Speak Now

Safe Actor Programming with Multiparty Session Types

DRAFT: March 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 both Erlang-style cascading failure handling and the ability to proactively switch between sessions. 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

The infrastructure underpinning our daily lives is powered by 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 such as Akka, are popular tools for writing scalable and resilient distributed applications: Erlang in particular powers the servers of WhatsApp, which has billions of users worldwide. Actor languages support lightweight processes that communicate through point-to-point asynchronous explicit message passing as opposed to shared memory, making them easy to implement in a distributed setting and enabling them to support 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) 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 **present approaches to using session types in actor languages are severely limited in expressiveness**. Other behavioural type disciplines for actors make it difficult to express *structured* interactions, and *cannot adequately handle failure.*

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

In this paper we present Maty, the first actor-based programming language fully supporting statically-checked multiparty session types and failure handling, allowing developers to benefit from both 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.*

1.1 Actor Languages

Actor languages and frameworks are inspired by the *actor model* [2, 21], 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:

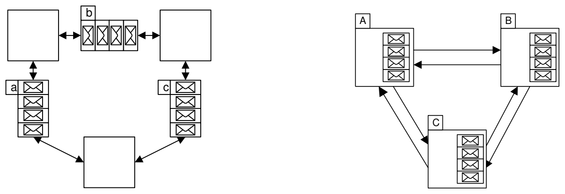
$$\mathsf{Server} = a(x).(P_{\mathsf{thread}}(x) \mid \mathsf{Server}) \qquad \text{or} \qquad \mathsf{Server} = !\, a(x).P_{\mathsf{thread}}(x)$$

```
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) Channels (b) Actors

Fig. 2. Channel- and actor-based languages [15]

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.

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., $\mathsf{Chan}(\mathsf{Int})$ or a *session type* such as $!\mathsf{Int}.!\mathsf{Int}.?\mathsf{Bool}.\mathsf{end}$ 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 [7]. 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 *hard 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 [34] 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 [16] 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 **KP1–5**. Harvey et al. [20] used multiparty session types in an actor language to support safe runtime adaptation, 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 [37] and Fowler [12] 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 describe two extensions to Maty: the ability to proactively switch to another session, and support for Erlang-style process supervision and cascading failure (§5).
(3) We detail our implementation of Maty using an API generation approach in Scala (§6), and demonstrate our implementation on a real-world case study from the factory domain as well as a chat server application.

Section 7 discusses related work, and Section 8 concludes. **We will submit our implementation and examples as an artifact.**

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.

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.

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 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.
- Once an actor is ready to receive a message, it suspends by installing a *message handler*, and reverts to an idle state. Suspension acts as a *yield point* to the event loop, and occurs at **precisely the same point as in real-world 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 sometimes omit effect annotations where they are not necessary.

```
idServer : AP(IDServerProtocol) ⟶ Unit
idServer = λap.
  register ap Server
    (idServer ap; suspend requestHandler)

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

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

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

Fig. 4. Maty implementation of ID Server

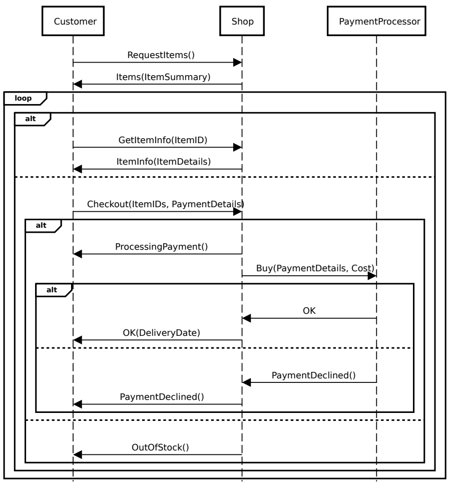
The idServer takes an *access point* [17] for the IDServerProtocol protocol as an argument, and registers for the Server role. An access point can be thought of as a "matchmaking service". Actors *register* to play a role in a session, and the access point establishes a session once at least one actor has registered for every role. The **register** construct takes three arguments: an access point, the role to register for, 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 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. An actor *installs* a message handler for the current session by invoking the **suspend** construct, which reverts the actor back to being idle and states 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, 11, 18] 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 pre-condition becomes 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 Unit 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 (we omit the details). Finally, `main` sets up the access point (associating Client with ClientTy and Server with ServerTy), then spawns idServer(*idServerAP*)



- 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.
- Separately, **Staff** can also log in using a different system and adjust the stock.

Fig. 5. Online Shop Scenario

and client($idServerAP$). The server starts with state $(0, \textbf{false})$; the client uses a dummy state. Both then register with the access point, which establishes the session.

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 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.

Session types. Figure 6 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. Note that ShopAdminTy is a different type of session altogether, detailing the interactions between the shop and the admin interface.

Shop message handlers. Figure 7 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.

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.

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}, and let StaffProtocol $=$ {Shop : ShopAdminTy, Staff : StaffTy}.

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

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

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

ShopAdminTy ≜
  Staff &{
    addItem((Name × Description × Price × Stock)) .
      ShopAdminTy,
    removeItem(ItemID) . ShopAdminTy}
```

Fig. 6. Local types for the Shop role

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

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

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

staffReqHandler : Handler(ShopAdminTy, [Item])
staffReqHandler ≜
    handler Staff {
      addItem((name, description, price, stock)) ↦
        let items = get in
        set add(name, description, price, stock, items)
        suspend staffReqHandler
      removeItem(itemID) ↦
        let items = get in
        set remove(itemID, items);
        suspend staffReqHandler
    }
```

Fig. 7. Implementation of Shop message handlers in Maty

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

```
registerForever(ap, role, callback) ≜
    register ap role (registerForever(ap, role, callback));
    callback ()

shop(custAP, staffAP) ≜
  register custAP Shop
    (registerForever(custAP, Shop, λ(). suspend itemReqHandler));
  register staffAP Shop
    (registerForever(staffAP, Shop, λ(). suspend staffReqHandler))
```

Syntax of types and type environments

$$
\begin{array}{llcl}
\text{Output session types} & S^{!} & ::= & \mathsf{p} \oplus \{\ell_i(A_i).S_i\}_i \\
\text{Input session types} & S^{?} & ::= & \mathsf{p} \,\&\, \{\ell_i(A_i).S_i\}_i \\
\text{Session types} & S, T & ::= & S^{!} \mid S^{?} \mid \mu X.S \\
 & & \mid & X \mid \mathsf{end}
\end{array}
\qquad
\begin{array}{llcl}
\text{Types} & A, B, C & ::= & D \mid A \xrightarrow[C]{S,T} B \mid \mathsf{AP}((\mathsf{p}_i : S_i)_i) \\
 & & \mid & \mathsf{Handler}(S^{?}, A) \\
\text{Base types} & D & ::= & \mathsf{Unit} \mid \mathsf{Bool} \mid \mathsf{Int} \mid \cdots \\
\text{Type envs.} & \Gamma & ::= & \cdot \mid \Gamma, x : A
\end{array}
$$

Value typing $\boxed{\Gamma \vdash_{\varphi} V : A}$

$$
\textsc{TV-Var}\ \frac{x : A \in \Gamma}{\Gamma \vdash x : A}
\qquad
\textsc{TV-Const}\ \frac{c \text{ has base type } D}{\Gamma \vdash c : D}
\qquad
\textsc{TV-Lam}\ \frac{\Gamma, x : A \mid C \mid S \triangleright M : B \triangleleft T}{\Gamma \vdash \lambda x.\, M : A \xrightarrow[C]{S,T} B}
$$

$$
\textsc{TV-Rec}\ \frac{\Gamma, x : A, f : A \xrightarrow[C]{S,T} B \mid C \mid S \triangleright M : A \xrightarrow[C]{S,T} B \triangleleft T}{\Gamma \vdash \mathbf{rec}\, f(x).M : A \xrightarrow[C]{S,T} B}
\qquad
\textsc{TV-Handler}\ \frac{(\Gamma, x : A_i \mid C \mid S_i \triangleright M_i : \mathsf{Unit} \triangleleft \mathsf{end})_i}{\Gamma \vdash \mathbf{handler}\ \mathsf{p}\ \{\ell_i(x_i) \mapsto M_i\}_i : \mathsf{Handler}(\mathsf{p} \,\&\, \{\ell_i(A_i).S_i\}_i, C)}
$$

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

$$
\textsc{T-Let}\ \frac{\Gamma \mid C \mid S_1 \triangleright M : A \triangleleft S_2 \qquad \Gamma, x : A \mid C \mid S_2 \triangleright N : B \triangleleft S_3}{\Gamma \mid C \mid S_1 \triangleright \mathbf{let}\ x \Leftarrow M\ \mathbf{in}\ N : B \triangleleft S_3}
\qquad
\textsc{T-Return}\ \frac{\Gamma \vdash V : A}{\Gamma \mid C \mid S \triangleright \mathbf{return}\ V : A \triangleleft S}
\qquad
\textsc{T-App}\ \frac{\Gamma \vdash V : A \xrightarrow[C]{S,T} B \qquad \Gamma \vdash W : A}{\Gamma \mid C \mid S \triangleright V\ W : B \triangleleft T}
$$

$$
\textsc{T-If}\ \frac{\Gamma \vdash V : \mathsf{Bool} \qquad \Gamma \mid C \mid S_1 \triangleright M : A \triangleleft S_2 \qquad \Gamma \mid C \mid S_1 \triangleright N : A \triangleleft S_2}{\Gamma \mid C \mid S_1 \triangleright \mathbf{if}\ V\ \mathbf{then}\ M\ \mathbf{else}\ N : A \triangleleft S_2}
\qquad
\textsc{T-Get}\ \frac{}{\Gamma \mid A \mid S \triangleright \mathbf{get} : A \triangleleft T}
\qquad
\textsc{T-Set}\ \frac{\Gamma \vdash V : A}{\Gamma \mid A \mid S \triangleright \mathbf{set}\ V : \mathsf{Unit} \triangleleft S}
$$

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

$$
\textsc{T-NewAP}\ \frac{\varphi \text{ is a safety property} \qquad \varphi((\mathsf{p}_i : T_i)_{i \in I})}{\Gamma \mid C \mid S \triangleright \mathbf{newAP}[(\mathsf{p}_i : T_i)_{i \in I}] : \mathsf{AP}((\mathsf{p}_i : T_i)_{i \in I}) \triangleleft S}
\qquad
\textsc{T-Register}\ \frac{j \in I \qquad \Gamma \vdash V : \mathsf{AP}((\mathsf{p}_i : T_i)_{i \in I}) \qquad \Gamma \mid C \mid T_j \triangleright M : \mathsf{Unit} \triangleleft \mathsf{end}}{\Gamma \mid C \mid S \triangleright \mathbf{register}\ V\ \mathsf{p}_j\ M : \mathsf{Unit} \triangleleft S}
$$

Fig. 8. Maty Static Semantics

The shop definition takes the two access points and then proceeds to *register* to take part both in a session to interact with customers, and also to interact with staff. The registerForever meta-level definition ensures that the actor re-registers whenever a session is established, meaning that the shop can accept an unlimited number of clients. 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 and staffReqHandler respectively.

3 MATY: A CORE ACTOR LANGUAGE WITH MULTIPARTY SESSION TYPES

3.1 Syntax and Typing Rules

Figure 8 shows the static semantics 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* [30], with different typing judgements for each. Unlike many session type systems, we do not need linear types when typing values or computations as session typing is enforced by effect typing; our approach is inspired by that of Harvey et al. [20].

Session types. Although global types are convenient for describing protocols, we instead follow Scalas and Yoshida [44] and base our formalism around local types (*projection* of global types onto roles is standard [22, 42]; the local types resulting from a projecting a global type satisfy the properties that we will see in §4 [44]). *Selection* session types $\mathsf{p} \oplus \{\ell_i(A_i) \,.\, S_i\}_{i \in I}$ 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 $\mathsf{p} \,\&\{\ell_i(A_i) \,.\, S_i\}_{i \in I}$ indicate that a process 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.

Types. Base types D are standard. Since our type system enforces session typing by pre- and post-conditions and also allows effectful state updates, 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, and can manipulate state of type C. An access point has type $\mathsf{AP}((\mathsf{p}_i : S_i)_i)$, 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.

Values. The value typing judgement has the form $\Gamma \vdash_\varphi V : A$ (we will return to behavioural properties φ in §4, and omit φ from the rules to avoid clutter). Typing rules for variables and constants are standard (we assume constants include at least the unit value () of type Unit), and typing rules for anonymous functions and anonymous recursive functions are adapted to include session pre- and postconditions. A *message handler* **handler** $\mathsf{p}\ \{\ell_i(x_i) \mapsto M_i\}_i$ specifies an actor's behaviour when a message is received from role p; for each clause with message label ℓ_i, the payload is bound to x_i in M_i. Rule TV-Handler states that the handler 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 all branches have the postcondition end.

Computations. The computation typing judgement has the form $\Gamma \mid S \mid M \triangleright_\varphi A : T \triangleleft C$, read as "under type environment Γ, and session precondition S, term M has type A and postcondition T and manipulates state of type C". Again, φ refers to a behavioural property and will be discussed in §4.

A let-binding **let** $x \Leftarrow M$ **in** N evaluates M and binds its result to x in N, with the session postcondition from typing M used as the precondition 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-If types a conditional if its condition is of type Bool and both continuations have the same return type and postcondition. Rules T-Get and T-Set handle state access and mutation.

The **spawn** $M\ V$ term spawns a new actor that evaluates term M and has initial state V; rule T-Spawn states that computation M must have return type Unit and pre- and postconditions end (since the spawned computation is not yet in a session and so cannot communicate). Note that the computation's state type can differ to that of the current actor. Rule T-Send types a send computation $\mathsf{p}\,!\,\ell(V)$ 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

Runtime syntax

$$
\begin{array}{llll}
\text{Actor names} & a, b & & \\
\text{Session names} & s & & \\
\text{AP names} & p & & \\
\text{Init. tokens} & \iota & & \\
\text{Runtime names} & \alpha & ::= & a \mid s \mid p \mid \iota \\
\text{Values} & U, V, W & ::= & \cdots \mid p \\
\text{Type env.} & \Gamma & ::= & \cdots \\
 & & \mid & \Gamma, p : \mathsf{AP}((\mathsf{p}_i : S_i)_i) \\
\text{Reduction labels} & l & ::= & s \mid \tau
\end{array}
$$

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

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

$$
C \parallel \mathcal{D} \equiv \mathcal{D} \parallel C \qquad C \parallel (\mathcal{D} \parallel \mathcal{D}') \equiv (C \parallel \mathcal{D}) \parallel \mathcal{D}' \qquad \frac{\alpha \notin \mathsf{fn}(C)}{C \parallel (\nu\alpha)\mathcal{D} \equiv (\nu\alpha)(C \parallel \mathcal{D})} \qquad \frac{}{(\nu s)(s \triangleright \epsilon) \parallel C \equiv C}
$$

$$
\frac{\mathsf{p}_1 \neq \mathsf{p}_2 \lor \mathsf{q}_1 \neq \mathsf{q}_2}{s \triangleright \sigma_1 \cdot (\mathsf{p}_1, \mathsf{q}_1, \ell_1(V_1)) \cdot (\mathsf{p}_2, \mathsf{q}_2, \ell_2(V_2)) \cdot \sigma_2 \equiv s \triangleright \sigma_1 \cdot (\mathsf{p}_2, \mathsf{q}_2, \ell_2(V_2)) \cdot (\mathsf{p}_1, \mathsf{q}_1, \ell_1(V_1)) \cdot \sigma_2}
$$

Fig. 9. Operational semantics (1)

wishes to receive a message, it must *suspend* itself and install a message handler using **suspend** V. The T-Suspend rule states that **suspend** V is typable if the handler is compatible with the current 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 annotated with the set of roles and local types to be involved in the session (T-NewAP). The rule ensures that the session types satisfy a *safety property*; we will describe this further in §4, but at a high level, if a set of session types is safe then the types are guaranteed never to cause a runtime type error due to a communication mismatch.

An actor can *register* to take part in a session as role p on access point V using **register** V p M; term M 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.

3.2 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 5-tuple $\langle a, \mathcal{T}, \sigma, \rho, U \rangle$, where $\mathcal{T}$ is a thread that can either be **idle**; 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. We say that 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

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

E-GET
$$\frac{}{\langle a, \mathcal{M}[\mathbf{get}], \sigma, \rho, V\rangle \xrightarrow{\tau} \langle a, \mathcal{M}[\mathbf{return}\ V], \sigma, \rho, V\rangle}$$

E-SET
$$\frac{}{\langle a, \mathcal{M}[\mathbf{set}\ W], \sigma, \rho, V\rangle \xrightarrow{\tau} \langle a, \mathcal{M}[\mathbf{return}\ ()], \sigma, \rho, W\rangle}$$

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

E-REACT
$$\frac{(\ell(x) \mapsto M) \in \overrightarrow{H}}{\begin{array}{c}\langle a, \mathbf{idle}, \sigma[s[\mathsf{p}] \mapsto \mathbf{handler}\ \mathsf{q}\ \{\overrightarrow{H}\}], \rho, U\rangle \parallel s \triangleright (\mathsf{q}, \mathsf{p}, \ell(V)) \cdot \delta \xrightarrow{s} \\ \langle a, (M\{V/x\})^{s[\mathsf{p}]}, \sigma, \rho, U\rangle \parallel s \triangleright \delta\end{array}}$$

E-SUSPEND
$$\frac{}{\begin{array}{c}\langle a, (\mathcal{E}[\mathbf{suspend}\ V])^{s[\mathsf{p}]}, \sigma, \rho, U\rangle \xrightarrow{\tau} \\ \langle a, \mathbf{idle}, \sigma[s[\mathsf{p}] \mapsto V], \rho, U\rangle\end{array}}$$

E-SPAWN
$$\frac{}{\begin{array}{c}\langle a, \mathcal{M}[\mathbf{spawn}\ M\ V], \sigma, \rho, U\rangle \xrightarrow{\tau} \\ (\nu b)(\langle a, \mathcal{M}[\mathbf{return}\ ()], \sigma, \rho, U\rangle \parallel \langle b, M, \epsilon, \epsilon, V\rangle)\end{array}}$$

E-RESET
$$\frac{}{\begin{array}{c}\langle a, \mathcal{Q}[\mathbf{return}\ ()], \sigma, \rho, U\rangle \xrightarrow{\tau} \\ \langle a, \mathbf{idle}, \sigma, \rho, U\rangle\end{array}}$$

E-NEWAP
$$\frac{p \text{ fresh}}{\begin{array}{c}\langle a, \mathcal{M}[\mathbf{newAP}[(\mathsf{p}_i : S_i)_{i \in I}]], \sigma, \rho, U\rangle \xrightarrow{\tau} \\ (\nu p)(\langle a, \mathcal{M}[\mathbf{return}\ p], \sigma, \rho, U\rangle \parallel p((\mathsf{p}_i \mapsto \epsilon)_{i \in I}))\end{array}}$$

E-REGISTER
$$\frac{\iota \text{ fresh}}{\begin{array}{c}\langle a, \mathcal{M}[\mathbf{register}\ p\ \mathsf{p}\ M], \sigma, \rho, U\rangle \parallel p(\chi[\mathsf{p} \mapsto \widetilde{\iota'}]) \xrightarrow{\tau} \\ (\nu \iota)(\langle a, \mathcal{M}[\mathbf{return}\ ()], \sigma, \rho[\iota \mapsto M], U\rangle \parallel p(\chi[\mathsf{p} \mapsto \widetilde{\iota'} \cup \{\iota\}]))\end{array}}$$

E-INIT
$$\frac{s \text{ fresh}}{\begin{array}{c}(\nu \iota_{\mathsf{p}_i})_{i \in 1..n}(p((\mathsf{p}_i \mapsto \widetilde{\iota'_{\mathsf{p}_i}} \cup \{\iota_{\mathsf{p}_i}\})_{i \in 1..n}) \parallel \langle a_i, \mathbf{idle}, \sigma_i, \rho_i[\iota_{\mathsf{p}_i} \mapsto M_i]_{i \in 1..n}, U\rangle) \xrightarrow{\tau} \\ (\nu s)(p((\mathsf{p}_i \mapsto \widetilde{\iota'_{\mathsf{p}_i}})_{i \in 1..n}) \parallel s \triangleright \epsilon \parallel \langle a_i, (M_i)^{s[\mathsf{p}_i]}, \sigma_i, (\rho_i)_{i \in 1..n}, U\rangle)\end{array}}$$

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

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

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)

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. Finally, U is a value representing the actor's state. Our reduction rules (Figure 10) make use of indexing notation as syntactic sugar for parallel composition: for example, $\langle a_i, \mathcal{T}_i, \sigma_i, \rho_i, U_i\rangle_{i \in 1..n}$ is syntactic sugar for the configuration $\langle a_1, \mathcal{T}_1, \sigma_1, \rho_1, U_1\rangle \parallel \cdots \parallel \langle a_n, \mathcal{T}_n, \sigma_n, \rho_n, U_n\rangle$.

An access point $p(\chi)$ has name p and state χ, where the state maps roles to lists 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 well-typed static term M is run by placing it in an *initial configuration* of the form $(\nu a)(\langle a, M, \epsilon, \epsilon, ()\rangle)$.

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).

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 $\boxed{\Delta \xrightarrow{\gamma} \Delta'}$

Lbl-Send	$\Delta, s[\mathsf{p}] : \mathsf{q} \oplus \{\ell_i(A_i).S_i\}_{i \in I}, s : Q$	$\xrightarrow{s:\mathsf{p}\uparrow\mathsf{q}::\ell_j}$	$\Delta, s[\mathsf{p}] : S_j, s : Q \cdot (\mathsf{p}, \mathsf{q}, \ell_j(A_j))$	(if $j \in I$)
Lbl-Recv	$\Delta, s[\mathsf{p}] : \mathsf{q} \& \{\ell_i(A_i).S_i\}_{i \in I}, s : (\mathsf{q}, \mathsf{p}, \ell_j(A_j)) \cdot Q$	$\xrightarrow{s:\mathsf{q}\downarrow\mathsf{p}::\ell_j}$	$\Delta, s[\mathsf{p}] : S_j, s : Q$	(if $j \in I$)
Lbl-End	$\Delta, s[\mathsf{p}] : \mathsf{end}$	$\xrightarrow{\mathsf{end}(s,\mathsf{p})}$	Δ	
Lbl-Rec	$\Delta, s[\mathsf{p}] : \mu X.S$	$\xrightarrow{\gamma}$	Δ'	(if $\Delta, s[\mathsf{p}] : S\{\mu X.S/X\} \xrightarrow{\gamma} \Delta'$)

Fig. 11. Labelled transition system on runtime type environments

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 a process 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, has a stored handler $\ell(x) \mapsto M$ for $s[\mathsf{p}]$, and there exists a matching message in the session queue, then the message is dequeued and the message handler is activated. If an actor is currently evaluating a computation in the context of a session $s[\mathsf{p}]$, rule E-Suspend evaluates **suspend** V by installing handler V for $s[\mathsf{p}]$ and returning the actor to the **idle** state.

Rule E-Spawn spawns a fresh actor with empty handler and initialisation state, and E-Reset returns an actor to the **idle** state once it has finished evaluating.

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 M by creating an initialisation token ι, storing a mapping from ι to the callback M in the requesting actor's initialisation environment, 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. The remaining rules are administrative.

4 METATHEORY

In order to prove metatheoretical properties about Maty, we define an extrinsic [41] 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 [44] we begin by showing a type semantics for sets of local types. Using this semantics we can define behavioural properties on types (such as *safety*, which ensures that communicated messages are always compatible; and *progress*, which ensures communication is deadlock-free). By making our configuration typing rules *parametric* in the particular behavioural property used, we can customise the property to show that behavioural properties on types give rise to corresponding guarantees about the behaviour of 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 channel endpoints, and also to ensure that there is precisely one instance of each actor and access point. Runtime type environments can contain 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 environment); session channel endpoints $s[\mathrm{p}] : S$; and finally session queue types $s : Q$. Queue types mirror the structure of queue entries and are a triple $(\mathrm{p}, \mathrm{q}, \ell(A))$. 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.

Safety property. Safety is the minimum property we require for type preservation: it ensures that communication does not introduce type errors. Intuitively a safety property 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.

Definition 4.1 (Safety property). φ is a *safety property* of runtime type environments Δ if:

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

A runtime environment is *safe*, written $\mathsf{safe}(\Delta)$, if $\varphi(\Delta)$ for a safety property φ.

We henceforth assume that all other properties are safety properties. Although checking safety for an asynchronous multiparty protocol is undecidable in general [44], there are various computationally tractable ways of ensuring that a protocol is safe. For example, syntactic projections from global types produce safe and deadlock-free sets of local types [44]. Furthermore, multiparty compatibility [10] allows safety to be verified by bounded model checking; this is the core approach implemented in Scribble [25], used by our implementation.

We have therefore designed our type system to be agnostic of any specific implementation method for validating safety, as common in recent MPST language design papers (e.g., [20, 29]).

4.1 Configuration typing

Figure 12 shows the typing rules for Maty configurations.

Configuration typing rules. The configuration typing judgement $\Gamma; \Delta \vdash_{\varphi} C$ can be read, "under type environment Γ and runtime type environment Δ, where the session types used in each session must satisfy behavioural property φ, configuration C is well typed". We omit φ from the rules to avoid clutter, and write $\Gamma; \Delta \vdash C$ when we wish to consider the largest safety property.

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

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

$$\textsc{T-SessionName}\ \frac{\Delta' = \{s[\mathsf{p}_i] : S_{\mathsf{p}_i}\}_i, s : Q \qquad \varphi(\Delta') \qquad s \notin \Delta \qquad \Gamma; \Delta, \Delta' \vdash C \qquad \varphi \text{ is a safety property}}{\Gamma; \Delta \vdash (\nu s)C}$$

$$\textsc{T-ActorName}\ \frac{\Gamma; \Delta, a \vdash C}{\Gamma; \Delta \vdash (\nu a)C} \qquad \textsc{T-Par}\ \frac{\Gamma; \Delta_1 \vdash \mathcal{C} \qquad \Gamma; \Delta_2 \vdash \mathcal{D}}{\Gamma; \Delta_1, \Delta_2 \vdash \mathcal{C} \parallel \mathcal{D}}$$

$$\textsc{T-AP}\ \frac{p : \mathsf{AP}((\mathsf{p}_i : S_i)_i) \in \Gamma \qquad \{(\mathsf{p}_i : S_i)_i\}\ \Delta \vdash \chi \qquad \varphi((\mathsf{p}_i : S_i)_i) \qquad \varphi \text{ is a safety property}}{\Gamma; \Delta, p \vdash p(\chi)}$$

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

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

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

$$\textsc{TT-Idle}\ \frac{}{\Gamma; \cdot \mid A \vdash \mathbf{idle}}$$

$$\textsc{TT-Sess}\ \frac{\Gamma \mid A \mid S \triangleright M : \mathsf{Unit} \triangleleft \mathsf{end}}{\Gamma; s[\mathsf{p}] : S \mid A \vdash (M)^{s[\mathsf{p}]}} \qquad \textsc{TT-NoSess}\ \frac{\Gamma \mid A \mid \mathsf{end} \triangleright M : \mathsf{Unit} \triangleleft \mathsf{end}}{\Gamma; \cdot \mid A \vdash M}$$

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

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

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

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

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

$$\textsc{TI-Empty}\ \frac{}{\Gamma; \cdot \mid A \vdash \epsilon} \qquad \textsc{TI-Callback}\ \frac{\Gamma \mid A \mid S \triangleright M : \mathsf{Unit} \triangleleft \mathsf{end} \qquad \Gamma; \Delta \mid A \vdash \rho}{\Gamma; \Delta, \iota^{+} : S \mid A \vdash \rho[\iota \mapsto M]}$$

Fig. 12. Typing of Configurations

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 [44]: the type environment Δ' consists of a set of session channel endpoints $\{s[\mathsf{p}_i]\}_i$ with session types S_{p_i}, along with a session queue $s : Q$. Environment Δ' must satisfy φ, where φ is at least a safety property.

Rule T-Par types the 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 state has a compatible type. We also require that the collection of roles that make up the access point satisfy a safety property in order to ensure that any established session is safe.

Rule T-Actor types an actor $\langle a, \mathcal{T}, \sigma, \rho, U \rangle$ 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 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. Tying 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 C.

THEOREM 4.2 (PRESERVATION). *Typability is preserved by structural congruence and reduction.*

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

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

4.2.2 Progress. *Progress* states that if all protocols are deadlock-free, then a configuration can either reduce, or it contains no sessions and no further sessions can be established. We start by classifying a *canonical form* for configurations.

Definition 4.3 (Canonical form). A configuration C is in *canonical form* if it can be written:

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

Every well typed configuration can be written in canonical form; the result follows from the structural congruence rules and Theorem 4.2. Next, we define *progress* on runtime environments, which is a safety property on types that ensures all sent messages are eventually received.

Definition 4.4 (Progress). A runtime environment Δ *satisfies progress*, written $\mathrm{prog}(\Delta)$, if $\Delta \Longrightarrow^{*} \Delta' \not\Longrightarrow$ implies that $\Delta' = s : \epsilon$.

If we require the session *types* in every session to satisfy progress, the property transfers to *configurations*: a non-reducing closed configuration cannot be blocked on any session communication and so cannot contain any sessions.

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

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

4.2.3 Global Progress. In the absence of general recursion, the system in fact enjoys *global progress*: every session will be able to reduce after a finite number of steps. The restriction on general recursion aligns with the expectation that message handlers should not run indefinitely and block the event loop. Nevertheless, finite recursive behaviour can be achieved using for example structural recursion [32] or a natural number recursor as in System T (c.f. [19]). Let $\Gamma \vdash^{\mathsf{f}} V : A$, $\Gamma \mid C \mid S \triangleright^{\mathsf{f}} M : A \triangleleft T$, and $\Gamma;\ \Delta \vdash^{\mathsf{f}} C$ be type judgements for finite values, terms, and configurations respectively, where terms cannot contain recursive functions. Given a configuration typing derivation it is sometimes useful to annotate session name restrictions with their associated runtime environments, i.e., $(\nu s : \Delta)C$. The *session progress* theorem shows that for every session, any reduction in its associated session typing environment can be (eventually) reflected by a session reduction.

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

THEOREM 4.7 (SESSION PROGRESS). *If* $\cdot;\ \cdot \vdash^{\mathsf{f}}_{prog} (\nu s : \Delta_s)C$ *where* $\mathit{active}(\Delta_s)$, *then* $C \xrightarrow{\tau}{}^{*} \xrightarrow{s}$.

The proof introduces an LTS for reduction of computations; standard techniques such as $\top\top$-lifting [31] show the existence of a finite reduction sequence to either a value or **suspend** V for some V. Global progress follows as a consequence of an operational correspondence result between the LTS and configurations, along with similar reasoning to that of Theorem 4.5.

Let us write $\mathrm{activeSessions}(C)$ for the set of names of sessions typable under active environments. 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 of Theorem 4.7.

Corollary 4.8 (Global Progress). *If* $\cdot;\cdot \vdash^{\mathsf{f}}_{prog} C$, *then for every* $s \in \mathit{activeSessions}(C)$, $C \equiv (\nu s)\mathcal{D}$ *for some* $\mathcal{D}$, *and* $\mathcal{D} \xrightarrow{\tau}{}^{*} \xrightarrow{s}$.

5 EXTENSIONS

In this section we discuss extending Maty with the ability to switch between sessions (allowing a message in one session to trigger communication in another), and the ability to support failure handling and supervision hierarchies; we concentrate on the latter and give a mostly informal description of session switching, but full details of both extensions can be found in Appendix A.

5.1 Switching Between Sessions

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. 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 $\text{Maty}_{\rightleftarrows}$. 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 {
    getItemInfo(itemID) ↦ [...]
    checkout((itemIDs, details)) ↦
      let items = get in
      if inStock(itemIDs, items) then [...]
      else
        Customer ! outOfStock();
        become Restock itemIDs;
        suspend? custReqHandler
  }

shop(custAP, staffAP, restockAP) ≜
  register custAP Shop
    (registerForever(custAP, Shop, λ_. suspend? itemReqHandler) ());
  register staffAP Shop
    (registerForever(staffAP, Shop, λ_. suspend? staffReqHandler) ());
  register restockAP Shop (suspend! Restock restockHandler)

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

The program is implicitly parameterised by a mapping from static names like $\underline{\text{Restock}}$ to pairs of session types and payload types (in our scenario, $\underline{\text{Restock}}$ 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 $\textbf{suspend}_{?}\ V$ (to suspend awaiting an incoming message, as previously), and $\textbf{suspend}_{!}\ \underline{\text{s}}\ V$ (to suspend session with name $\underline{\text{s}}$ given a function V, until switched into), and introduce the **become** $\underline{\text{s}}\ V$ construct to switch into a suspended session. Specifically, **become** $\underline{\text{s}}\ V$ queues $\underline{\text{s}}$ 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. $\text{Maty}_{\rightleftarrows}$ 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 $\text{Maty}_{\rightleftarrows}$ instead enjoys progress up-to invocation of send-suspended sessions (see Appendix D).

Syntax

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

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

$$\textsc{T-Spawn}\ \frac{\Gamma \mid A \mid \mathsf{end} \triangleright M : \mathsf{Unit} \triangleleft \mathsf{end} \qquad \Gamma \vdash V : A}{\Gamma \mid C \mid S \triangleright \textbf{spawn } M\ V : \mathsf{Pid} \triangleleft S}
\qquad
\textsc{T-Suspend}\ \frac{\Gamma \vdash V : \mathsf{Handler}(S^?, C) \\ \Gamma \mid C \mid \mathsf{end} \triangleright M : \mathsf{Unit} \triangleleft \mathsf{end}}{\Gamma \mid C \mid S^? \triangleright \textbf{suspend } V\ M : A \triangleleft T}$$

$$\textsc{T-Monitor}\ \frac{\Gamma \vdash V : \mathsf{Pid} \\ \Gamma \mid C \mid \mathsf{end} \triangleright M : \mathsf{Unit} \triangleleft \mathsf{end}}{\Gamma \mid C \mid S \triangleright \textbf{monitor } V\ M : \mathsf{Unit} \triangleleft S}
\qquad
\textsc{T-Raise}\ \frac{}{\Gamma \mid C \mid S \triangleright \textbf{raise} : A \triangleleft T}$$

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

$$\begin{array}{ll}
\textsc{E-React} & \langle a, \textbf{idle}, \sigma[s[\mathsf{p}] \mapsto (\textbf{handler } \mathsf{q}\ \{\overrightarrow{H}\}, N)], \rho, U, \omega \rangle \parallel s \triangleright (\mathsf{q}, \mathsf{p}, \ell(V)) \cdot \delta \\
 & \quad \xrightarrow{s} \langle a, (M\{V/x\})^{s[\mathsf{p}]}, \sigma, \rho, U, \omega \rangle \parallel s \triangleright \delta \quad \text{if } (\ell(V) \mapsto M) \in \overrightarrow{H}
\end{array}$$

$$\begin{array}{llcl}
\textsc{E-Spawn} & \langle a, \mathcal{M}[\textbf{spawn } M\ V], \sigma, \rho, U, \omega \rangle & \xrightarrow{\tau} & (\nu b)(\langle a, \mathcal{M}[\textbf{return } b], \sigma, \rho, U, \omega \rangle \parallel \langle b, M, \epsilon, \epsilon, V, \epsilon \rangle) \\
\textsc{E-Suspend} & \langle a, (\mathcal{E}[\textbf{suspend } V\ M])^{s[\mathsf{p}]}, \sigma, \rho, U, \omega \rangle & \xrightarrow{\tau} & \langle a, \textbf{idle}, \sigma[s[\mathsf{p}] \mapsto (V, M)], \rho, U, \omega \rangle \\
\textsc{E-Monitor} & \langle a, \mathcal{M}[\textbf{monitor } b\ M], \sigma, \rho, U, \omega \rangle & \xrightarrow{\tau} & \langle a, \mathcal{M}[\textbf{return } ()], \sigma, \rho, U, \omega \cup \{(b, M)\} \rangle \\
\textsc{E-InvokeM} & \langle a, \textbf{idle}, \sigma, \rho, U, \omega \cup \{(b, M)\} \rangle \parallel \lightning b & \xrightarrow{\tau} & \langle a, M, \sigma, \rho, U, \omega \rangle \parallel \lightning b \\
\textsc{E-Raise} & \langle a, \mathcal{E}[\textbf{raise}], \sigma, \rho, U, \omega \rangle & \xrightarrow{\tau} & \lightning a \parallel \lightning \sigma \parallel \lightning \rho \\
\textsc{E-RaiseS} & \langle a, (\mathcal{E}[\textbf{raise}])^{s[\mathsf{p}]}, \sigma, \rho, U, \omega \rangle & \xrightarrow{\tau} & \lightning a \parallel \lightning s[\mathsf{p}] \parallel \lightning \sigma \parallel \lightning \rho \\
\textsc{E-CancelMsg} & s \triangleright (\mathsf{p}, \mathsf{q}, \ell(V)) \cdot \delta \parallel \lightning s[\mathsf{q}] & \xrightarrow{\tau} & s \triangleright \delta \parallel \lightning s[\mathsf{q}] \\
\textsc{E-CancelAP} & (\nu\iota)(p(\chi[\mathsf{p} \mapsto \widetilde{\iota'} \cup \{\iota\}]) \parallel \lightning \iota) & \xrightarrow{\tau} & p(\chi[\mathsf{p} \mapsto \widetilde{\iota'}])
\end{array}$$

$$\begin{array}{ll}
\textsc{E-CancelH} & \langle a, \textbf{idle}, \sigma[s[\mathsf{p}] \mapsto (\textbf{handler } \mathsf{q}\ \{\overrightarrow{H}\}, M)], \rho, U, \omega \rangle \parallel s \triangleright \delta \parallel \lightning s[\mathsf{q}] \\
 & \quad \xrightarrow{\tau} \langle a, M, \sigma, \rho, U, \omega \rangle \parallel s \triangleright \delta \parallel \lightning s[\mathsf{q}] \parallel \lightning s[\mathsf{p}] \quad \text{if } \mathsf{messages}(\mathsf{q}, \mathsf{p}, \delta) = \emptyset
\end{array}$$

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

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

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

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

5.2 Supervision & Cascading Failure

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* [14, 20, 29, 35]: 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 $\mathsf{Maty}_{\lightning}$. We make actors *addressable*, so **spawn** returns process identifier (PID) of type Pid. The **monitor** $V\ M$ construct installs a callback M to be evaluated should the actor referred to by V crash; and **raise** causes an actor to crash and cancels all the sessions in which it is involved. We also modify the **suspend** construct to take an additional computation M to be run if the sender fails and the message is never sent; a sensible piece of syntactic sugar would be **suspend** $V \triangleq$ **suspend** V **raise** to propagate the failure.

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

$$\mathsf{shopSup}(custAP, staffAP) \triangleq \mathbf{monitor}\ (\mathbf{spawn}\ \mathsf{shop}(custAP, staffAP))\ (\mathsf{shopSup}(custAP, staffAP))$$

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 $\langle a, \mathcal{T}, \sigma, \rho, U, \omega\rangle$, where ω pairs monitored PIDs with callbacks to be evaluated should the actor crash. We also introduce three kinds of "zapper thread", $\lightning\, a$, $\lightning\, s[\mathsf{p}]$, $\lightning\, \iota$ to indicate the cancellation of an actor, role, or initialisation token respectively.

Reduction rules by example. Consider the supervised Shop example (with state stock) 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*:

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

For brevity we shorten Shop, Customer, and PaymentProcessor to S, C, and PP respectively. We also define configuration contexts $\mathcal{G} ::= [\,]\ \mid\ (\nu\alpha)\mathcal{G}\ \mid\ \mathcal{G} \parallel \mathcal{C}$; and concretely let $\mathcal{G} = (\nu sup)(\nu shop)(\nu cust)(\nu pp)(\nu s)([\,] \parallel \langle sup, \mathbf{idle}, \epsilon, \epsilon, (), (shop, \mathsf{shopSup}(cAP, sAP))\rangle)$.

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

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

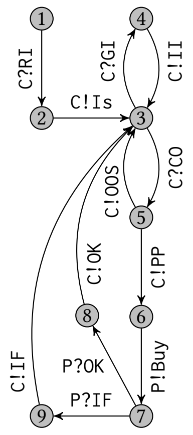
Next, since $s[\mathsf{S}]$ has been cancelled, the checkout message can never be received and so is removed from the queue (E-CancelMsg). Similarly since both C and PP are waiting for messages from cancelled role S, they both evaluate their failure computations, **raise** (E-CancelH). In turn this results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

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

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.

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

Metatheory. All metatheoretical properties continue to hold in the presence of failure (see Appendix D). A modified version of global progress holds: in every active session, a finite number of reductions will either lead to a communication action or result in all endpoints being cancelled.



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. 14. (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. 15. Example handler code from a Maty actor implemented in Scala using the toolchain-generated API

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 [24]:

(1) The user specifies *global types* in the Scribble protocol description language [48].
(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) [5].
(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. 6). Fig. 14 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, which match those in Fig. 5). 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.

Fig. 15 demonstrates a user implementation of an event handler in a Shop actor, for the extended Shop+Restock protocols, using the generated APIs. This code can be compared with Fig. 7 and §5.1. The API guides the user through the protocol to construct a Maty actor with compliant handlers for every possible input event. For example, Fig. 15 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.

For user convenience, our toolchain supports an *inline* version of **become**, as used in Fig. 15. It allows the callback for a session switching behaviour to be performed inline with the currently active handler. 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. The trade-off is this entity must be treated linearly, which our current framework checks dynamically (see below).

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.

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 [46, 49] 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). However, our API design requires *linear* usage of state type objects (e.g., s and succ). Following other works [6, 24, 39, 43, 47], we check linearity in a hybrid fashion: the Done return types in Fig. 14 statically require suspend to be invoked at least once, but our APIs rule out multiple uses *dynamically*. We exploit our formal support for failure handling (Sec. 5.2) 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 will enjoy global progress provided every handler is terminating.

6.2 Evaluation

Table 1 summarises selected examples from the Savina [27] benchmark suite (lower) and larger case studies (upper); Appendix B 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.

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. 7)	✓	✓		✓	✓	✓			✓			
ShopRestock (§ 5.1)	✓	✓		✓	✓	✓			✓		✓	
Robot [13]	✓			✓		✓	✓	(✓)				
Chat [12]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
Ping-self [27]	✓	✓	✓	✓	✓				✓		✓	✓
Ping [27]	✓	✓										
Fib [26]				✓	✓	✓	✓	✓	✓	✓	✓	
Dining-self [27]	✓	✓	✓	✓	✓	✓	✓	(✓)	✓		✓	✓
Dining [27]	✓	✓		✓	✓	✓	✓	(✓)	✓			
Sieve [27]	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	

⊕/& = 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 = **become** self = Self communication

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. We reimplemented a real-world factory use case from Actyx AG [1], originally described by Fowler et al. [13]. In this scenario, multiple Robots access a Warehouse with a single door, with only one Robot 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).

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 (cf. the example registerForever function in Sec. 2).

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

Chat server. This use case [12] 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), noting that Maty's support for event-driven processing of concurrent sessions again allows us to handle the decomposed sessions with distinct roles naturally within a single Client/Room actor (mSA, mRA). We use **become** (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. [49] 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. [33] for session-typed web applications, and by Thiemann [46] in Agda [38]. In contrast, our approach only yields control to the event loop on actor *receives*, as in idiomatic actor programming.

Hu et al. [23] and Kouzapas et al. [28] introduced a binary session calculus with primitives used to implement an event loop; our work instead encodes an event loop directly in the semantics. Viering et al. [47] 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; our global progress is also stronger as it ensures possible progress on *every session in the system*. These works all concentrate on process calculi as opposed to programming language design.

Mostrous and Vasconcelos [34] 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 [16] 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 [37] (later implemented in Erlang [12]), but our language design supports *static* checking and is formalised. Neykova and Yoshida [36] show how causality information in global types enables *protocol-guided recovery*, leading to speedups over naïve Erlang recovery strategies. Their implementation is again dynamically-checked. Harvey et al. [20] introduce EnsembleS, which 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 [9, 13], inspired by earlier work on typestate [8, 40], capture the expected contents of an actor 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.

Scalas et al. [45] introduce a behavioural typing discipline 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 choice). 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

Actor languages are powerful tools for writing reliable distributed applications. 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 implement our approach in a static typing tool for Elixir and to investigate path-dependent types in our implementation.

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] 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.

[5] 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

[6] 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

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

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

[9] 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.

[10] 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.

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

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

[13] 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.

[14] 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.

[15] 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.

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

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

[18] 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.

[19] Robert Harper. 2016. *Practical Foundations for Programming Languages (2nd. Ed.)*. Cambridge University Press.

[20] 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.

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

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

[23] 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

[24] 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.

[25] 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.

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

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

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

[29] 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.

[30] 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.

[31] Sam Lindley and Ian Stark. 2005. Reducibility and TT-Lifting for Computation Types. In *TLCA (Lecture Notes in Computer Science, Vol. 3461)*. Springer, 262–277.

[32] Erik Meijer, Maarten M. Fokkinga, and Ross Paterson. 1991. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *FPCA (Lecture Notes in Computer Science, Vol. 523)*. Springer, 124–144.

[33] 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.

[34] 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.

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

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

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

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

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

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

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

[42] 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.

[43] 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

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

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

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

[47] 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.

[48] 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.

[49] 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

Modified syntax

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

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

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

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

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 $\boxed{\Gamma; \Delta \vdash \mathcal{C}}\ \boxed{\Gamma; \Delta \mid C \vdash \sigma}\ \boxed{\Gamma \vdash \theta}$

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

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

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 $\boxed{\mathcal{C} \longrightarrow \mathcal{D}}$

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

Fig. 16. Maty$_{\rightleftarrows}$: Modified syntax, typing, and reduction rules

A DETAILS OF STATE AND SWITCHING EXTENSIONS

A.1 Session switching

Our extension to allow session switching is shown in Figure 16. 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: $\textbf{suspend}_{?}\ V$ (which, as before, installs a message handler and suspends an actor) and $\textbf{suspend}_{!}\ \underline{s}\ V$, which suspends a session in a *send* state, installing a function taking a payload of the given type. Finally we introduce a $\textbf{become}\ \underline{s}\ V$ construct that queues a request for the event loop to invoke $\underline{s}$ next time the actor is available.

Metatheory. As would be expected, Maty$_{\rightleftarrows}$ satisfies preservation.

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

However, 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, $\text{Maty}_{\rightleftarrows}$ satisfies a weaker version of progress where non-reducing configurations can contain send-suspended sessions.

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

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

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

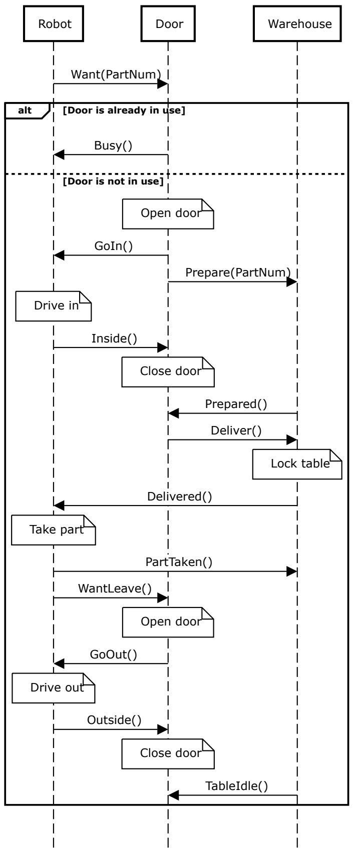
B DETAILS OF CASE STUDY PROTOCOLS

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

B.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;
  }
}
```

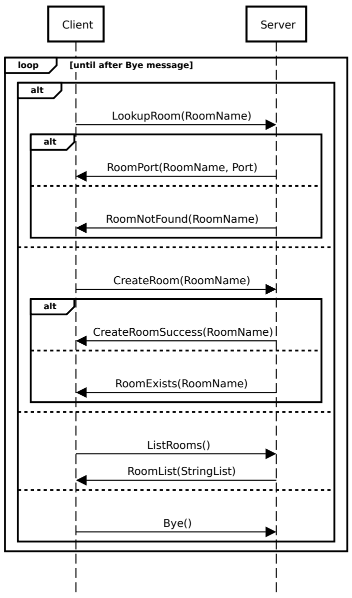


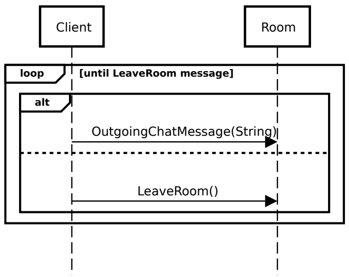
B.2 Chat Server

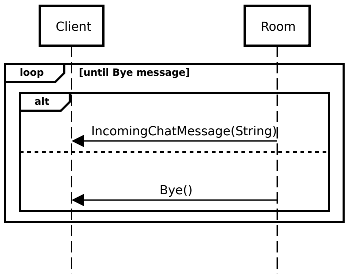
```
global protocol ChatServer(role C, role S) {
  choice at C {
    LookupRoom(RoomName) from C to S;
    choice at S {
      RoomPort(RoomName, Port) from S to C;
    } or {
      RoomNotFound(RoomName) from S to C;
    }
    do ChatServer(C, S);
  } or {
    CreateRoom(RoomName) from C to S;
    choice 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;
  }
}
```







C OMITTED DEFINITIONS AND PROOFS

C.1 Omitted Definitions

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

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

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

C.2 Preservation

We begin with some unsurprising auxiliary lemmas.

Lemma C.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 $\Gamma_1, x : A \mid C \mid S \triangleright M : B \triangleleft T$. □

Lemma C.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* E.

Proof. By induction on the structure of E. □

Lemma C.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* 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 E. □

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

Lemma C.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 $\Gamma; \Delta \vdash \mathcal{C}$ *and* $x \notin dom(\Gamma)$*, then* $\Gamma, x : A \vdash V : B$.

Proof. By mutual induction on all premises. □

Lemma C.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 C.6. *If* $safe(\Delta, \Delta')$*, then* $safe(\Delta)$.

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

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

Proof sketch. 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 C.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 C.9 (Preservation (Equivalence)). *If $\Gamma;\, \Delta \vdash \mathcal{C}$ and $\mathcal{C} \equiv \mathcal{D}$ then there exists some $\Delta' \equiv \Delta$ such that $\Gamma;\, \Delta' \vdash \mathcal{D}$.*

Proof. By induction on the derivation of $\mathcal{C} \equiv \mathcal{D}$. 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 C.10 (Preservation (Configuration reduction)). *If $\Gamma;\, \Delta \vdash \mathcal{C}$ with $safe(\Delta)$ and $\mathcal{C} \longrightarrow \mathcal{D}$, then there exists some Δ' such that $\Delta \Longrightarrow^{?} \Delta'$ and $\Gamma;\, \Delta' \vdash \mathcal{D}$.*

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

Case E-Get.

$$\langle a, \mathcal{M}[\mathbf{get}], \sigma, \rho, V\rangle \xrightarrow{\tau} \langle a, \mathcal{M}[\mathbf{return}\ V], \sigma, \rho, V\rangle$$

There are two subcases based on whether $\mathcal{M} = (\mathcal{E}[-])^{s[\mathsf{p}]}$ or $\mathcal{M} = (\mathcal{E}[-])$. We prove the former; the latter is similar.

Assumption:

$$\dfrac{\dfrac{\Gamma \mid C \mid S \triangleright \mathcal{E}[\mathbf{get}] : \mathsf{Unit} \triangleleft \mathsf{end}}{\Gamma; s[\mathsf{p}] : S \mid C \vdash (\mathcal{E}[\mathbf{get}])^{s[\mathsf{p}]}} \qquad \Gamma; \Delta_1 \mid C \vdash \sigma \qquad \Gamma; \Delta_2 \mid C \vdash \rho \qquad \Gamma \vdash V : C}{\Gamma;\ \Delta_1, \Delta_2, s[\mathsf{p}] : S, a \vdash \langle a, (\mathcal{E}[\mathbf{get}])^{s[\mathsf{p}]}, \sigma, \rho, V\rangle}$$

By Lemma C.2, $\Gamma \mid C \mid S \triangleright \mathbf{get} : C \triangleleft S$, and so by T-Return and Lemma C.3, $\Gamma \mid C \mid S \triangleright \mathcal{E}[\mathbf{return}\ V] : \mathsf{Unit} \triangleleft \mathsf{end}$.

Recomposing:

$$\dfrac{\dfrac{\Gamma \mid C \mid S \triangleright \mathcal{E}[\mathbf{return}\ V] : \mathsf{Unit} \triangleleft \mathsf{end}}{\Gamma; s[\mathsf{p}] : S \mid C \vdash (\mathcal{E}[\mathbf{return}\ V])^{s[\mathsf{p}]}} \qquad \Gamma; \Delta_1 \mid C \vdash \sigma \qquad \Gamma; \Delta_2 \mid C \vdash \rho \qquad \Gamma \vdash V : C}{\Gamma;\ \Delta_1, \Delta_2, s[\mathsf{p}] : S, a \vdash \langle a, (\mathcal{E}[\mathbf{return}\ V])^{s[\mathsf{p}]}, \sigma, \rho, V\rangle}$$

as required.

Case E-Set.

$$\langle a, \mathcal{M}[\mathbf{set}\ W], \sigma, \rho, V\rangle \xrightarrow{\tau} \langle a, \mathcal{M}[\mathbf{return}\ ()], \sigma, \rho, W\rangle$$

Again, we prove the case where $\mathcal{M} = (\mathcal{E}[-])^{s[\mathsf{p}]}$.
Assumption:

$$\dfrac{\dfrac{\Gamma \mid C \mid S \triangleright \mathcal{E}[\mathbf{set}\ W] : \mathsf{Unit} \triangleleft \mathsf{end}}{\Gamma; s[\mathsf{p}] : S \mid C \vdash (\mathcal{E}[\mathbf{set}\ W])^{s[\mathsf{p}]}} \qquad \Gamma;\Delta_1 \mid C \vdash \sigma \qquad \Gamma;\Delta_2 \mid C \vdash \rho \qquad \Gamma \vdash V : C}{\Gamma;\ \Delta_1, \Delta_2, s[\mathsf{p}] : S, a \vdash \langle a, (\mathcal{E}[\mathbf{set}\ W])^{s[\mathsf{p}]}, \sigma, \rho, V\rangle}$$

By Lemma C.2:

$$\dfrac{\Gamma \vdash W : C}{\Gamma \mid C \mid S \triangleright \mathbf{set}\ W : \mathsf{Unit} \triangleleft S}$$

By Lemma C.3 we have $\Gamma \mid C \mid S \triangleright \mathcal{E}[\mathbf{return}\ ()] : \mathsf{Unit} \triangleleft \mathsf{end}$ and so recomposing:

$$\dfrac{\dfrac{\Gamma \mid C \mid S \triangleright \mathcal{E}[\mathbf{return}\ ()] : \mathsf{Unit} \triangleleft \mathsf{end}}{\Gamma; s[\mathsf{p}] : S \mid C \vdash (\mathbf{return}\ ()])^{s[\mathsf{p}]}} \qquad \Gamma;\Delta_1 \mid C \vdash \sigma \qquad \Gamma;\Delta_2 \mid C \vdash \rho \qquad \Gamma \vdash W : C}{\Gamma;\ \Delta_1, \Delta_2, s[\mathsf{p}] : S, a \vdash \langle a, (\mathcal{E}[\mathbf{return}\ ()])^{s[\mathsf{p}]}, \sigma, \rho, W\rangle}$$

Case E-Send.

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

Assumption:

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

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

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

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

Case E-React.

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

For simplicity (and equivalently) let us refer to ℓ as ℓ_j.
Let **D** be the following derivation:

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

Assumption:

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

where $S^? = \mathsf{p}\,\&\{\ell_i(B_i).S_i\}_{i\in I}$.
Since $\mathsf{safe}(\Delta_2, \Delta_3, s[\mathsf{p}] : S^?, s : ((\mathsf{q}, \mathsf{p}, \ell_j(A)) \cdot Q))$ we have that $j \in I$ and $A = B_j$.
Similarly since $\ell_j(x_j) \mapsto M \in \overrightarrow{H}$ we have that $\Gamma, x : B_j \mid C \mid S_j \triangleright M : \mathsf{Unit} \triangleleft \mathsf{end}$.
By Lemma C.1, $\Gamma \mid C \mid S_j \triangleright M\{V/x_j\} : \mathsf{Unit} \triangleleft \mathsf{end}$.
Let **D′** be the following derivation:

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

Recomposing:

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

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

Case E-Suspend.

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

Assumption:

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

By Lemma C.2 we have that:

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

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

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

as required.
Case E-Spawn.

$$\langle a, \mathcal{M}[\textbf{spawn}\ M\ V], \sigma, \rho, U\rangle \longrightarrow \langle a, \mathcal{M}[\textbf{return}\ ()], \sigma, \rho, U\rangle \parallel \langle a, M, \epsilon, \epsilon, V\rangle$$

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

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

By Lemma C.2:

$$\frac{\Gamma \mid A \mid \mathsf{end} \triangleright M : \mathsf{Unit} \triangleleft \mathsf{end} \qquad \Gamma \vdash V : A}{\Gamma \mid C \mid S \triangleright \textbf{spawn}\ M\ V : \mathsf{Unit} \triangleleft S}$$

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

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

as required.
Case E-Reset.

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

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:

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

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

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

as required.
Case E-NewAP.

$$\dfrac{c \text{ fresh}}{\langle a, \mathcal{M}[\textbf{newAP}[(\mathsf{p}_i : S_i)_{i \in I}]], \sigma, \rho, U\rangle \longrightarrow (\nu p)(\langle a, \mathcal{M}[\textbf{return}\ p], \sigma, \rho, U\rangle \parallel p((\mathsf{p}_i \mapsto \epsilon)_{i \in 1..n}))}$$

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

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

By Lemma C.2:

$$\dfrac{\varphi \text{ is a safety property} \qquad \varphi((\mathsf{p}_i : S_i)_{i \in I})}{\Gamma \mid C \mid T \triangleright \textbf{newAP}[(\mathsf{p}_i : S_i)_{i \in I}] : \mathsf{AP}((\mathsf{p}_i : S_i)_{i \in I}) \triangleleft T}$$

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

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma' \mid C \mid T \triangleright \mathcal{E}[\textbf{return}\ c] : \mathsf{Unit} \triangleleft \mathsf{end}}{\Gamma'; s[\mathsf{p}] : T \mid C \vdash (\mathcal{E}[\textbf{return}\ c])^{s[\mathsf{p}]}} \qquad \begin{array}{l}\Gamma'; \Delta_2 \mid C \vdash \sigma \\ \Gamma'; \Delta_3 \mid C \vdash \rho \\ \Gamma' \vdash U : C\end{array}}{\Gamma';\ \Delta_2, \Delta_3, s[\mathsf{p}] : T \vdash \langle a, (\mathcal{E}[\textbf{return}\ c])^{s[\mathsf{p}]}, \sigma, \rho, U\rangle} \qquad \dfrac{\begin{array}{cc} c : \mathsf{AP}((\mathsf{p}_i : S_i)_{i \in 1..n}) \in \Gamma & (\cdot \vdash \epsilon : S_i)_{i \in 1..n} \\ \varphi((\mathsf{p}_i : S_i)_{i \in 1..n}) & \varphi \text{ is a safety property}\end{array}}{\Gamma';\ c : \mathsf{AP} \vdash c((\mathsf{p}_i \mapsto \epsilon)_{i \in 1..n})}}{\Gamma';\ \Delta_2, \Delta_3, s[\mathsf{p}] : T, c : \mathsf{AP} \vdash \langle a, (\mathcal{E}[\textbf{return}\ c])^{s[\mathsf{p}]}, \sigma, \rho, U\rangle \parallel p((\mathsf{p}_i \mapsto \epsilon)_{i \in 1..n})}}{\Gamma;\ \Delta_2, \Delta_3, s[\mathsf{p}] : T \vdash (\nu c)(\langle a, (\mathcal{E}[\textbf{return}\ c])^{s[\mathsf{p}]}, \sigma, \rho, U\rangle \parallel p((\mathsf{p}_i \mapsto \epsilon)_{i \in 1..n}))}$$

as required.
Case E-Register.

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

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

Let **D** be the following derivation:

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

Assumption:

$$\dfrac{\mathbf{D} \qquad \dfrac{\dfrac{\{(\mathsf{p}_i : S_i)_{i\in 1..n}\}\ \Delta_4 \vdash \chi}{\{(\mathsf{p}_i:S_i)_{i\in1..n}\}\ \Delta_4, \overrightarrow{\iota'^{-}_j : S_j} \vdash \chi[\mathsf{p}_j \mapsto \widetilde{\iota'}]} \qquad \begin{array}{l} c : \mathsf{AP}((\mathsf{p}_i:S_i)_{i\in1..n}) \in \Gamma \\ \varphi((\mathsf{p}_i:S_i)_{i\in1..n}) \\ \varphi \text{ is a safety property}\end{array}}{\Gamma;\ \Delta_4, \overrightarrow{\iota'^{-}_j : S_j}, p : \mathsf{AP} \vdash p(\chi[\mathsf{p}_j\mapsto\widetilde{\iota'}])}}{\Gamma;\ \Delta \vdash \langle a, (\mathcal{E}[\textbf{register}\ p\ \mathsf{p}_j\ M])^{s[\mathsf{q}]},\sigma,\rho,U\rangle \parallel p(\chi[\mathsf{p}_j\mapsto\widetilde{\iota'}])}$$

By Lemma C.2:

$$\dfrac{\Gamma\vdash c : \mathsf{AP}((\mathsf{p}_i:S_i)_i) \qquad \Gamma\mid C\mid S_j \triangleright M : \mathsf{Unit}\triangleleft \mathsf{end}}{\Gamma\mid C\mid T\triangleright \textbf{register}\ c\ \mathsf{p}_j\ M : \mathsf{Unit}\triangleleft T}$$

By Lemma C.3, $\Gamma \mid C \mid T \triangleright \mathcal{E}[\textbf{return}\ ()] : \mathsf{Unit} \triangleleft \mathsf{end}$.
Now, let **D′** be the following derivation:

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

Finally, we can recompose:

$$\dfrac{\dfrac{\mathbf{D}\qquad \dfrac{\dfrac{\{(\mathsf{p}_i:S_i)_{i\in1..n}\}\ \Delta_4\vdash\chi}{\{(\mathsf{p}_i:S_i)_{i\in1..n}\}\ \Delta_4,\overrightarrow{\iota'^{-}_j : S_j}, \iota^- : S_j \vdash \chi[\mathsf{p}_j\mapsto\widetilde{\iota'}\cup\{\iota\}]} \qquad \begin{array}{l} c : \mathsf{AP}((\mathsf{p}_i:S_i)_{i\in1..n}) \in \Gamma \\ \varphi((\mathsf{p}_i:S_i)_{i\in1..n}) \\ \varphi \text{ is a safety property}\end{array}}{\Gamma;\ \Delta_4,\overrightarrow{\iota'^{-}_j : S_j},\iota^- : S_j, p:\mathsf{AP}\vdash p(\chi[\mathsf{p}_j\mapsto\widetilde{\iota'}\cup\{\iota\}])}}{\Gamma;\ \Delta,\iota^+ : S_j,\iota^- : S_j\vdash\langle a,(\mathcal{E}[\textbf{return}\ ()])^{s[\mathsf{q}]},\sigma,\rho,U\rangle\parallel p(\chi[\mathsf{p}_j\mapsto\widetilde{\iota'}\cup\{\iota\}])}}{\Gamma;\ \Delta\vdash(\nu\iota)(\langle a,(\mathcal{E}[\textbf{return}\ ()])^{s[\mathsf{q}]},\sigma,\rho,U\rangle\parallel p(\chi[\mathsf{p}_j\mapsto\widetilde{\iota'}\cup\{\iota\}]))}$$

as required.
Case E-Init.

$$\dfrac{s \text{ fresh}}{\begin{array}{c}(\nu\iota_{\mathsf{p}_i})_{i\in1..n}(p((\mathsf{p}_i\mapsto\widetilde{\iota'_{\mathsf{p}_i}}\cup\{\iota_{\mathsf{p}_i}\})_{i\in1..n}) \parallel \langle a_i, \textbf{idle}, \sigma_i, \rho_i[\iota_{\mathsf{p}_i}\mapsto M_i], U_i\rangle_{i\in1..n}) \xrightarrow{\tau} \\ (\nu s)(p((\mathsf{p}_i\mapsto\widetilde{\iota'_{\mathsf{p}_i}})_{i\in1..n}) \parallel s\triangleright\epsilon\parallel\langle a_i,(M_i)^{s[\mathsf{p}_i]},\sigma_i,\rho_i,U_i\rangle_{i\in1..n})\end{array}}$$

For each actor composed in parallel we have:

$$\dfrac{\dfrac{\Gamma \mid C_i \mid S_i \triangleright M_i : \mathsf{Unit} \triangleleft \mathsf{end} \qquad \Gamma; \Delta_{i_3} \mid \rho \vdash}{\Gamma; \Delta_{i_3}, \iota_i^+ : S_i \mid \rho_i[\iota_{\mathsf{p}_i} \mapsto M_i] \vdash} \qquad \begin{array}{l} \Gamma; \cdot \mid C_i \vdash \mathbf{idle} \\ \Gamma; \Delta_{i_2} \mid C_i \vdash \sigma_i \\ \Gamma \vdash U_i : C_i \end{array}}{\Gamma;\ \Delta_{i_2}, \Delta_{i_3}, \iota_i^+ : S_i, a_i \vdash \langle a_i, \mathbf{idle}, \sigma_i, \rho_i, C_i \rangle}$$

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 $\Gamma;\ \Delta_a, \Delta_{tok+} \vdash (\langle a, \mathbf{idle}, \sigma_i, \rho_i[\iota_{\mathsf{p}_i} \mapsto M_i], U_i \rangle)_{i \in 1..n}$
Assumption (given some Δ):

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

Through the access point typing rules we can show that we can remove each ι_{p_i} from the access point: $\Gamma;\ \Delta \vdash p((\mathsf{p}_i \mapsto \widetilde{\iota'_{\mathsf{p}_i}})_{i \in 1..n})$.
Similarly, for each actor composed in parallel we can construct:

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

Let $\Delta_s = s[\mathsf{p}_1] : S_1, \ldots, s[\mathsf{p}_n] : S_n$
Then by repeated use of TC-Par we have that $\Gamma;\ \Delta_a, \Delta_s \vdash \langle a, (M_i)^{s[\mathsf{p}_i]}, \sigma_i, \rho_i, U_i \rangle_{i \in 1..n}$.
Recomposing:

$$\dfrac{\dfrac{\begin{array}{l} \varphi(\Delta_s) \\ \varphi \text{ is a safety property} \\ \Gamma;\ \Delta \vdash p((\mathsf{p}_i \mapsto \widetilde{\iota'_{\mathsf{p}_i}})_{i \in 1..n}) \end{array} \qquad \dfrac{\dfrac{}{\Gamma;\ s : \epsilon \vdash s \triangleright \epsilon} \qquad \Gamma;\ \Delta_a, \Delta_s \vdash (\langle a, (M_i)^{s[\mathsf{p}_i]}, \sigma_i, \rho_i, U_i \rangle)_{i \in 1..n}}{\Gamma;\ \Delta_a, \Delta_s, s : \epsilon \vdash s \triangleright \epsilon \parallel (\langle a, (M_i)^{s[\mathsf{p}_i]}, \sigma_i, \rho_i, U_i \rangle)_{i \in 1..n}}}{\Gamma;\ \Delta, \Delta_a, \Delta_s, s : \epsilon \vdash p((\mathsf{p}_i \mapsto \widetilde{\iota'_{\mathsf{p}_i}})_{i \in 1..n}) \parallel s \triangleright \epsilon \parallel (\langle a, (M_i)^{s[\mathsf{p}_i]}, \sigma_i, \rho_i, U_i \rangle)_{i \in 1..n}}}{\Gamma;\ \Delta, \Delta_a \vdash (\nu s)(p((\mathsf{p}_i \mapsto \widetilde{\iota'_{\mathsf{p}_i}})_{i \in 1..n}) \parallel s \triangleright \epsilon \parallel (\langle a, (M_i)^{s[\mathsf{p}_i]}, \sigma_i, \rho_i, U_i \rangle)_{i \in 1..n})}$$

as required.

Case E-Lift.
Immediate by Lemma C.5.

Case E-Nu.
Immediate by the IH, noting that by the definition of safety, reduction of a safe context results in another safe context.

Case E-Par.
Immediate by the IH and Lemma C.7.

Case E-Struct.
Immediate by the IH and Lemma C.9.

□

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

Proof. Immediate from Lemmas C.9 and C.10. □

C.3 Progress

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

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

Functional reduction satisfies progress.

Lemma C.11 (Term Progress). *If $\Psi \mid S_1 \triangleright M : A \triangleleft S_2$ then either:*

- *$M = \textbf{return}\ V$ 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.*
  - *$M = \textbf{spawn}\ N$ for some N; or*
  - *$M = \mathtt{p} \,!\, m(V)$ for some role $\mathtt{p}$ and message $m(V)$; or*
  - *$M = \textbf{suspend}\ V$ or some V; or*
  - *$M = \textbf{newAP}[(\mathtt{p}_i : T_i)]$ for some collection of participants $(\mathtt{p}_i : T_i)$*
  - *$M = \textbf{register}\ V\ \mathtt{p}$ for some value V and role $\mathtt{p}$*

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 C.12 (Thread Progress). *Let $C = \mathcal{G}[\langle a, \mathcal{T}, \sigma, \rho, V\rangle]$. If $\cdot; \cdot \vdash C$ then either $\mathcal{T} = \textbf{idle}$, or there exist $\mathcal{G}', \mathcal{T}', \sigma', \rho', V'$ such that $C \longrightarrow \mathcal{G}'[\langle a, \mathcal{T}', \sigma', \rho', V'\rangle]$.*

Proof. If $\mathcal{T} = \textbf{idle}$ then the theorem is satisfied, so consider the cases where $\mathcal{T} = M$ or $\mathcal{T} = (M)^{s[\mathtt{p}]}$. By Lemma C.11, 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**$[(\mathtt{p}_i : S_i)_i]$ can reduce by E-NewAP

Next, consider **register** p $\mathtt{p}$ 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 $M = \mathtt{q} \,!\, \ell(V)$. It cannot be the case that $\mathcal{T} = \mathtt{q} \,!\, \ell(V)$ 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 $\mathcal{T} = (\mathtt{q} \,!\, \ell(V))^{s[\mathtt{p}]}$ for some $s, \mathtt{p}$. 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 C.13. *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 $\cdot; \cdot \vdash_{prog} C$, then either there exists some $\mathcal{D}$ such that $C \longrightarrow \mathcal{D}$, or C is structurally congruent to the following canonical form:*

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

Proof. By Proposition C.13 C can be written in canonical form:

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

By repeated applications of Lemma C.12, either the configuration can reduce or all threads are idle:

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

By the linearity of runtime type environments Δ, each role endpoint $s[\mathtt{p}]$ 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:

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

□

C.4 Global Progress

The overview of the global progress proof is as follows:

- We design a labelled transition system semantics for term reduction (Figure 17).
- We argue that our LTS is strongly-normalising up to **suspend** (Proposition C.14).
- We prove an operational correspondence between the LTS reduction and configuration reduction, specifically that reductions in the LTS semantics can drive configuration reduction, and that every configuration reduction affecting an actor term can be reflected by the LTS.
- Finally we can use this result to show that any session can eventually reduce.

C.4.1 LTS Semantics.

Figure 17 shows the labelled transition semantics for our language of computations. Standard β-reductions are reflected as τ-transitions, and communication and concurrency actions reduce in a single step and are accounted for using labelled reductions.

Proposition C.14 (Strong Normalisation (LTS)). *If $\Psi \mid C \mid S \triangleright^{\mathsf{f}} M : A \triangleleft \mathsf{end}$ then there exists some finite reduction sequence such that either:*

- $M \overset{\mathcal{L}_1}{\hookrightarrow} \cdots \overset{\mathcal{L}_n}{\hookrightarrow} V$ *for some* V*; or*
- $M \overset{\mathcal{L}_1}{\hookrightarrow} \cdots \overset{\mathcal{L}_n}{\hookrightarrow} \mathcal{E}[\textbf{\textit{suspend}}\ V]$ *for some* $\mathcal{E}$ *and* V

Proof sketch. Fine-grain call-by-value is strongly normalising; this can be shown using techniques such as $\top\top$-lifting [31], which also extends to exceptions. These results extend to our LTS as all additional constructs reduce immediately. □

Lemma C.15 (Reduction under contexts). *If $\Gamma; \Delta \vdash \mathcal{G}[C]$ and $C \longrightarrow \mathcal{D}$ then there exists some $\mathcal{G}'$ such that $\mathcal{G}[C] \longrightarrow \mathcal{G}'[\mathcal{D}]$.*

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

We also have a special case for straightforward β-reduction:

Additional Syntax

$$\begin{array}{llcl} \text{Labels} & \mathscr{L} & ::= & \tau \mid \mathsf{get}(V) \mid \mathsf{set}(V) \mid \mathsf{spawn}(M) \\ & & \mid & \mathsf{send}(\mathsf{p}, \ell, V) \mid \mathsf{newAP}(a) \mid \mathsf{register}(p, \mathsf{p}, M) \end{array}$$

Labelled Transition Semantics for Computations $\boxed{M \overset{\mathscr{L}}{\hookrightarrow} N}$

LTS-Get
$$\frac{}{\mathbf{get} \overset{\mathsf{get}(V)}{\hookrightarrow} \mathbf{return}\ V}$$

LTS-Set
$$\frac{}{\mathbf{set}\ V \overset{\mathsf{set}(V)}{\hookrightarrow} \mathbf{return}\ ()}$$

LTS-Spawn
$$\frac{}{\mathbf{spawn}\ M \overset{\mathsf{spawn}(M)}{\hookrightarrow} \mathbf{return}\ ()}$$

LTS-Send
$$\frac{}{\mathsf{p} \,!\, \ell(V) \overset{\mathsf{send}(\mathsf{p},\ell,V)}{\hookrightarrow} \mathbf{return}\ ()}$$

LTS-NewAP
$$\frac{}{\mathbf{newAP}[(\mathsf{p}_i : S_i)_i] \overset{\mathsf{newAP}(a)}{\hookrightarrow} \mathbf{return}\ a}$$

LTS-Register
$$\frac{}{\mathbf{register}\ p\ \mathsf{p}\ M \overset{\mathsf{register}(p,\mathsf{p},M)}{\hookrightarrow} \mathbf{return}\ ()}$$

LTS-Beta
$$\frac{M \longrightarrow_{\mathsf{M}} N}{M \overset{\tau}{\hookrightarrow} N}$$

LTS-Lift
$$\frac{M \overset{\mathscr{L}}{\hookrightarrow} N}{\mathcal{E}[M] \overset{\mathscr{L}}{\hookrightarrow} \mathcal{E}[N]}$$

Fig. 17. LTS Semantics for Computations

Lemma C.16 (Term reduction under contexts). *Let $C = \mathcal{G}[\langle a, \mathcal{Q}[\mathcal{E}[M]], \sigma, \rho, U\rangle]$. If $\Gamma; \Delta \vdash C$ and $M \overset{\tau}{\hookrightarrow} N$, then $C \longrightarrow \mathcal{G}[\langle a, \mathcal{Q}[\mathcal{E}[N]], \sigma, \rho, U\rangle]$.*

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

Lemma C.17 (Simulation). *Suppose $\Psi; \cdot \vdash^{\mathsf{f}} C$ where $C = \mathcal{G}[\langle a, \mathcal{Q}[M], \sigma, \rho, U\rangle]$. If $M \overset{\mathscr{L}}{\hookrightarrow} N$, then there exist some $\mathcal{G}'$, σ', ρ', and U' such that $C \longrightarrow \mathcal{G}'[\langle a, \mathcal{Q}[M'], \sigma', \rho', U'\rangle]$.*

Proof. By induction on the derivation of $M \overset{\mathscr{L}}{\hookrightarrow} N$.

Case LTS-Spawn.
Assumption:

$$\frac{}{\mathbf{spawn}\ M \overset{\mathsf{spawn}(M)}{\hookrightarrow} \mathbf{return}\ ()}$$

By Lemma C.15, E-Spawn, and E-Lift,

$$\mathcal{G}[\langle a, \mathcal{Q}[\mathbf{spawn}\ M], \sigma, \rho, U\rangle] \longrightarrow \mathcal{G}'[\langle a, \mathcal{Q}[\mathbf{return}\ ()], \sigma, \rho, U\rangle \parallel \langle b, M, \sigma, \rho\rangle]$$

which we can write as $\mathcal{G}''[\langle a, \mathcal{Q}[\mathbf{return}\ ()], \sigma, \rho, U\rangle]$ as required.

Case LTS-Send.
Assumption:

$$\frac{}{\mathsf{p} \,!\, \ell(V) \overset{\mathsf{send}(\mathsf{p},\ell,V)}{\hookrightarrow} \mathbf{return}\ ()}$$

Since $\Psi; \cdot \vdash^{\mathsf{f}} C$ where $C = \mathcal{G}[\langle a, \mathcal{Q}[\mathsf{p} \,!\, \ell(V)], \sigma, \rho, U\rangle]$, by T-Session and the linearity of runtime environments, there must exist some $\mathcal{G}'$ such that $C \equiv \mathcal{G}'[\langle a, \mathcal{Q}[\mathsf{p} \,!\, \ell(V)], \sigma, \rho, U\rangle] \parallel s \triangleright \delta$ which can reduce by E-Send to $\mathcal{G}'[\langle a, \mathcal{Q}[\mathbf{return}\ ()], \sigma, \rho, U\rangle] \parallel s \triangleright \delta \cdot (\mathsf{p}, \mathsf{q}, \ell(V))$ as required.

Case LTS-NewAP.

Assumption:

$$\frac{}{\mathbf{newAP}[(\mathsf{p}_i : S_i)_i] \overset{\mathsf{newAP}(a)}{\hookrightarrow} \mathbf{return}\ a}$$

We have that $\Psi; \cdot \vdash^{\mathsf{f}} C$ where $C = \mathcal{G}[\langle a, Q[\mathbf{newAP}[(\mathsf{p}_i : S_i)_i]], \sigma, \rho, U\rangle]$; the result follows from reduction by E-NewAP.

Case LTS-Register.
Assumption:

$$\frac{}{\mathbf{register}\ p\ \mathsf{p}\ M \overset{\mathsf{register}(p,\mathsf{p},M)}{\hookrightarrow} \mathbf{return}\ ()}$$

Since $\Psi; \cdot \vdash^{\mathsf{f}} C$ where $C = \mathcal{G}[\langle a, Q[\mathbf{register}\ p\ \mathsf{p}\ M], \sigma, \rho, U\rangle]$, by T-Session and the linearity of runtime environments, there must exist some $\mathcal{G}'$ such that $C \equiv \mathcal{G}'[\langle a, Q[\mathbf{register}\ p\ \mathsf{p}\ M], \sigma, \rho, U\rangle] \parallel p(\chi[\mathsf{p} \mapsto \tilde{\iota}])$ which can reduce by E-Register to $(\nu\iota')(\mathcal{G}'[\langle a, \mathbf{return}\ (), \sigma, \rho[\iota' \mapsto M], U\rangle] \parallel p(\chi[\mathsf{p} \mapsto \tilde{\iota} \cup \{\iota'\}]))$ as required.

Case LTS-Beta.
Immediate by Lemma C.16.

Case LTS-Lift.
Immediate by the IH and E-LiftM. □

Lemma C.18 (Determinism (Term Reduction)). *Suppose $\Psi; \Delta \vdash^{\mathsf{f}} C$ where $C = \mathcal{G}[\langle a, Q[M], \sigma, \rho\rangle]$. If:*

- *$C \longrightarrow \mathcal{G}_1[\langle a, Q[N_1], \sigma_1, \rho_1, U_1\rangle]$, where $M \neq N_1$*
- *$C \longrightarrow \mathcal{G}_2[\langle a, Q[N_2], \sigma_2, \rho_2, U_1\rangle]$, where $M \neq N_2$*

then up to the identities of fresh variables, $\mathcal{G}_1 = \mathcal{G}_2$, and $N_1 = N_2$, and $\sigma_1 = \sigma_2$, and $\rho_1 = \rho_2$, and $U_1 = U_2$.

Proof. Since $M \neq N_1$ and $M \neq N_2$ the overall reduction must be driven by the reduction from M into N_1 or N_2 respectively. The result then follows by inspection on the reduction rules, noting that β-reduction is deterministic, as are the relevant rules E-Get, E-Set, E-Send, E-Spawn, E-NewAP, and E-Register. □

Lemma C.19 (Reflection). *Suppose $\Psi; \Delta \vdash^{\mathsf{f}} C$ where $C = \mathcal{G}[\langle a, Q[M], \sigma, \rho, U\rangle]$.
If $C \longrightarrow \mathcal{G}'[\langle a, Q[N], \sigma', \rho', U\rangle]$ for some $\mathcal{G}'$, N, σ' and ρ' where $M \neq N$, then there exists some $\mathscr{L}$ such that $M \overset{\mathscr{L}}{\hookrightarrow} N$.*

Proof. Since $M \neq N$, by Lemma C.18, the reduction from C must be unique, and will be the one specified by Lemma C.17. □

Proposition C.20 (Operational Correspondence).

Suppose $\Psi; \Delta \vdash^{\mathsf{f}} C$ where $C = \mathcal{G}[\langle a, Q[M], \sigma, \rho, U\rangle]$.

- *If $M \overset{\mathscr{L}}{\hookrightarrow} N$, then there exist some $\mathcal{G}'$, σ', ρ', and U' such that $C \longrightarrow \mathcal{G}'[\langle a, Q[N], \sigma', \rho', U'\rangle]$.*
- *If $C \longrightarrow \mathcal{G}'[\langle a, Q[N], \sigma', \rho', U'\rangle]$ for some $\mathcal{G}'$, N, σ' and ρ' where $M \neq N$, then there exists some $\mathscr{L}$ such that $M \overset{\mathscr{L}}{\hookrightarrow} N$.*

Proof. Follows as a consequence of Lemmas C.17 and C.19. □

Lemma C.21. *If $\cdot; \cdot \vdash (vs : \Delta)C$ and $C \xrightarrow{\tau} \mathcal{D}$, then $\cdot; \cdot \vdash (vs : \Delta)\mathcal{D}$.*

Proof. A straightforward corollary of Theorem 4.2. □

Theorem 4.7 (Session Progress). *If* $\cdot; \cdot \vdash^{\mathsf{f}}_{prog} (\nu s : \Delta_s)C$ *where* $active(\Delta_s)$, *then* $C \xrightarrow{\tau}{}^{*} \xrightarrow{s}$.

Proof. By T-SessionName we have that $\cdot; s[\mathtt{p}_1] : S_1, \ldots, s[\mathtt{p}_n] : S_n \vdash^{\mathsf{f}} C$ and thus by the linearity of Δ_s alongside rule T-Actor we have some set of actors:

$$\{\langle a_i, \mathcal{T}_i, \sigma_i, \rho_i, U_i \rangle\}_{i \in 1..m}$$

such that for each role $\mathtt{p}_j$ for $j \in 1..n$, either:

- there exists some $\mathcal{T}_k$ such that $\mathcal{T}_k = (M)^{s[\mathtt{p}_j]}$ for some M
- $s[\mathtt{p}_j] \in \mathrm{dom}(\sigma_k)$ for some $k \in 1..m$

Consider the subset of actors where $\mathcal{T}_i \neq$ **idle**, i.e., $\mathcal{T}_i = N_i$ or $\mathcal{T}_i = (N_i)^{s'[\mathtt{p}_j]}$ for some N_i. In this case, for each actor, by Proposition C.14 we have that $N_i \overset{\mathscr{L}_1}{\hookrightarrow} \cdots \overset{\mathscr{L}_n}{\hookrightarrow} N_i'$ where either $N_i' =$ **return** $()$, or $N_i' = E[$**suspend** $V]$ for some value V. By Proposition C.20, we can simulate each reduction sequence as a configuration reduction (and moreover, by the reflection direction, each term can *only* follow this reduction sequence). At this point we can revert each actor to idle by either E-Suspend or E-Reset.

If any labelled reduction, simulated as a configuration reduction, is labelled with session s then we can conclude. Otherwise we have that $C \xrightarrow{\tau}{}^{*} \mathcal{D}$ where again by typing we have some subset of actors such that:

$$\{\langle a_i, \textbf{idle}, \sigma_i, \rho_i, U_i \rangle\}_{i \in 1..m'}$$

By Lemma C.21 we have that $\cdot; \cdot \vdash^{\mathsf{f}}_{\mathrm{prog}} (\nu s : \Delta_s)\mathcal{D}$ and thus it remains the case that $\Delta \Longrightarrow$. Thus by similar reasoning to Theorem 4.5 it must be the case that some actor a_j (where $j \in 1..m'$) can reduce by E-React as required. □

D PROOFS FOR SECTION 5

This appendix details the proofs of the metatheoretical properties enjoyed by $\text{Maty}_{\rightleftarrows}$ and $\text{Maty}_{\lightning}$; we omit the proofs for Maty with state, which is entirely standard.

D.1 $\text{Maty}_{\rightleftarrows}$

D.1.1 Preservation.

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

Proof. Preservation of typing under structural congruence follows straightforwardly.

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

Case E-Suspend$_!$-1.
Similar to E-Suspend$_!$-2.

Case E-Suspend$_!$-2.

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

Assumption:

$$\frac{\dfrac{\Gamma \mid C \mid S \triangleright \mathcal{E}[\textbf{suspend}_! \ \underline{s} \ V] : \mathsf{Unit} \triangleleft \mathsf{end}}{\Gamma; s[\mathsf{p}] : S \mid C \vdash (\mathcal{E}[\textbf{suspend}_! \ \underline{s} \ V])^{s[\mathsf{p}]}} \quad \dfrac{\Gamma; \Delta_1 \mid C \vdash \sigma \qquad \Sigma(\underline{s}) = (S^!, A) \qquad (\Gamma \vdash W_i : A \xrightarrow[C]{S^!,\mathsf{end}} \mathsf{Unit})_i}{\Gamma; \Delta_1, (s_i[\mathsf{q}_i] : S^!)_i \mid C \vdash \sigma[\underline{s} \mapsto (s_i[\mathsf{q}_i], W_i)_i]} \quad \begin{array}{l} \Gamma; \Delta_2 \mid C \vdash \rho \\ \Gamma \vdash \Delta_3 C \theta \\ \Gamma \vdash U : C \end{array}}{\Gamma;\ \Delta_1, \Delta_2, \Delta_3, s[\mathsf{p}] : S, (s_i[\mathsf{q}_i] : S^!)_i, a \vdash \langle a, \textbf{idle}, \sigma[\underline{s} \mapsto (s_i[\mathsf{q}_i], W_i)_i], \rho, U, \theta\rangle}$$

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

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

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

Recomposing:

$$\frac{\dfrac{}{\Gamma; \cdot \mid \textbf{idle} \vdash} \quad \dfrac{\begin{array}{c} \Gamma; \Delta_1 \mid C \vdash \sigma \qquad \Sigma(\underline{s}) = (S^!, A) \\ (\Gamma \vdash W_i : A \xrightarrow[C]{S^!,\mathsf{end}} \mathsf{Unit})_i \qquad \Gamma \vdash V : A \xrightarrow[C]{S^!,\mathsf{end}} \mathsf{Unit} \end{array}}{\Gamma; \Delta_1, (s_i[\mathsf{q}_i] : S^!)_i, s[\mathsf{p}] : S^! \mid C \vdash \sigma[\underline{s} \mapsto (s_i[\mathsf{q}_i], W_i)_i \cdot (s[\mathsf{p}], V)]} \quad \begin{array}{l} \Gamma; \Delta_2 \mid C \vdash \rho \\ \Gamma \vdash \theta \\ \Gamma \vdash U : C \end{array}}{\Gamma;\ \Delta_1, \Delta_2, s[\mathsf{p}] : S, (s_i[\mathsf{q}_i] : S^!)_i, a \vdash \langle a, (\mathcal{E}[\textbf{suspend}_! \ \underline{s} \ V])^{s[\mathsf{p}]}, \sigma[\underline{s} \mapsto (s_i[\mathsf{q}_i], W_i)_i \cdot (s[\mathsf{p}], V)], \rho, U, \theta\rangle}$$

as required.

Case E-Become.

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

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

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

By Lemma C.2 we have:

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

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

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

as required.
Case E-Activate.

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

Let **D** be the subderivation:

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

Assumption:

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

Recomposing:

$$\frac{\dfrac{\dfrac{\Gamma \vdash V : A \xrightarrow[C]{S^!, \mathsf{end}} \mathsf{Unit} \qquad \Gamma \vdash W : A}{\Gamma \mid C \mid S^! \triangleright V\ W : \mathsf{Unit} \triangleleft \mathsf{end}}}{\Gamma; s[\mathsf{p}] : S^! \mid C \vdash (V\ W)^{s[\mathsf{p}]}} \qquad \dfrac{\begin{array}{c}\Gamma; \Delta_1 \mid C \vdash \sigma \\ \Sigma(\underline{s}) = (S^!, A) \qquad (\Gamma \vdash V_i : A \xrightarrow[)]{S^!, \mathsf{end}} \mathsf{Unit}_i\end{array}}{\Gamma; \Delta_1, (s_i[\mathsf{p}_i] : S^!)_i \mid C \vdash \sigma, \underline{s} \mapsto (s_i[\mathsf{p}_i], V_i)_i} \qquad \Gamma; \Delta_2 \mid C \vdash \rho \qquad \Gamma \vdash \theta}{\Gamma;\ \Delta_1, \Delta_2, s[\mathsf{p}] : S^!, (s_i[\mathsf{p}_i] : S^!)_i, a \vdash \langle a, (V\ W)^{s[\mathsf{p}]}, \sigma[\underline{s} \mapsto (s_i[\mathsf{p}_i], V_i)_i], \rho, U, \theta \rangle}$$

as required. □

Runtime syntax

Cancellation-aware runtime envs.	Φ	$::=$	$\cdot \mid \Phi, p \mid \Phi, \iota^{\pm} : S \mid \Phi, s[\mathsf{p}] : S \mid \Phi, s[\mathsf{p}] : \lightning \mid \Phi, s : Q$
Labels	γ	$::=$	$\cdots \mid \lightning s[\mathsf{p}] \mid s : \mathsf{p} \lightning \mathsf{q}{::}\ell \mid s : \mathsf{p} \lightning \mathsf{q}$

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

T-ActorName
$$\frac{\Gamma, a : \mathsf{Pid}; \Phi, a \vdash C}{\Gamma; \Phi \vdash (\nu a) C}$$

T-ZapActor
$$\frac{}{\Gamma; a \vdash \lightning a}$$

T-ZapRole
$$\frac{}{\Gamma; s[\mathsf{p}] : \lightning \vdash \lightning s[\mathsf{p}]}$$

T-ZapTok
$$\frac{}{\Gamma; \iota^{+} : S \vdash \lightning \iota}$$

T-Actor
$$\frac{\begin{array}{c} \Gamma; \Phi_1 \mid C \vdash \mathcal{T} \qquad \Gamma; \Phi_2 \mid C \vdash \sigma \qquad \Gamma; \Phi_3 \mid C \vdash \rho \\ \Gamma \vdash U : C \qquad \forall (b, M) \in \omega.\ \Gamma \vdash b : \mathsf{Pid} \wedge \Gamma \mid \mathsf{end} \triangleright M : \mathsf{Unit} \triangleleft \mathsf{end} \end{array}}{\Gamma; \Phi_1, \Phi_2, \Phi_3, a \vdash \langle a, \mathcal{T}, \sigma, \rho, U, \omega \rangle}$$

TH-Handler
$$\frac{\begin{array}{c} \Gamma \vdash V : \mathsf{Handler}(S^{?}, C) \\ \Gamma \mid C \mid \mathsf{end} \triangleright M : \mathsf{Unit} \triangleleft \mathsf{end} \qquad \Gamma; \Phi \mid C \vdash \sigma \end{array}}{\Gamma; \Phi, s[\mathsf{p}] : S^{?} \mid C \vdash \sigma[s[\mathsf{p}] \mapsto (V, M)]}$$

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

Lbl-ZapMsg	$\Phi, s[\mathsf{q}] : \lightning, s : (\mathsf{p}, \mathsf{q}, \ell(A)) \cdot Q$	$\xrightarrow{s:\mathsf{p} \lightning \mathsf{q}::\ell}$	$\Phi, s[\mathsf{q}] : \lightning, s : Q$	
Lbl-ZapRecv	$\Phi, s[\mathsf{p}]{:}\mathsf{q} \,\&\{\ell_i(A_i).S_i\}_{i \in I}, s[\mathsf{q}]{:}\lightning, s{:}Q$	$\xrightarrow{s:\mathsf{p} \lightning \mathsf{q}}$	$\Phi, s[\mathsf{p}]{:}\lightning, s[\mathsf{q}]{:}\lightning, s{:}Q$	(if $\mathsf{messages}(\mathsf{q}, \mathsf{p}, Q) = \emptyset$)
Lbl-Zap	$\Phi, s[\mathsf{p}] : S$	$\overset{s[\mathsf{p}]}{\rightsquigarrow}$	$\Phi, s[\mathsf{p}] : \lightning$	

Fig. 18. Maty$_{\lightning}$: Modified configuration typing rules and type LTS

D.1.2 Progress.

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

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

where for each session s_j there exists some mapping $s_j[\mathsf{p}] \mapsto (\underline{s}, V)$ (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 C.12) holds as before, since we can always evaluate **become** by E-Become, and we can always evaluate $\textbf{suspend}_!$ 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:

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

However, there are now *three* places each role endpoint $s[\mathsf{p}]$ 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. □

D.2 Maty$_{\lightning}$

Figure 18 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 $s[\mathsf{p}] : \lightning$, denoting that endpoint $s[\mathsf{p}]$ has been cancelled. We also need to extend the type LTS to account for failure propagation; we take a similar approach to Barwell et al. [4]. 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.

All metatheoretical results continue to hold.

D.2.1 Preservation. 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 $\text{roles}(\Delta) = \{\mathsf{p} \mid s[\mathsf{p}] : S \in \Phi\}$ to retrieve the roles from an environments.

Let us also define the operation $\text{zap}(\Phi, \widetilde{\mathsf{p}})$ that cancels any role in the given set, i.e., $\text{zap}(s[\mathsf{p}_1] : S_1, s[\mathsf{p}_2] : S_2, a, \{\mathsf{p}_1\}) = s[\mathsf{p}_1] : \lightning, s[\mathsf{p}_2] : S_2, a$.

Lemma D.1. *If $\mathit{safe}(\Phi)$ then $\mathit{safe}(\mathit{zap}(\Phi, \widetilde{\mathsf{p}}))$ for any $\widetilde{\mathsf{p}} \subseteq \mathit{roles}(\Phi)$.*

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

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 D.2 (Preservation ($\longrightarrow$, Maty$_{\lightning}$)). *If $\Gamma;\ \Phi \vdash \mathcal{C}$ with $\mathit{safe}(\Phi)$ and $\mathcal{C} \longrightarrow \mathcal{D}$, then there exists some Φ' such that $\Phi \Rrightarrow \Phi'$ 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 $\mathcal{C} \longrightarrow \mathcal{D}$, concentrating on the new rules rather than the adapted rules (which are straightforward changes to the existing proof).

Case E-Monitor.

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

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

Assumption:

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

where $\forall (b, N) \in \omega.\ \Gamma \vdash b : \mathsf{Pid} \land \Gamma \mid C \mid \mathsf{end} \triangleright N : \mathsf{Unit} \triangleleft \mathsf{end}$.

By Lemma C.2, we know:

$$\dfrac{\Gamma \vdash b : \mathsf{Pid} \qquad \Gamma \mid C \mid \mathsf{end} \triangleright M : \mathsf{Unit} \triangleleft \mathsf{end}}{\Gamma \mid C \mid S \triangleright \mathbf{monitor}\ b\ M : \mathsf{Unit} \triangleleft S}$$

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

Recomposing:

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

noting that $\omega \cup (b, N)$ is safe since $\Gamma \vdash b : \mathsf{Pid}$ and $\Gamma \mid S \triangleright \mathcal{E}[\textbf{return }()] : \mathsf{Unit} \triangleleft \mathsf{end}$, as required.

Case E-InvokeM.

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

Assumption:

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

where $\forall (a', M) \in \omega \cup \{(b, N)\}.\ \Gamma \vdash b : \mathsf{Pid} \ \wedge\ \Gamma \mid \mathsf{end} \triangleright M : \mathsf{Unit} \triangleleft \mathsf{end}$.

Recomposing:

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

as required.

Case E-Raise.

Similar to E-RaiseS.

Case E-RaiseS.

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

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

where $\forall (b, M) \in \omega.\ \Gamma \vdash b : \mathsf{Pid} \ \wedge\ \Gamma \mid \mathsf{end} \triangleright M : \mathsf{Unit} \triangleleft \mathsf{end}$.

Let us write $\lightning\Phi = \{s[\mathsf{p}] : \lightning \mid s[\mathsf{p}] : S \in \Phi\}$. 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:

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

with the environment reduction:

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

as required.

Case E-CancelMsg.

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

Assumption:

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

Recomposing, we have:

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

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:

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

Recomposing:

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

as required.

Case E-CancelH.

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

Let **D** be the following derivation:

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

Assumption:

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

We can recompose as follows. Let $\mathbf{D}'$ be the following derivation:

$$\frac{\dfrac{\Gamma \mid \mathsf{end} \triangleright M : \mathsf{Unit} \triangleleft \mathsf{end}}{\Gamma; \cdot \mid C \vdash M} \qquad \Gamma; \Phi_1 \mid C \vdash \sigma \qquad \Gamma; \Phi_2 \mid C \vdash \rho \qquad \Gamma \vdash U : C}{\Gamma;\ \Phi_1, \Phi_2, a \vdash \langle a, M, \sigma, \rho, U, \omega \rangle}$$

Then we can construct the remaining derivation:

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

Finally, we need to show environment reduction:

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

as required. □

D.2.2 Progress. $\mathsf{Maty}_{\lightning}$ enjoys a similar progress property 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 progress property on environments to account for session failure.

Definition D.3 (Progress (Cancellation-aware environments)). A runtime environment Φ *satisfies progress*, written $\mathsf{prog}_{\lightning}(\Phi)$, if $\Phi \Longrightarrow^{*} \Phi' \not\Longrightarrow$ implies that either $\Phi' = s : \epsilon$ or $\Phi' = (s[\mathsf{p}_i] : \lightning)_i, s : \epsilon$.

We first need to define a canonical form that takes zapper threads into account.

Definition D.4 (Canonical form ($\mathsf{Maty}_{\lightning}$)). A $\mathsf{Maty}_{\lightning}$ configuration $\mathcal{C}$ is in *canonical form* if it can be written:

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

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

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 D.5. *If $\Gamma;\ \Phi \vdash \mathcal{C}$ then there exists a $\mathcal{D} \equiv \mathcal{C}$ 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 D.6. *If $\mathsf{prog}_{\lightning}(\Phi)$ then $\mathsf{prog}_{\lightning}(zap(\Phi, \widetilde{\mathsf{p}}))$ for any $\widetilde{\mathsf{p}} \subseteq roles(\Phi)$.*

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

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

Lemma D.7 (Thread Progress). *Let $\mathcal{C} = \mathcal{G}[\langle a, \mathcal{T}, \sigma, \rho\rangle]$. If $\cdot;\ \cdot \vdash \mathcal{C}$ then either:*

- *$\mathcal{T} =$ **idle**, or*
- *there exist $\mathcal{G}', \mathcal{T}', \sigma', \rho'$ such that $\mathcal{C} \longrightarrow \mathcal{G}'[\langle a, \mathcal{T}', \sigma', \rho'\rangle]$, or*

- $C \longrightarrow \mathcal{G}'[\lightning a \parallel \lightning \sigma \parallel \lightning \rho]$ *if* $\mathcal{T} = \mathcal{E}[\textbf{\textit{raise}}]$, *or*
- $C \longrightarrow \mathcal{G}'[\lightning a \parallel \lightning s[\mathsf{p}] \parallel \lightning \sigma \parallel \lightning \rho]$ *if* $\mathcal{T} = (\mathcal{E}[\textbf{\textit{raise}}])^{s[\mathsf{p}]}$.

Proof. As with Lemma C.12 but taking into account that:

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

□

Theorem D.8 (Progress (Maty$_\lightning$)). *If* $\cdot;\cdot \vdash_{prog_\lightning} C$, *then either there exists some* $\mathcal{D}$ *such that* $C \longrightarrow \mathcal{D}$, *or* C *is structurally congruent to the following canonical form:*

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

Proof. The reasoning is similar to that of Theorem 4.5. By Lemma D.5, C can be written in canonical form:

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

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

By repeated applications of Lemma D.7, either the configuration can reduce or all threads are idle:

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

By the linearity of runtime type environments Δ, each role endpoint $s[\mathsf{p}]$ must either be contained in an actor, or exist as a zapper thread $\lightning s[\mathsf{p}] \in \widetilde{\lightning\alpha}$. 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:

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

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 $(\nu s)(\lightning s[\mathsf{p}_1] \parallel \cdots \parallel \lightning s[\mathsf{p}_n] \parallel s \triangleright \epsilon)$, and we are left with a configuration of the form:

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

as required. □

D.2.3 Global Progress.

Lemma D.9 (Session Progress (Maty$_\lightning$)). *If* $\cdot; \cdot \vdash^{\mathsf{f}}_{prog} (\nu s : \Delta_s)C$*, then there exists some* $\mathcal{D}_1$ *such that* $C \xrightarrow{\tau}{}^{*} \mathcal{D}_1$ *and either* $\mathcal{D}_1 \xrightarrow{s}$*, or* $(\nu s)\mathcal{D}_1 \equiv \mathcal{D}_2$ *for some* $\mathcal{D}_2$ *where* $s \notin activeSessions(\mathcal{D}_2)$*.*

Proof. The proof is as with Theorem 4.7, except we must account for failed sessions arising as a consequence of reduction. □

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 D.10 (Global Progress (Maty$_\lightning$)). *If* $\cdot; \cdot \vdash^{\mathsf{f}}_{prog_\lightning} C$*, then for every* $s \in activeSessions(C)$*, 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* $s \notin activeSessions(\mathcal{D}_2)$*.*

Proof. Arises as a corollary of Lemma D.9. □