarrow$, $\text{Maty}_{\lightning}$)). *If $\Gamma;\Phi \vdash C$ with safe(Φ) and $C \longrightarrow \mathcal{D}$, then there exists some Φ' such that $\Phi \Rrightarrow \Phi'$ and safe(Φ') and $\Gamma;\Phi' \vdash \mathcal{D}$.*

Proof. Preservation of typability by structural congruence is straightforward, so we concentrate on preservation of typability by reduction. We proceed by induction on the derivation of $C \longrightarrow \mathcal{D}$, concentrating on the new rules rather than the adapted rules (which are straightforward changes to the existing proof).

Case E-Monitor.

$$\langle a, \mathcal{M}[\mathbf{monitor}\ b\ V], \sigma, \rho, \omega\rangle \xrightarrow{\tau} \langle a, \mathcal{M}[\mathbf{return}\ ()], \sigma, \rho, \omega \cup \{(b, V)\}\rangle$$

We consider the case where $\mathcal{M} = \mathcal{E}[-]$ for some $\mathcal{E}$; the case in the context of a session is similar.

Assumption:

$$\frac{\dfrac{\Gamma \mid S \mid C \triangleright \mathcal{E}[\mathbf{monitor}\ b\ V] : C \triangleleft \mathsf{end}}{\Gamma;\cdot \mid C \vdash \mathcal{E}[\mathbf{monitor}\ b\ V]} \qquad \Gamma;\Phi_1 \mid C \vdash \sigma \qquad \Gamma;\Phi_2 \mid C \vdash \rho}{\Gamma;\Phi_1, \Delta_2, a \vdash \langle a, \mathcal{E}[\mathbf{monitor}\ b\ V], \sigma, \rho, \omega\rangle}$$

where $\forall (b, W) \in \omega.\ \Gamma \vdash b : \mathsf{Pid} \wedge \Gamma \vdash W : C \xrightarrow[C]{\mathsf{end},\mathsf{end}} C$.

By Lemma B.2, we know:

$$\frac{\Gamma \vdash b : \mathsf{Pid} \qquad \Gamma \vdash V : C \xrightarrow[C]{\mathsf{end},\mathsf{end}} C}{\Gamma \mid C \mid S \triangleright \mathbf{monitor}\ b\ V : \mathsf{Unit} \triangleleft S}$$

By Lemma B.3 we know $\Gamma \mid C \mid S \triangleright \mathcal{E}[\mathbf{return}\ ()] : C \triangleleft \mathsf{end}$.

Recomposing:

$$\frac{\dfrac{\Gamma \mid C \mid S \triangleright \mathcal{E}[\mathbf{return}\ ()] : C \triangleleft \mathsf{end}}{\Gamma;\cdot \mid C \vdash \mathcal{E}[\mathbf{return}\ ()]} \qquad \Gamma;\Phi_1 \mid C \vdash \sigma \qquad \Gamma;\Phi_2 \mid C \vdash \rho}{\Gamma;\Phi_1, \Delta_2, a \vdash \langle a, \mathcal{E}[\mathbf{return}\ ()], \sigma, \rho, \omega \cup (b, V)\rangle}$$

noting that $\omega \cup (b, V)$ is well-typed since $\Gamma \vdash b : \mathsf{Pid}$ and $\Gamma \vdash V : C \xrightarrow[C]{\mathsf{end},\mathsf{end}} C$, as required.

Case E-InvokeM.

$$\langle a, \mathbf{idle}(U), \sigma, \rho, \omega \cup \{(b, V)\}\rangle \parallel \lightning\, b \xrightarrow{\tau} \langle a, V\ U, \sigma, \rho, \omega\rangle \parallel \lightning\, b$$

Assumption:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash U : C}{\Gamma;\cdot \mid C \vdash \mathbf{idle}(U)} \qquad \Gamma;\Phi_1 \mid C \vdash \sigma \qquad \Gamma;\Phi_2 \mid C \vdash \rho}{\Gamma;\Phi_1,\Phi_2,a \vdash \langle a, \mathbf{idle}(U), \sigma, \rho, \omega \cup \{(b,V)\}\rangle} \qquad \dfrac{}{\Gamma;b \vdash \lightning b}}{\Gamma;\Phi_1,\Phi_2,a,b \vdash \langle a, \mathbf{idle}(U), \sigma, \rho, \omega \cup \{(b,V)\}\rangle \parallel \lightning b}$$

where $\forall (a', W) \in \omega \cup \{(b,V)\}.\ \Gamma \vdash b : \mathsf{Pid} \wedge \Gamma \vdash W : C \xrightarrow[C]{\mathsf{end},\mathsf{end}} C$.

Recomposing:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash V : C \xrightarrow[C]{\mathsf{end},\mathsf{end}} C \qquad \Gamma \vdash U : C}{\Gamma \mid C \mid \mathsf{end} \triangleright V\ U : C \triangleleft \mathsf{end}}}{\Gamma;\cdot \mid C \vdash V\ U} \qquad \Gamma;\Phi_1 \mid U \vdash \sigma \qquad \Gamma;\Phi_2 \mid U \vdash \rho}{\Gamma;\Phi_1,\Phi_2,a \vdash \langle a, V\ U, \sigma, \rho, \omega\rangle} \qquad \dfrac{}{\Gamma;b \vdash \lightning b}}{\Gamma;\Phi_1,\Phi_2,a,b \vdash \langle a, V\ U, \sigma, \rho, \omega\rangle \parallel \lightning b}$$

as required.

Case E-Raise.

Similar to E-RaiseS.

Case E-RaiseS.

$$\langle a, (\mathcal{E}[\mathbf{raise}])^{s[\mathrm{p}]}, \sigma, \rho, \omega\rangle \xrightarrow{\tau} \lightning a \parallel \lightning s[\mathrm{p}] \parallel \lightning \sigma \parallel \lightning \rho$$

$$\dfrac{\dfrac{\Gamma \mid C \mid S \triangleright \mathcal{E}[\mathbf{raise}] : \mathsf{Unit} \triangleleft \mathsf{end}}{\Gamma;s[\mathrm{p}] : S \mid C \vdash (\mathcal{E}[\mathbf{raise}])^{s[\mathrm{p}]}} \qquad \Gamma;\Phi_1 \mid C \vdash \sigma \qquad \Gamma;\Phi_2 \mid C \vdash \rho \qquad \Gamma \vdash U : C}{\Gamma;\Phi_1,\Phi_2,s[\mathrm{p}] : S, a \vdash \langle a, (\mathcal{E}[\mathbf{raise}])^{s[\mathrm{p}]}, \sigma, \rho, \omega\rangle}$$

Let us write $\lightning\Phi = \{s[\mathrm{p}] : \lightning \mid s[\mathrm{p}] : S \in \Phi\}$. It follows that for a given environment, $\Phi \rightsquigarrow^{*}_{\triangleright} \lightning\Phi$.

The result follows by noting that due to TH-Handler and TI-Callback we have that $\mathsf{fn}(\Phi_1) = \mathsf{fn}(\sigma)$ and $\mathsf{fn}(\Phi_2) = \mathsf{fn}(\rho)$. Thus:

- $\Gamma;\lightning\Phi_1 \vdash \lightning\sigma$,
- $\Gamma;\lightning\Phi_2 \vdash \lightning\rho$,
- $\Gamma;\lightning\Phi_1,\lightning\Phi_2,s[\mathrm{p}] : \lightning, a \vdash \lightning a \parallel \lightning s[\mathrm{p}] \parallel \lightning\sigma \parallel \lightning\rho$

with the environment reduction:

$$\Phi_1,\Phi_2,s[\mathrm{p}] : S, a \rightsquigarrow^{+}_{\triangleright} \lightning\Phi_1,\lightning\Phi_2,s[\mathrm{p}] : \lightning, a$$

as required.

Case E-CancelMsg.

$$s \triangleright (\mathrm{p},\mathrm{q},\ell(V)) \cdot \delta \parallel \lightning s[\mathrm{q}] \xrightarrow{\tau} s \triangleright \delta \parallel \lightning s[\mathrm{q}]$$

Assumption:

$$\dfrac{\dfrac{\Gamma \vdash V : A \qquad \Gamma;s : Q \vdash s \triangleright \delta}{\Gamma;s : (\mathrm{p},\mathrm{q},\ell(A)) \cdot Q \vdash s \triangleright (\mathrm{p},\mathrm{q},\ell(V)) \cdot \delta} \qquad \Gamma;s[\mathrm{q}] : \lightning \vdash \lightning s[\mathrm{q}]}{\Gamma;s[\mathrm{q}] : \lightning, s : (\mathrm{p},\mathrm{q},\ell(V)) \cdot Q \vdash s \triangleright (\mathrm{p},\mathrm{q},\ell(V)) \cdot \delta \parallel \lightning s[\mathrm{q}]}$$

Recomposing, we have:

$$\frac{\Gamma; s : Q \vdash s \triangleright \delta \qquad \Gamma; s[\mathsf{q}] : \lightning \vdash \lightning s[\mathsf{q}]}{\Gamma; s[\mathsf{q}] : \lightning, s : Q \vdash s \triangleright \delta \parallel \lightning s[\mathsf{q}]}$$

with

$$s[\mathsf{q}] : \lightning, s : (\mathsf{p}, \mathsf{q}, \ell(V)) \cdot Q \xrightarrow{s:\mathsf{p}\lightning\mathsf{q}::\ell} s[\mathsf{q}] : \lightning, s : Q$$

as required.

Case E-CancelAP.

$$(\nu\iota)(p(\chi[\mathsf{p} \mapsto \widetilde{\iota'} \cup \{\iota\}]) \parallel \lightning \iota) \xrightarrow{\tau} p(\chi[\mathsf{p} \mapsto \widetilde{\iota'}])$$

Assumption:

$$\dfrac{\dfrac{p : \mathsf{AP}(\mathsf{p}_i : S_i)_i \qquad \dfrac{\{(\mathsf{p}_i : S_i)_i\}\ \ \Phi \vdash \chi}{\{(\mathsf{p}_i : S_i)_i\}\ \ \Phi, \widetilde{\iota'^{-} : S_j}, \iota^{-} : S_j \vdash \chi[\mathsf{p}_j \mapsto \widetilde{\iota'} \cup \{\iota\}]}}{\Gamma; \Phi, \widetilde{\iota'^{-} : S_j}, \iota^{-} : S_j \vdash p(\chi[\mathsf{p}_j \mapsto \widetilde{\iota'} \cup \{\iota\}])} \qquad \overline{\Gamma; \iota^{+} : S_j \vdash \lightning \iota}}{\Gamma; \Phi, \widetilde{\iota'^{-} : S_j}, \iota^{+} : S_j, \iota^{-} : S_j, p \vdash p(\chi[\mathsf{p}_j \mapsto \widetilde{\iota'} \cup \{\iota\}]) \parallel \lightning \iota}}{\Gamma; \Phi, \widetilde{\iota'^{-} : S_j}, p \vdash (\nu\iota)(p(\chi[\mathsf{p}_j \mapsto \widetilde{\iota'} \cup \{\iota\}]) \parallel \lightning \iota)}$$

Recomposing:

$$\frac{p : \mathsf{AP}(\mathsf{p}_i : S_i)_i \qquad \dfrac{\{(\mathsf{p}_i : S_i)_i\}\ \ \Phi \vdash \chi}{\{(\mathsf{p}_i : S_i)_i\}\ \ \Phi, \widetilde{\iota'^{-} : S_j} \vdash \chi[\mathsf{p}_j \mapsto \widetilde{\iota'}]}}{\Gamma; \Phi, \widetilde{\iota'^{-} : S_j}, p \vdash p(\chi[\mathsf{p}_j \mapsto \widetilde{\iota'}])}$$

as required.

Case E-CancelH.

$$\langle a, \mathbf{idle}(U), \sigma[s[\mathsf{p}] \mapsto (V, W), \rho, \omega\rangle \parallel s \triangleright \delta \parallel \lightning s[\mathsf{q}] \xrightarrow{\tau} \langle a, W\ U, \sigma, \rho, \omega\rangle \parallel s \triangleright \delta \parallel \lightning s[\mathsf{q}] \parallel \lightning s[\mathsf{p}] \quad \text{if } \mathsf{messages}(\mathsf{q}, \mathsf{p}, \delta) = \emptyset$$

Let **D** be the following derivation:

$$\dfrac{\dfrac{\Gamma \vdash U : C}{\Gamma; \cdot \mid C \vdash \mathbf{idle}(U)} \qquad \dfrac{T = \mathsf{q}\,\&\{\ell_i(x_i) \mapsto S_i\}_i \qquad \Gamma \vdash V : \mathsf{Handler}(T,) \qquad \Gamma \vdash W : C \xrightarrow[C]{\mathsf{end},\mathsf{end}} C \qquad \Gamma; \Phi_1 \mid C \vdash \sigma}{\Gamma; \Phi_1, s[\mathsf{p}] : T \mid C \vdash \sigma[s[\mathsf{p}] \mapsto (V, W)]} \qquad \Gamma; \Phi_2 \mid C \vdash \rho \qquad \Gamma \vdash U : C}{\Gamma; \Phi_1, \Phi_2, s[\mathsf{p}] : T, a \vdash \langle a, \mathbf{idle}(U), \sigma[s[\mathsf{p}] \mapsto (V, W)], \rho, \omega\rangle}$$

Assumption:

$$\dfrac{\mathbf{D} \qquad \dfrac{\Gamma; s : Q \vdash s \triangleright \delta \qquad \Gamma; s[\mathsf{p}] : \lightning \vdash \lightning s[\mathsf{p}]}{\Gamma; s : Q, s[\mathsf{p}] : \lightning \vdash s \triangleright \delta \parallel \lightning s[\mathsf{p}]}}{\Gamma; \Phi_1, \Phi_2, s[\mathsf{p}] : T, s : Q, s[\mathsf{q}] : \lightning, a \vdash \langle a, \mathbf{idle}(U), \sigma[s[\mathsf{p}] \mapsto (V, W)], \rho, \omega\rangle \parallel s \triangleright \delta \parallel \lightning s[\mathsf{p}]}$$

We can recompose as follows. Let **D′** be the following derivation:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash W : C \xrightarrow[C]{\mathsf{end},\mathsf{end}} C \qquad \Gamma \vdash U : C}{\Gamma \mid C \mid \mathsf{end} \triangleright W\ U : C \triangleleft \mathsf{end}}}{\Gamma;\cdot \mid C \vdash W\ U} \qquad \Gamma;\Phi_1 \mid C \vdash \sigma \qquad \Gamma;\Phi_2 \mid C \vdash \rho}{\Gamma;\Phi_1,\Phi_2, a \vdash \langle a, W\ U, \sigma, \rho, \omega\rangle}$$

Then we can construct the remaining derivation:

$$\dfrac{\mathbf{D}' \qquad \dfrac{\Gamma;s : Q \vdash s \triangleright \delta \qquad \dfrac{\overline{\Gamma;s[\mathrm{p}] : \lightning \vdash \lightning s[\mathrm{p}]} \qquad \overline{\Gamma;s[\mathrm{q}] : \lightning \vdash \lightning s[\mathrm{q}]}}{\Gamma;s[\mathrm{p}] : \lightning, s[\mathrm{q}] : \lightning \vdash \lightning s[\mathrm{p}] \parallel \lightning s[\mathrm{q}]}}{\Gamma;s : Q, s[\mathrm{p}] : \lightning, s[\mathrm{q}] : \lightning \vdash s \triangleright \delta \parallel \lightning s[\mathrm{p}] \parallel \lightning s[\mathrm{q}]}}{\Gamma;\Phi_1,\Phi_2, s : Q, s[\mathrm{p}] : \lightning, s[\mathrm{q}] : \lightning, a \vdash \langle a, W\ U, \sigma, \rho, \omega\rangle \parallel s \triangleright \delta \parallel \lightning s[\mathrm{p}] \parallel \lightning s[\mathrm{q}]}$$

Finally, we need to show environment reduction:

$$\Phi_1,\Phi_2, s[\mathrm{p}] : T, s : Q, s[\mathrm{q}] : \lightning, a \xrightarrow{s:\mathrm{p}\lightning\mathrm{q}} \Phi_1,\Phi_2, s : Q, s[\mathrm{p}] : \lightning, s[\mathrm{q}] : \lightning, a$$

as required. □

C.1 Progress

Thread progress needs to change to take into account the possibility of an exception due to E-Raise or E-RaiseExn:

Lemma C.2 (Thread Progress). *Let $C = \mathcal{G}[\langle a, \mathcal{T}, \sigma, \rho\rangle]$. If $\cdot;\cdot \vdash C$ then either:*

- *$\mathcal{T} = \textbf{\textit{idle}}(V)$, or*
- *there exist $\mathcal{G}', \mathcal{T}', \sigma', \rho'$ such that $C \longrightarrow \mathcal{G}'[\langle a, \mathcal{T}', \sigma', \rho'\rangle]$, or*
- *$C \longrightarrow \mathcal{G}'[\lightning a \parallel \lightning \sigma \parallel \lightning \rho]$ if $\mathcal{T} = \mathcal{E}[\textbf{\textit{raise}}]$, or*
- *$C \longrightarrow \mathcal{G}'[\lightning a \parallel \lightning s[\mathrm{p}] \parallel \lightning \sigma \parallel \lightning \rho]$ if $\mathcal{T} = (\mathcal{E}[\textbf{\textit{raise}}])^{s[\mathrm{p}]}$.*

Proof. As with Lemma B.14 but taking into account that:

- **monitor** $b\ V$ can always reduce by E-Monitor;
- **raise** can always reduce by either E-Raise or E-RaiseS.

□

As before, all well-typed configurations can be written in canonical form; as usual the proof relies on the fact that structural congruence is type-preserving.

Lemma C.3. *If $\Gamma;\Phi \vdash C$ then there exists a $\mathcal{D} \equiv C$ where $\mathcal{D}$ is in canonical form.*

It is also useful to see that the progress property on environments is preserved even if some roles become cancelled.

Lemma C.4. *If $df_{\lightning}(\Phi)$ then $df_{\lightning}(zap(\Phi, \widetilde{\mathrm{p}}))$ for any $\widetilde{\mathrm{p}} \subseteq roles(\Phi)$.*

Proof. Zapping a role may prevent Lbl-Recv from firing, but in this case would enable either a Lbl-ZapRecv or Lbl-ZapMsg reduction. □

Theorem 5.4 (Progress ($\textsc{Maty}_{\lightning}$)). *If $\cdot;\cdot \vdash C$, then either there exists some $\mathcal{D}$ such that $C \longrightarrow \mathcal{D}$, or C is structurally congruent to the following canonical form:*

$$(\nu\tilde{\imath})(\nu p_{i\in 1..m})(\nu a_{j\in 1..n})(p_1(\chi_1)_{i\in 1..m} \parallel \langle a_j, \textbf{\textit{idle}}(U_j), \epsilon, \rho_j, \omega_j\rangle_{j\in 1..n'-1} \parallel (\lightning a_j)_{j\in n'..n})$$

Proof. The reasoning is similar to that of Theorem 4.5. By Lemma C.3, C can be written in canonical form:

$$(\nu\tilde{\imath})(\nu p_{i\in 1..l})(\nu s_{j\in 1..m})(\nu a_{k\in 1..n})(p_i(\chi_i)_{i\in 1..l} \parallel (s_j \triangleright \delta_j)_{j\in 1..m} \parallel \langle a_k, \mathcal{T}_k, \sigma_k, \rho_k, \omega_k\rangle_{k\in 1..n'-1} \parallel \widetilde{\lightning\,\alpha})$$

with $(\lightning\, a_k)_{k\in n'..n}$ contained in $\widetilde{\lightning\,\alpha}$.

By repeated applications of Lemma C.2, either the configuration can reduce or all threads are idle:

$$(\nu\tilde{\imath})(\nu p_{i\in 1..l})(\nu s_{j\in 1..m})(\nu a_{k\in 1..n})(p_i(\chi_i)_{i\in 1..l} \parallel (s_j \triangleright \delta_j)_{j\in 1..m} \parallel \langle a_k, \mathbf{idle}(U_k), \sigma_k, \rho_k, \omega_k\rangle_{k\in 1..n'-1} \parallel \widetilde{\lightning\,\alpha})$$

By the linearity of runtime type environments Δ, each role endpoint $s[\mathrm{p}]$ must either be contained in an actor, or exist as a zapper thread $\lightning\, s[\mathrm{p}] \in \widetilde{\lightning\,\alpha}$. Let us first consider the case that the endpoint is contained in an actor; we know by previous reasoning that each role must have an associated stored handler.

Since the types for each session must satisfy progress, the collection of local types must reduce. There are two potential reductions: either Lbl-Sync-Recv in the case that the queue has a message, or Lbl-ZapRecv if the sender is cancelled and the queue does not have a message. In the case of Lbl-Sync-Recv, since all actors are idle we can reduce using E-React as usual. In the case of Lbl-ZapRecv typing dictates that we have a zapper thread for the sender and so can reduce by E-CancelH.

It now suffices to reason about the case where all endpoints are zapper threads (and thus by linearity, where all handler environments are empty). In this case we can repeatedly reduce with E-CancelMsg until all queues are cleared, at which point we have a configuration of the form:

$$(\nu\tilde{\imath})(\nu p_{i\in 1..l})(\nu s_{j\in 1..m})(\nu a_{k\in 1..n})(p_i(\chi_i)_{i\in 1..l} \parallel (s_j \triangleright \epsilon)_{j\in 1..m} \parallel \langle a_k, \mathbf{idle}(U_k), \epsilon, \rho_k, \omega_k\rangle_{k\in 1..n'-1} \parallel \widetilde{\lightning\,\alpha})$$

We must now account for the remaining zapper threads. If there exists a zapper thread $\lightning\, a$ where a is contained within some monitoring environment ω then we can reduce with E-InvokeM. If a does not occur free in any initialisation callback or monitoring callback then we can eliminate it using the garbage collection congruence $(\nu a)(\lightning\, a) \parallel C \equiv C$.

Next, we eliminate all zapper threads for initialisation tokens using E-CancelAP.

Finally, we can eliminate all failed sessions $(\nu s)(\lightning\, s[\mathrm{p}_1] \parallel \cdots \parallel \lightning\, s[\mathrm{p}_n] \parallel s \triangleright \epsilon)$, and we are left with a configuration of the form:

$$(\nu\tilde{\imath})(\nu p_{i\in 1..m})(\nu a_{j\in 1..n})(p_1(\chi_1)_{i\in 1..m} \parallel \langle a_j, \mathbf{idle}(U_k), \epsilon, \rho_j, \omega_j\rangle_{j\in 1..n'-1} \parallel (\lightning\, a_j)_{j\in n'..n})$$

as required. □

C.1.1 Global Progress. A modified version of global progress holds: for every active session, in a finite number of reductions, either the session can make a communication action, or all endpoints become cancelled and can be garbage collected.

Theorem 5.5 (Global Progress (Maty$_\lightning$)). *If $\cdot;\cdot \vdash C$ where C is thread-terminating, then for every $s \in \mathit{activeSessions}(C)$, then there exist $\mathcal{D}$ and $\mathcal{D}_1$ such that $C \equiv (\nu s)\mathcal{D}$ where $\mathcal{D} \xrightarrow{\tau}{}^{*} \mathcal{D}_1$ and either $\mathcal{D}_1 \xrightarrow{s}$, or $\mathcal{D}_1 \equiv \mathcal{D}_2$ for some $\mathcal{D}_2$ where $s \notin \mathit{activeSessions}(\mathcal{D}_2)$.*

Proof. Follows the same structure as the proof of Corollary 4.13, the main difference being that instead of E-React firing, it may be the case that E-CancelH fires to propagate a failure. In this case, if all session endpoints for an active session s are cancelled, then it would be possible to use the garbage collection congruence to eliminate the failed session. □

Modified syntax

Session names	$\underline{s}, \underline{t}$		
Values	V, W	::=	$\cdots \mid (V, W)$
Computations	M, N	::=	$\cdots \mid \textbf{let}\ (x, y) = M\ \textbf{in}\ N$
		$\mid$	$\textbf{suspend}_!\ \underline{s}\ V\ W \mid \textbf{suspend}_?\ V\ W \mid \textbf{become}\ \underline{s}\ V$
Send-suspended sessions	D	::=	$(s[\mathrm{p}], V)$
Handler state	σ	::=	$\epsilon \mid \sigma, s[\mathrm{p}] \mapsto V \mid \sigma, \underline{s} \mapsto \overrightarrow{D}$
Switch request queue	θ	::=	$\epsilon \mid \theta \cdot (\underline{s}, V)$
Configurations	$\mathcal{C}, \mathcal{D}$	::=	$\cdots \mid \langle a, \mathcal{T}, \sigma, \rho, V\rangle\theta$

Modified typing rules $\boxed{\Gamma \vdash V : A}$ $\boxed{\Gamma \mid C \mid S \triangleright M : A \triangleleft T}$

$$\frac{\Gamma \vdash V : A \qquad \Gamma \vdash W : B}{\Gamma \vdash (V, W) : (A \times B)} \qquad \frac{\Gamma \vdash V : (A_1 \times A_2) \qquad \Gamma \mid C \mid S \triangleright M : B \triangleleft T}{\Gamma \mid C \mid S \triangleright \textbf{let}\ (x, y) = V\ \textbf{in}\ M : B \triangleleft T}$$

T-Suspend?

$$\frac{\Gamma \vdash V : \mathsf{Handler}(S^?, C) \qquad \Gamma \vdash W : C}{\Gamma \mid C \mid S^? \triangleright \textbf{suspend}_?\ V\ W : A \triangleleft T}$$

T-Suspend!

$$\frac{\Sigma(\underline{s}) = (S^!, A) \qquad \Gamma \vdash V : (A \times C) \xrightarrow[C]{S^!,\mathsf{end}} C}{\Gamma \mid C \mid S^! \triangleright \textbf{suspend}_!\ \underline{s}\ V : B \triangleleft T}$$

T-Become

$$\frac{\Sigma(\underline{s}) = (T, A) \qquad \Gamma \vdash V : A}{\Gamma \mid C \mid S \triangleright \textbf{become}\ \underline{s}\ V : \mathsf{Unit} \triangleleft S}$$

Modified configuration typing rules $\boxed{\Gamma;\Delta \vdash C}$ $\boxed{\Gamma;\Delta \mid C \vdash \sigma}$ $\boxed{\Gamma \vdash \theta}$

T-Actor

$$\frac{\Gamma;\Delta_1 \mid U \vdash \mathcal{T} \qquad \Gamma;\Delta_2 \mid U \vdash \sigma \qquad \Gamma;\Delta_3 \mid U \vdash \rho \qquad \Gamma \vdash \theta}{\Gamma;\Delta_1,\Delta_2,\Delta_3, a \vdash \langle a, \mathcal{T}, \sigma, \rho, \theta\rangle}$$

TH-SendHandler

$$\frac{\Gamma;\Delta \mid C \vdash \sigma \qquad \Sigma(\underline{s}) = (S^!, A) \qquad (\Gamma \vdash V_i : (A \times C) \xrightarrow[C]{S^!,\mathsf{end}} C)_i}{\Gamma;\Delta, (s_i[\mathrm{p}_i] : S^!)_i \mid C \vdash \sigma, \underline{s} \mapsto (s_i[\mathrm{p}_i], V_i)_i}$$

TR-Empty

$$\frac{}{\Gamma \vdash \epsilon}$$

TR-Request

$$\frac{\Gamma \vdash \theta \qquad \Sigma(\underline{s}) = (S^!, A) \qquad \Gamma \vdash V : A}{\Gamma \vdash \theta \cdot (\underline{s}, V)}$$

Modified reduction rules $\boxed{\mathcal{C} \longrightarrow \mathcal{D}}$

$$\begin{array}{lrcll}
\text{E-Suspend}_!\text{-1} & \langle a, (\mathcal{E}[\textbf{suspend}_!\ \underline{s}\ V\ W])^{s[\mathrm{p}]}, \sigma, \rho, \theta\rangle & \xrightarrow{\tau} & \langle a, \textbf{idle}(W), \sigma[\underline{s} \mapsto (s[\mathrm{p}], V)], \rho, \theta\rangle & (\underline{s} \notin \mathrm{dom}(\sigma)) \\
\text{E-Suspend}_!\text{-2} & \langle a, (\mathcal{E}[\textbf{suspend}_!\ \underline{s}\ V\ W])^{s[\mathrm{p}]}, \sigma[\underline{s} \mapsto \overrightarrow{D}], \rho, \theta\rangle & \xrightarrow{\tau} & \langle a, \textbf{idle}(W), \sigma[\underline{s} \mapsto \overrightarrow{D} \cdot (s[\mathrm{p}], V)], \rho, \theta\rangle & \\
\text{E-Become} & \langle a, \mathcal{M}[\textbf{become}\ \underline{s}\ V], \sigma, \rho, \theta\rangle & \xrightarrow{\tau} & \langle a, \mathcal{M}[\textbf{return}\ ()], \sigma, \rho, \theta \cdot (\underline{s}, V)\rangle & \\
\text{E-Activate} & \langle a, \textbf{idle}(U), \sigma[\underline{s} \mapsto (s[\mathrm{p}], V) \cdot \overrightarrow{D}], \rho, (\underline{s}, W) \cdot \theta\rangle & \xrightarrow{\tau} & \langle a, (V(W, U))^{s[\mathrm{p}]}, \sigma[\underline{s} \mapsto \overrightarrow{D}], \rho, \theta\rangle &
\end{array}$$

Fig. 17. Maty$_{\rightleftarrows}$: Modified syntax, typing, and reduction rules

D FORMAL MODEL OF SESSION SWITCHING EXTENSION

In Section 6 we described the implementation of Maty to support proactive switching between sessions. In this appendix we introduce a formal model of a similar feature that switches between sessions by queueing requests to invoke a send-suspended session, and activates send-suspended sessions when the event loop reverts to being idle.

Suppose that we want to adapt our Shop example to maintain a long-running session with a supplier and request a delivery whenever an item runs out of stock. The key difference to our original example is that we need to *switch* to the Restock session *as a consequence* of receiving a buy message in a customer session.

We can describe a Restock session with the following simple local types:

```
ShopRestock ≜                          SupplierRestock ≜
  μ loop.                                μ loop.
    Supplier ⊕ order(([ItemID] × Quantity)) .    Shop & order(([ItemID] × Quantity)) .
    Supplier & ordered(Quantity) . loop          Shop ⊕ ordered(Quantity) . loop
```

Whereas before we only needed to suspend an actor in a *receiving* state, this workflow requires us to also suspend an actor in a *sending* state, and switch into the session at a later stage. We call this extension Maty$_{\rightleftarrows}$. Below, we can see the extension of the shop example with the ability to switch into the restocking session; the new constructs are shaded.

```
ShopRestock ≜
  μ loop.
    Supplier ⊕ order(([ItemID] × Quantity)) .
    Supplier & ordered(Quantity) . loop

custReqHandler ≜
  handler Customer st {
    getItemInfo(itemID) ↦ [...]
    checkout((itemIDs, details)) ↦
      let items = get in
      if inStock(itemIDs, items) then [...]
      else
        Customer ! outOfStock();
        become Restock itemIDs;
        suspend? custReqHandler st
  }

shop ≜ λ(custAP, restockAP).
  register custAP Shop
    (λst.shop (custAP, Shop) λst. suspend? itemReqHandler st);
  register restockAP Shop (λst.  suspend! Restock restockHandler st);
                                 initialStock

restockHandler ≜ λ(itemIDs, st) .
  Supplier ! order((itemIDs, 10));
  suspend? (
    handler Supplier st {
      ordered(quantity) ↦
        increaseStock(itemIDs, quantity);
        suspend! Restock restockHandler st})
```

The program is implicitly parameterised by a mapping from static names like Restock to pairs of session types and payload types (in our scenario, Restock maps to (ShopRestock, [ItemID]) to show that an actor can suspend when its session type is ShopRestock, and must provide a list of ItemIDs when switching back into the session). We split the **suspend** construct into **suspend**$_?$ V (to suspend awaiting an incoming message, as previously), and **suspend**$_!$ s V (to suspend session with name s given a function V, until switched into), and introduce the **become** s V construct to switch into a suspended session. Specifically, **become** s V queues s to run when the actor is next idle. We modify the shop definition to also register with the *restockAP* access point, suspending the session (in a state that is ready to send) with the restockHandler. The restockHandler takes an item ID, sends an order message to the supplier, and suspends again.

Metatheory. Maty$_{\rightleftarrows}$ satisfies preservation. Since (by design) **become** operations are dynamic and not encoded in the protocol (for example, we might wish to queue two invocations of a send-suspended session to be executed in turn), there is no type-level mechanism of guaranteeing that a send-suspended session is invoked, so Maty$_{\rightleftarrows}$ instead enjoys progress up-to invocation of send-suspended sessions (see Appendix C).

Our extension to allow session switching is shown in Figure 17. We introduce a set of distinguished *session identifiers* $\underline{s}$; each session identifier is associated with a local type and a payload in an environment Σ, i.e., for each $\underline{s}$ we have $\Sigma(\underline{s}) = (S^!, A)$ for some $S^!, A$. We then split the **suspend** construct into two: **suspend**$_?$ V W (which, as before, installs a message handler V and suspends an actor with updated state W) and **suspend**$_!$ $\underline{s}$ V W, which suspends a session in a *send* state, installing a function V taking a payload of the given type. Finally we introduce a **become** $\underline{s}$ V construct that queues a request for the event loop to invoke $\underline{s}$ next time the actor is idle and a send-suspended session is available.

D.1 Metatheory

D.1.1 Preservation. As would be expected, Maty$_{\rightleftarrows}$ satisfies preservation.

Theorem D.1 (Preservation). *Preservation (as defined in Theorem 4.2) continues to hold in Maty$_{\rightleftarrows}$.*

Proof. Preservation of typing under structural congruence follows straightforwardly.

For preservation of typing under reduction, we proceed by induction on the derivation of $C \longrightarrow \mathcal{D}$.

Case E-Suspend$_!$-1.

Similar to E-Suspend$_!$-2.

Case E-Suspend$_!$-2.

$$\langle a, (\mathcal{E}[\mathbf{suspend}_!\ \underline{s}\ V\ W])^{s[p]}, \sigma[\underline{s} \mapsto \overrightarrow{D}], \rho, \theta\rangle \xrightarrow{\tau} \langle a, \mathbf{idle}(W), \sigma[\underline{s} \mapsto \overrightarrow{D} \cdot (s[p], V)], \rho, \theta\rangle$$

Assumption:

$$\dfrac{\dfrac{\Gamma \mid C \mid S \triangleright \mathcal{E}[\mathbf{suspend}_!\ \underline{s}\ V\ W] : C \triangleleft \mathsf{end}}{\Gamma; s[p] : S \mid C \vdash (\mathcal{E}[\mathbf{suspend}_!\ \underline{s}\ V\ W])^{s[p]}} \quad \dfrac{\Gamma;\Delta_1 \mid C \vdash \sigma \qquad \Sigma(\underline{s}) = (S^!, A) \qquad (\Gamma \vdash W_i : (A \times C) \xrightarrow[C]{S^!,\mathsf{end}} \mathsf{Unit})_i}{\Gamma;\Delta_1, (s_i[q_i] : S^!)_i \mid C \vdash \sigma[\underline{s} \mapsto (s_i[q_i], W_i)_i]} \quad \begin{array}{l}\Gamma;\Delta_2 \mid C \vdash \rho \\ \Gamma \vdash \theta\end{array}}{\Gamma;\Delta_1, \Delta_2, s[p] : S, (s_i[q_i] : S^!)_i, a \vdash \langle a, \mathcal{E}[\mathbf{suspend}_!\ \underline{s}\ V\ W], \sigma[\underline{s} \mapsto (s_i[q_i], W_i)_i], \rho, \theta\rangle}$$

Consider the subderivation $\Gamma \mid S \triangleright \mathcal{E}[\mathbf{suspend}_!\ \underline{s}\ V\ W] : \mathsf{Unit} \triangleleft \mathsf{end}$. By Lemma B.2 there exists a subderivation:

$$\dfrac{\Sigma(\underline{s}) = (S^!, A) \qquad \Gamma \vdash V : (A \times C) \xrightarrow[C]{S^!,\mathsf{end}} C \qquad \Gamma \vdash W : C}{\Gamma \mid S^! \triangleright \mathbf{suspend}_!\ \underline{s}\ V\ W : B \triangleleft \mathsf{end}}$$

Therefore we have that $S = S^!$.

Recomposing:

$$\dfrac{\dfrac{\Gamma \vdash W : C}{\Gamma; \cdot \mid C \vdash \mathbf{idle}(W)} \quad \dfrac{\Gamma;\Delta_1 \mid C \vdash \sigma \qquad \Sigma(\underline{s}) = (S^!, A) \qquad (\Gamma \vdash W_i : (A \times C) \xrightarrow[C]{S^!,\mathsf{end}} C)_i \qquad \Gamma \vdash V : (A \times C) \xrightarrow[C]{S^!,\mathsf{end}} C}{\Gamma;\Delta_1, (s_i[q_i] : S^!)_i, s[p] : S^! \mid C \vdash \sigma[\underline{s} \mapsto (s_i[q_i], W_i)_i \cdot (s[p], V)]} \quad \begin{array}{l}\Gamma;\Delta_2 \mid C \vdash \rho \\ \Gamma \vdash \theta\end{array}}{\Gamma;\Delta_1, \Delta_2, s[p] : S, (s_i[q_i] : S^!)_i, a \vdash \langle a, \mathbf{idle}(W), \sigma[\underline{s} \mapsto (s_i[q_i], W_i)_i \cdot (s[p], V)], \rho, \theta\rangle}$$

as required.

Case E-Become.

$$\langle a, \mathcal{M}[\textbf{become}\ \underline{s}\ V], \sigma, \rho, \theta\rangle \xrightarrow{\tau} \langle a, \mathcal{M}[\textbf{return}\ ()], \sigma, \rho, \theta \cdot (\underline{s}, V)\rangle$$

Assumption (considering the case that $\mathcal{M} = \mathcal{E}[-]$ for some $\mathcal{E}$; the case in the context of a session is identical):

$$\frac{\dfrac{\Gamma \mid S \mid C \triangleright \mathcal{E}[\textbf{become}\ \underline{s}\ V] : C \triangleleft \mathsf{end}}{\Gamma; \cdot \mid C \vdash \mathcal{E}[\textbf{become}\ \underline{s}\ V]} \qquad \begin{array}{l}\Gamma; \Delta_1 \mid C \vdash \sigma \\ \Gamma; \Delta_2 \mid C \vdash \rho \\ \Gamma \vdash \theta\end{array}}{\Gamma; \Delta_1, \Delta_2, a \vdash \langle a, \mathcal{T}, \sigma, \rho, \theta\rangle}$$

By Lemma B.2 we have:

$$\frac{\Sigma(\underline{s}) = (T, A) \qquad \Gamma \vdash V : A}{\Gamma \mid S \mid C \triangleright \textbf{become}\ \underline{s}\ V : \mathsf{Unit} \triangleleft S}$$

By Lemma B.3 we can show that $\Gamma \mid S \mid C \triangleright \mathcal{E}[\textbf{return}\ ()] : C \triangleleft \mathsf{end}$.

Recomposing:

$$\frac{\dfrac{\Gamma \mid C \mid S \triangleright \mathcal{E}[\textbf{return}\ ()] : \mathsf{Unit} \triangleleft \mathsf{end}}{\Gamma; \cdot \mid C \vdash \mathcal{E}[\textbf{return}\ ()]} \qquad \begin{array}{l}\Gamma; \Delta_1 \mid C \vdash \sigma \\ \Gamma; \Delta_2 \mid C \vdash \rho\end{array} \qquad \dfrac{\Gamma \vdash \theta \qquad \Sigma(\underline{s}) = (S^!, A) \qquad \Gamma \vdash V : A}{\Gamma \vdash \theta \cdot (\underline{s}, V)}}{\Gamma; \Delta_1, \Delta_2, a \vdash \langle a, \mathcal{E}[\textbf{return}\ ()], \sigma, \rho, \theta \cdot (\underline{s}, V)\rangle}$$

as required.

Case E-Activate.

$$\langle a, \textbf{idle}(U), \sigma[\underline{s} \mapsto (s[\mathsf{p}], V) \cdot \overrightarrow{D}], \rho, (\underline{s}, W) \cdot \theta\rangle \xrightarrow{\tau} \langle a, (V\ (W, U))^{s[\mathsf{p}]}, \sigma[\underline{s} \mapsto \overrightarrow{D}], \rho, \theta\rangle$$

Let **D** be the subderivation:

$$\frac{\Sigma(\underline{s}) = (S^!, A) \qquad \Gamma \vdash V : (A \times C)\ \xrightarrow[C]{S^!, \mathsf{end}}\ C \qquad (\Gamma \vdash V_i : (A \times C)\ \xrightarrow[C]{S^!, \mathsf{end}}\ C)_i \qquad \Gamma; \Delta_1 \mid C \vdash \sigma}{\Gamma; \Delta_1, s[\mathsf{p}] : S^!, (s_i[\mathsf{p}_i] : S^!)_i \mid C \vdash \sigma, \underline{s} \mapsto (s[\mathsf{p}], V) \cdot (s_i[\mathsf{p}_i], V_i)_i}$$

Assumption:

$$\frac{\dfrac{\Gamma \vdash U : C}{\Gamma; \cdot \mid C \vdash \textbf{idle}(U)} \quad \mathbf{D} \quad \Gamma; \Delta_2 \mid C \vdash \rho \qquad \dfrac{\Gamma \vdash \theta \qquad \Sigma(\underline{s}) = (S^!, A) \qquad \Gamma \vdash W : A}{\Gamma \vdash (\underline{s}, W) \cdot \theta}}{\Gamma; \Delta_1, \Delta_2, s[\mathsf{p}] : S^!, (s_i[\mathsf{p}_i] : S^!)_i, a \vdash \langle a, \textbf{idle}(U), \sigma[\underline{s} \mapsto (s[\mathsf{p}], V) \cdot (s_i[\mathsf{p}_i], V_i)_i], \rho, (\underline{s}, W) \cdot \theta\rangle}$$

Let $\mathbf{D}'$ be the subderivation:

$$\frac{\dfrac{\Gamma \vdash V : A\ \xrightarrow[C]{S^!, \mathsf{end}}\ C \qquad \dfrac{\Gamma \vdash W : A \qquad \Gamma \vdash U : C}{\Gamma \vdash (W, U) : (A \times C)}}{\Gamma \mid C \mid S^! \triangleright V\ (W, U) : C \triangleleft \mathsf{end}}}{\Gamma; s[\mathsf{p}] : S^! \mid C \vdash (V\ (W, U))^{s[\mathsf{p}]}}$$

Recomposing:

$$\mathbf{D}\quad \frac{\dfrac{\Sigma(\underline{s}) = (S^!, A) \qquad (\Gamma \vdash V_i : (A \times C) \xrightarrow{S^!,\mathsf{end}}_{C} C)_i}{\Gamma; \Delta_1, (s_i[\mathrm{p}_i] : S^!)_i \mid C \vdash \sigma, \underline{s} \mapsto (s_i[\mathrm{p}_i], V_i)_i} \qquad \Gamma; \Delta_2 \mid C \vdash \rho \qquad \Gamma \vdash \theta}{\Gamma; \Delta_1, \Delta_2, s[\mathrm{p}] : S^!, (s_i[\mathrm{p}_i] : S^!)_i, a \vdash \langle a, (V\ (W, U))^{s[\mathrm{p}]}, \sigma[\underline{s} \mapsto (s_i[\mathrm{p}_i], V_i)_i], \rho, \theta\rangle}$$

as required. □

D.1.2 Progress. Since (by design) **become** operations are dynamic and not encoded in the protocol (for example, we might wish to queue two invocations of a send-suspended session to be executed in turn), there is no type-level mechanism of guaranteeing that a send-suspended session is ever invoked. Although all threads can reduce as before, $\mathsf{Maty}_{\rightleftarrows}$ satisfies a weaker version of progress where non-reducing configurations can contain send-suspended sessions.

Theorem D.2 (Progress ($\mathsf{Maty}_{\rightleftarrows}$)). *If $\cdot;\cdot \vdash_{df} C$, then either there exists some $\mathcal{D}$ such that $C \longrightarrow \mathcal{D}$, or C is structurally congruent to the following canonical form:*

$$(\nu\tilde{\imath})(\nu p_{i\in 1..l})(\nu s_{j\in 1..m})(\nu a_{k\in 1..n})(p_i(\chi_i)_{i\in 1..l} \parallel (s_j \triangleright \delta_j)_{j\in 1..m} \parallel \langle a_k, \textbf{\textit{idle}}(U_k), \sigma_k, \rho_k, \theta_k\rangle_{k\in 1..n})$$

where for each session s_j there exists some mapping $s_j[\mathrm{p}] \mapsto (\underline{s}, V)$ (for some role p, static session name $\underline{s}$, and callback V) contained in some σ_k where θ_k does not contain any requests for $\underline{s}$.

Proof. The proof follows that of Theorem 4.5. Thread progress (Lemma B.14) holds as before, since we can always evaluate **become** by E-Become, and we can always evaluate $\mathbf{suspend}_!$ by E-Suspend-$!_1$ or E-Suspend-$!_2$.

Following the same reasoning as Theorem 4.5 we can write C in canonical form, where all threads are idle:

$$(\nu\tilde{\imath})(\nu p_{i\in 1..l})(\nu s_{j\in 1..m})(\nu a_{k\in 1..n})(p_i(\chi_i)_{i\in 1..l} \parallel (s_j \triangleright \delta_j)_{j\in 1..m} \parallel \langle a_k, \mathbf{idle}(V_k), \sigma_k, \rho_k, \theta_k\rangle_{k\in 1..n})$$

However, there are now *three* places each role endpoint $s[\mathrm{p}]$ can be used: either by TT-Sess to run a term in the context of a session or by TH-Handler to record a receive-suspended session type as before, but now also by TH-SendHandler to record a send-suspended session type. As before, the former is impossible as all threads are idle, so now we must consider the cases for TH-Handler.

Following the same reasoning as Theorem 4.5, we can reduce any handlers that have waiting messages. Thus we are finally left with the scenario where the session type LTS can reduce, but not the configuration: this can only happen when the sending reduction is send-suspended, as required. □