Speak Now

Safe Actor Programming with Multiparty Session Types

DRAFT: July 2024

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

Actor programming languages such as Erlang and Elixir are extensively used to implement scalable and reliable distributed applications. Actor languages rely on *explicit message passing*, but since communication patterns are only informally specified, programs written in actor languages are vulnerable to costly errors such as communication mismatches and deadlocks that are only caught after deployment. *Multiparty session types* provide a mechanism for checking that programs satisfy pre-defined protocols, and therefore provide a lightweight way to rule out communication errors early in the development process. However, until now, the nature of actor communication has made it difficult to apply session types to actor languages.

In this paper we describe 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. Our approach not only allows us to guarantee preservation and session fidelity properties, but also the strong property of *global progress*: communication is eventually possible in every session. We extend our language design to support both cascading failure handling and the ability to proactively switch between sessions. We describe a Scala implementation of Maty using an API generation approach, and we demonstrate our implementation by implementing a factory scenario and a chat server.

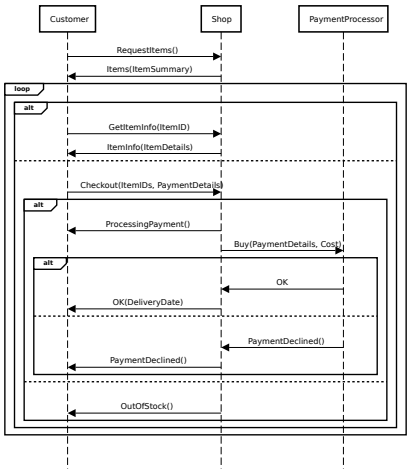
1 INTRODUCTION

Nowadays, writing distributed applications is inescapable, and 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 failure handling, fault tolerance, and conformance to complex communication protocols.

Actor languages such as Erlang and Elixir, as well as frameworks such as Akka, are popular tools for writing scalable and resilient distributed applications. Actor languages are communication-centric programming languages based on message passing (as opposed to shared memory concurrency), and are inspired by the actor model of computation [1, 21]: each actor can respond to an incoming message by spawning other actors, sending messages, or changing how it will respond to future messages. Because communication in actor languages is asynchronous and every message is stored locally to the actor that will process it, actor languages support idioms such as *supervision hierarchies* that allow a failed process to be restarted if it crashes. Erlang in particular has been used to ensure the reliability of real-time systems such as telephone switches [2] and powers the servers of WhatsApp, which has billions of users worldwide.

In spite of these advantages, the communication-centric nature of actor languages is 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: for example, waiting for a message that will never arrive, sending a message that cannot be handled, or sending message at a point that it is not expected. *Multiparty session types* (MPSTs) are a promising type discipline for communication protocols. MPSTs allow a developer to check, at compile time, that they have correctly followed all communication protocols, in turn avoiding costly bugs manifesting themselves after an application has been deployed.

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



(a) Sequence diagram for Shop example

```
handle_cast({checkout, ItemIDs, PaymentDetails,
             ReplyTo}, Items) ->
  AllItemsInStock = check_stock(ItemIDs, Items),
  if
    AllItemsInStock ->
      ReplyTo ! processing_payment,
      case gen_server:call(payment_processor,
          { buy, PaymentDetails,
            cost(ItemIDs, Items) }) of
        ok ->
          NewItems =
            decrease_stock(ItemIDs, Items),
          ReplyTo ! { ok, tomorrow },
          { noreply, NewItems };
        payment_declined ->
          ReplyTo ! payment_declined,
          { noreply, Items }
      end;
    true ->
      ReplyTo ! out_of_stock,
      {noreply, Items}
  end;
```

(b) Handling the `checkout` message

Fig. 1. Protocol and implementation for Shop example

This paper presents the first actor-based programming language design fully supporting statically-checked multiparty session types, thereby allowing developers to benefit from both the error prevention mechanism of session types and the scalability and fault tolerance of actor languages.

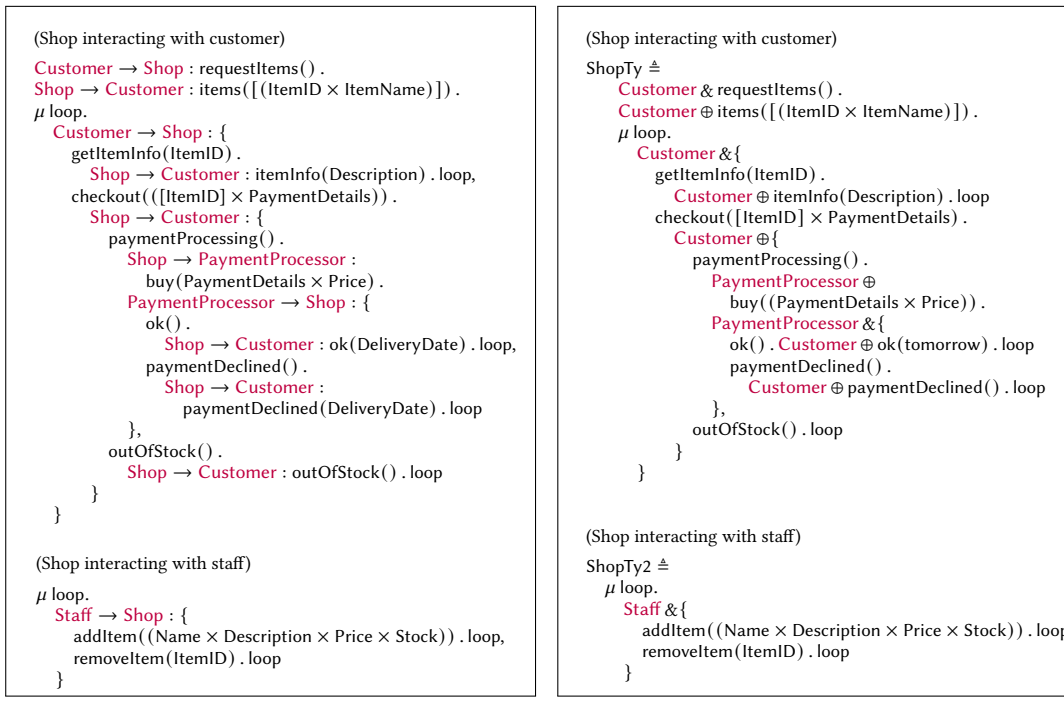
1.1 Motivating Example

Consider the following scenario, depicted in Figure 1a:

- 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** then relays the result to the **Customer**, along with a delivery date if the purchase was successful.
- Separately, **Staff** can also log in and adjust the stock.

Erlang applications often make use of the Erlang/OTP framework [7] that includes pre-defined *behaviours* that provide common behaviour in a uniform way. In particular, the `gen_server` behaviour encapsulates the common use case of a server that can communicate with multiple clients at the same time, while maintaining some state. The `gen_server` behaviour allows the server to handle asynchronous messages (`cast` messages) and also synchronous calls.

In Erlang, we would implement the **Shop** using the `gen_server` behaviour. The `gen_server` stores the shop's current stock as its state. All synchronous calls are handled by the `handle_call` callback



(a) Global types (b) Local types

Fig. 2. Global and local types for shop scenario

that takes three arguments: the message to process, the sender's process ID (which is unused), and the current state (`Items`). We handle the `request_items` and `get_item_info` messages as follows:

```
handle_call(request_items, _, Items) ->
  Summary = item_summary(Items),
  {reply, {items, Summary}, Items};
```

```
handle_call({get_item_info, ID}, _,
                  Items) ->
  Item = lookup_item_info(ID, Items),
  {reply, {item_info, Item}, Items}.
```

In both of these cases, we generate a response to the message (either the summary of all items, or the detailed item description) and respond to the request by returning three-tuple of the `reply` atom (signifying that the event loop should send a message in response); the message to send; and the updated state.

The `checkout`, `add_item`, and `remove_item` messages are handled by the `handle_cast` callback since they are not synchronous calls. Handling the `checkout` message is more involved, as shown in Figure 1b. The server handles the `checkout` message by firstly checking if all items are in stock. If so, then it notifies the customer with a `processing_payment` messages and makes a call to the `payment_processor`. The payment processor replies either with `ok` (indicating that the payment was successful), in which case the stock is decreased; or `payment_declined`. In both cases the result is relayed to the customer. Finally, the `add_item` and `remove_item` messages are handled by updating the shop's state; their message handlers are straightforward and so are omitted.

Even though the scenario is quite small, there is a lot of room for error in the implementation: for example, forgetting the `ReplyTo ! out_of_stock` line would result in the customer waiting indefinitely, whereas a payload or arity mismatch on any of the messages would result in a runtime error.

1.2 Multiparty session types

Structured interactions, like those in our motivating example, can be captured by *multiparty session types* (MPSTs) [22]. A *global type* can be projected to *local types* that describe the interactions from the perspective of each participant, and are used in typechecking. Figure 2a shows the global types for our scenario: the first global type shows the interactions between the customer, shop, and payment processor, whereas the second shows the (simpler) interactions between the staff and the shop. A global type is a sequence of interactions between participants: for example $\mathsf{Shop} \rightarrow \mathsf{Customer} : \mathsf{items}([(\mathsf{ItemID} \times \mathsf{ItemName})])$. G denotes the Shop role sending an items message to the Customer role before proceeding as G, with the message payload set as a list of pairs of item IDs and names. Interactions may have different branches (e.g., getItemInfo and checkout), indicating a choice at the sender.

The corresponding *local types* for the Shop role are shown in Figure 2b. Note that the local types only contains the communication actions that are relevant to the shop. Selection (output) actions are denoted with $\oplus$, whereas offering (receive) actions are denoted with $\&$.

MPSTs are a convenient and successful approach that allow us to statically check conformance to communication protocols. However, there are significant challenges to applying MPSTs to actor-style programming: session types were originally investigated in the context of *communication channels*, whereas actors have a single *mailbox* (see [15] for a detailed comparison). The uni-directional communication model for mailboxes makes it difficult to apply session types directly.

Multiparty session actors. Neykova and Yoshida [38] introduced a programming methodology for actor programming with multiparty session types, which was later applied to Erlang [12]. The idea (Figure 3) is that each actor can be involved in *multiple sessions*, with incoming and outgoing messages passing through FSM-based monitors. However, the dynamic approach detects violations late, and monitoring incurs performance overheads. Furthermore, these works have not been formalised, so there is no formal account of their metatheoretical properties.

There are also significant gaps between existing work on languages with statically-checked MPSTs and the actor paradigm: key to the session actor paradigm is the ability to take part in *multiple sessions*. However, most existing systems offer few guarantees when a process is involved with more than one session: without sophisticated type systems [8] or more restrictive communication topologies that enforce separation between sessions [26] we cannot rule out deadlocks.

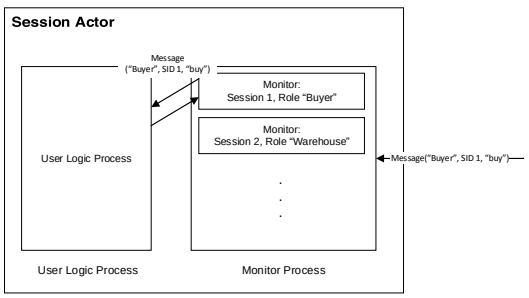


Fig. 3. Multiparty session actors as introduced by Neykova and Yoshida [38] (image from [12])

The majority of session-typed languages and frameworks do not allow a process to listen for messages on multiple sessions, in spite of this being a common pattern in concurrent programming. The notable exceptions are systems that combine session types with event-driven programming (e.g., [50, 53]), but these require a *full* inversion of control, leading to code that can be difficult to follow. These approaches are also formalised as process calculi, leaving a significant gap between the formalism and the concrete programming language design.

Key ideas. The main contribution of this work is the first *statically-typed* language design, Maty, that supports actor programming with multiparty session types that can fully make use of the programming model introduced by Neykova and Yoshida [37] (in contrast to more

limited designs [16, 20] that only allow actors to take part in a single session). Our key insight is to combine a *flow-sensitive* effect type system [32] (similar to work on parameterised monads [3]) with *event-driven programming* and *first-class message handlers*. Actors register to participate in a session through an *access point* [18]. After session establishment, each actor can perform computations and send messages in direct style, and suspend with an message handler when they are waiting for a message from another participant. Our approach allows idiomatic actor-style programming with statically-checked session types, and also scales to extensions such as state, proactively switching between sessions, and gracefully handles failures using Erlang-style supervision hierarchies.

1.3 Maty by example

Maty is a functional programming language with support for lightweight processes that communicate using session-typed message passing. In this section we will show how our Erlang shop actor can be written in Maty; the other components can be written similarly.

The entry point of our program creates two *access points* [18]: one for customer sessions, and one for staff sessions. An access point can be thought of as a "matchmaking service": different participants can register their intention to take part in a session, and the session is established once all participants are available. We spawn customer and payment processor actors (details omitted), and also a shop actor. Each access point is created by specifying the set of roles involved with the session along with their local types; ShopTy is above but we omit the other local types.

```
main ≜
let custAP = newAP  Shop : ShopTy,
                    Customer : CustTy,
                    PaymentProcessor : PPTy
in
let staffAP = newAP  Shop : ShopTy2,
                     Staff : StaffTy
in
spawn shop(custAP, staffAP) initialState;
spawn staff(staffAP) ();
spawn customer(custAP) ()
```

```
registerForever(ap, role, callback) ≜
  rec install(_).
    register ap role (install ());
    callback ()

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

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. In general, by evaluating **register** V p M an actor registers with access point V to take part in the session as role p, storing computation M to be invoked when the session is established. 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 *message handlers* itemReqHandler and staffReqHandler respectively. Suspending places the actor in an idle state and installs the handler to be invoked when a message arrives.

Figure 4 shows the implementation of the shop's message handlers. A message handler has the form **handler** p $\{\ell_i(x_i) \mapsto M_i\}_{i \in I}$, where p is the role from which we are expecting to receive; each $\ell_i(x_i)$ denotes a message tag ℓ_i (for example buy or removeItem) and a variable x_i to bind the message's payload in the computation M_i. For simplicity, we use meta-level recursion as a shorthand for a (mutually)-recursive definition; we assume the usual encoding using anonymous recursive functions. The structure of the program closely mirrors that of the corresponding `gen_server` code. The main differences are:

- Communication takes place using *role names* as opposed to process IDs. This avoids the need to pass PIDs as part of messages.

```
itemReqHandler ≜
  handler Customer {
    requestItems() ↦
      let items = get in
      Customer ! itemSummary(summary(items));
      suspend custReqHandler

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
        PaymentProcessor ! buy((total, details));
        suspend paymentResponseHandler(itemIDs)
      else
        Customer ! outOfStock();
        suspend custReqHandler
  }

paymentResponseHandler(itemIDs) ≜
  handler PaymentProcessor {
    ok() ↦
      let items = get in
      let newItems = decreaseStock(itemIDs, items) in
      set newItems;
      Customer ! ok(deliveryDate(itemIDs));
      suspend custReqHandler
    paymentDeclined() ↦
      Customer ! paymentDeclined();
      suspend custReqHandler
  }

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

Fig. 4. Implementation of Shop message handlers in Maty

- Unlike the `gen_server` code, where code for all messages must be specified in the `handle_call` and `handle_cast` functions, our structured programming model means that each handler only needs to consider messages that are relevant at that point of the session.
- The code makes use of an effectful treatment of state through the **get** and **set** constructs.

Importantly, each actor can be in *multiple sessions at once*, so can handle requests from many clients, with all communication checked statically.

1.4 Contributions

The overarching contribution of this paper is the first statically-typed actor language design with multiparty session types where each actor can be involved in multiple sessions, making essential use of a flow-sensitive effect typing discipline and first-class message handlers. Concretely, we make three specific contributions:

(1) We introduce Maty, the first actor language design with full support for multiparty session types (§2). 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 (§3).
(2) We describe three extensions to Maty: state; the ability to proactively switch to another session; and support for process supervision and cascading failure as in Erlang (§4).
(3) We detail our implementation of Maty using an API generation approach in Scala (§5), and demonstrate our implementation on a real-world case study from the factory domain as well as a chat server application.

Section 6 discusses related work, and Section 7 concludes. We intend to submit our implementation as an artifact.

Syntax of terms

Roles	p, q		
Variables	x, y, z, f		
Values	V, W	::=	$x \mid \lambda x.\, M \mid \mathbf{rec}\, f(x).M \mid c \mid \mathbf{handler}\ \mathsf{p}\ \{\overrightarrow{H}\}$
Message Handlers	H	::=	$\ell(x) \mapsto M$
Computations	M, N	::=	$\mathbf{let}\ x \Leftarrow M\ \mathbf{in}\ N \mid \mathbf{return}\ V \mid V\ W$
		|	$\mathbf{if}\ V\ \mathbf{then}\ M\ \mathbf{else}\ N$
		|	$\mathbf{spawn}\ M \mid \mathsf{p}\,!\,\ell(V) \mid \mathbf{suspend}\ V$
		|	$\mathbf{newAP}_{(\mathsf{p}_i:T_i)_i} \mid \mathbf{register}\ V\ \mathsf{p}\ M$

Syntax of types and type environments

Output session types	$S^!$	::=	$\mathsf{p} \oplus \{\ell_i(A_i).S_i\}_i$
Input session types	$S^?$	::=	$\mathsf{p}\,\&\{\ell_i(A_i).S_i\}_i$
Session types	S, T	::=	$S^! \mid S^? \mid \mu X.S \mid X \mid \mathsf{end}$
Types	A, B	::=	$C \mid A \xrightarrow{S,T} B \mid \mathsf{AP}((\mathsf{p}_i : S_i)_i) \mid \mathsf{Handler}(S^?)$
Base types	C	::=	$\mathbf{1} \mid \mathsf{Bool} \mid \mathsf{Int} \mid \cdots$
Type environments	Γ	::=	$\cdot \mid \Gamma, x : A$

Fig. 5. Syntax of terms and types

2 MATY: AN ACTOR LANGUAGE WITH STATICALLY-CHECKED MULTIPARTY SESSION TYPES

2.1 Syntax

Figure 5 shows the syntax of Maty. We omit state (as in our Shop example) from our core calculus as it is orthogonal, considering it as an extension (§4). We let p, q range over roles, and $x, y, z, f \ldots$ range over variables. It is technically convenient to stratify the calculus into values V, W and computations M, N in the style of *fine-grain call-by-value* [29].

Values. Variables, functions, recursive functions, and constants c are standard; we assume constants include at least the unit value $()$ of type $\mathbf{1}$. A message handler $\mathbf{handler}\ \mathsf{p}\ \{\overrightarrow{H}\}$ specifies the behaviour when a message is received from role p; each clause H is of the form $\ell(x) \mapsto M$, which states that when a message with label ℓ is received, the payload is bound to x in M.

Computations. Computations are potentially side-effecting operations. A let-binding $\mathbf{let}\ x \Leftarrow M\ \mathbf{in}\ N$ evaluates M, binding its result to x in N; note that this is the only kind of evaluation context in the system. A return expression $\mathbf{return}\ V$ is a trivial computation returning value V. Function application $V\ W$ and conditionals $\mathbf{if}\ V\ \mathbf{then}\ M\ \mathbf{else}\ N$ are standard.

The $\mathbf{spawn}\ M$ term spawns a new actor that evaluates term M. An actor sends a message with label ℓ and payload V to role p using the term $\mathsf{p}\,!\,\ell(V)$. There is *no* ***receive*** *construct*, since receiving messages is handled by the event loop. Instead, when an actor wishes to receive a message, it must *suspend* itself and install a message handler using $\mathbf{suspend}\ V$.

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 $\mathbf{newAP}_{(\mathsf{p}_i:S_i)_i}$. An actor can *register* to take part in a session as role p on access point V using $\mathbf{register}\ V\ \mathsf{p}\ M$; term M is a callback to be invoked once the session is established.

Session types. Although global types are a convenient way of representing protocols, they are not necessary for our formalism. Instead, we follow Scalas and Yoshida [46] and base our metatheoretical properties around local types. *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. It is technically convenient to refer to let $S^!$ range over selection (or *output*) session types, and $S^?$ to range over branching (or *input*) session types. Session type $\mu X.S$ indicates a recursive

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

$$\textsc{TV-Var}\ \frac{}{\Gamma \vdash x : A}\ x : A \in \Gamma \qquad \textsc{TV-Lam}\ \frac{\Gamma, x : A \mid S \triangleright M : B \triangleleft T}{\Gamma \vdash \lambda x.M : A \xrightarrow{S,T} B} \qquad \textsc{TV-Rec}\ \frac{\Gamma, x : A, f : A \xrightarrow{S,T} B \mid S \triangleright M : A \xrightarrow{S,T} B \triangleleft T}{\Gamma \vdash \mathbf{rec}\, f(x).M : A \xrightarrow{S,T} B} \qquad \textsc{TV-Const}\ \frac{c \text{ has base type } C}{\Gamma \vdash c : C}$$

$$\textsc{TV-Handler}\ \frac{(\Gamma, x : A_i \mid S_i \triangleright M_i : \mathbf{1} \triangleleft \mathsf{end})_i}{\Gamma \vdash \mathbf{handler}\ \mathrm{p}\ \{\ell_i(x_i) \mapsto M_i\}_i : \mathsf{Handler}(\mathrm{p}\,\&\{\ell_i(A_i).S_i\}_i)}$$

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

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

$$\textsc{T-If}\ \frac{\Gamma \vdash V : \mathsf{Bool} \qquad \Gamma \mid S_1 \triangleright M : A \triangleleft S_2 \qquad \Gamma \mid S_1 \triangleright N : A \triangleleft S_2}{\Gamma \mid S_1 \triangleright \mathbf{if}\ V\ \mathbf{then}\ M\ \mathbf{else}\ N : A \triangleleft S_2} \qquad \textsc{T-Spawn}\ \frac{\Gamma \mid \mathsf{end} \triangleright M : \mathbf{1} \triangleleft \mathsf{end}}{\Gamma \mid S \triangleright \mathbf{spawn}\ M : \mathbf{1} \triangleleft S}$$

$$\textsc{T-Send}\ \frac{j \in I \qquad \Gamma \vdash V : A_j}{\Gamma \mid \mathrm{p} \oplus \{\ell_i(A_i).S_i\}_{i \in I} \triangleright \mathrm{p}\,!\,\ell_j(V) : \mathbf{1} \triangleleft S_j} \qquad \textsc{T-Suspend}\ \frac{\Gamma \vdash V : \mathsf{Handler}(S^?)}{\Gamma \mid S^? \triangleright \mathbf{suspend}\ V : A \triangleleft S'}$$

$$\textsc{T-NewAP}\ \frac{\varphi \text{ is a safety property} \qquad \varphi((\mathrm{p}_i : T_i)_{i \in I})}{\Gamma \mid S \triangleright \mathbf{newAP}_{(\mathrm{p}_i : T_i)_{i \in I}} : \mathsf{AP}((\mathrm{p}_i : T_i)_{i \in I}) \triangleleft S} \qquad \textsc{T-Register}\ \frac{j \in I \qquad \Gamma \vdash V : \mathsf{AP}((\mathrm{p}_i : T_i)_{i \in I}) \qquad \Gamma \mid T_j \triangleright M : \mathbf{1} \triangleleft \mathsf{end}}{\Gamma \mid S \triangleright \mathbf{register}\ V\ \mathrm{p}_j\ M : \mathbf{1} \triangleleft S}$$

Fig. 6. Typing rules

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.

Remark 2.1 (Global types). We can nevertheless straightforwardly define global types as follows:

$$\text{Global types} \quad G \quad ::= \quad \mathrm{p} \to \mathrm{q}\{\ell_i(A_i)\,.\,G_i\}_{i \in I} \;\mid\; \mu X.G \;\mid\; X \;\mid\; \mathsf{end}$$

Here, $\mathrm{p} \to \mathrm{q}\{\ell_i(A_i)\,.\,G_i\}_{i \in I}$ states that role p can send a message with label ℓ_j and payload A_i to role q (assuming $j \in I$), with the remainder of the protocol described by G_j.

Projection of global types onto roles is standard [22, 44]; the set of local types resulting from a global type satisfy the safety and progress properties that we will see in §3 [46].

Types. Base types C are standard. Since our type system enforces session typing by pre- and post-conditions (c.f. parameterised monads [3]), a function type $A \xrightarrow{S,T} B$ 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. An access point has type $\mathsf{AP}((\mathrm{p}_i : S_i)_i)$, mapping each role to a local type. Finally, a message handler has type $\mathsf{Handler}(S^?)$ where $S^?$ is an *input* session type.

2.2 Typing rules

Figure 6 shows the typing rules for Maty; there are different judgements for values and computations. Unlike many session type systems, we do not need linearity when typing values or computations as session typing is enforced by effect typing; our approach is inspired by that of Harvey et al. [20].

Value typing. The value typing judgement has the form $\Gamma \vdash V : A$. Typing rules for variables and constants are standard, and typing rules for anonymous functions and anonymous recursive functions are adapted to take session pre- and post-conditions into account. Rule TV-Handler states that a handler **handler** p $\{\ell_i(x_i) \mapsto M_i\}_i$ is typable with type Handler(p $\&\{\ell_i(A_i)\,.\,S_i\}_i$) 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.

Computation typing. The computation typing judgement has the form $\Gamma \mid S \triangleright M : A \triangleleft T$ and can be read, "under type environment Γ and given session precondition S, term M has type A and session postcondition T". Rule T-Return embeds a value in a computation and thus does not affect session typing, and rule T-Let sequences two computations. 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 post-condition 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.

Rule T-Spawn types **spawn** M with the unit type; the spawned thread M must have return type **1** and pre- and post-conditions end (since the spawned computation is not yet in a session and so cannot communicate). Rule T-Send types a send computation 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. The T-Suspend rule states that **suspend** V is typable if the handler is compatible with the current session type precondition; since the computation does not return, it can be given an arbitrary return type and postcondition.

Rule T-NewAp types the **newAP** construct, which must annotated with the set of roles and local types to be involved in the session. The rule ensures that the session types satisfy a *safety property*; we will describe this further in §3, 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.

Finally, rule T-Register types the **register** V p M construct: 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.

2.3 Operational semantics

With the syntax and typing in place, we can now discuss the operational semantics. Figure 7 introduces runtime syntax, structural congruence, and term reduction rules.

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

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

Runtime syntax

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

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

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

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

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

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

$$\begin{array}{rcl} \mathbf{let}\ x \Leftarrow \mathbf{return}\ V\ \mathbf{in}\ M & \longrightarrow_{\mathrm{M}} & M\{V/x\} \\ (\lambda x.M)\ V & \longrightarrow_{\mathrm{M}} & M\{V/x\} \\ (\mathbf{rec}\ f(x).M)\ V & \longrightarrow_{\mathrm{M}} & M\{\mathbf{rec}\ f(x).M/f, V/x\} \end{array}$$

$$\begin{array}{rcl} \mathbf{if\ true\ then}\ M\ \mathbf{else}\ N & \longrightarrow_{\mathrm{M}} & M \\ \mathbf{if\ false\ then}\ M\ \mathbf{else}\ N & \longrightarrow_{\mathrm{M}} & N \\ \mathcal{E}[M] & \longrightarrow_{\mathrm{M}} & \mathcal{E}[N] \quad (\text{if } M \longrightarrow_{\mathrm{M}} N) \end{array}$$

Fig. 7. Operational semantics (1)

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

An access point $p(\chi)$ has name p and state χ, where the state maps roles to 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 $(\mathrm{p}, \mathrm{q}, \ell(V))$ denoting a message $\ell(V)$ sent from p to q.

Structural congruence and term reduction. Structural congruence is the smallest congruence relation defined by the axioms in Figure 7. As with the π-calculus, parallel composition is associative and commutative, and we have the usual scope extrusion rule; we write $\mathsf{fn}(\mathcal{C})$ to refer to the set of free names in a configuration $\mathcal{C}$. 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 is standard β-reduction.

Communication and concurrency. Figure 8 gives the reduction rules on configurations; 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 $s[\mathrm{p}]$, 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 $s[\mathrm{p}]$, rule E-Suspend evaluates **suspend** V by installing handler V for $s[\mathrm{p}]$ and returning the actor to the **idle** state. Analogously to raising an exception, **suspend** discards the current evaluation context.

Reduction labels

$$\text{Reduction labels} \quad l \quad ::= \quad s \mid \tau \qquad\qquad l - \alpha = \begin{cases} \tau & \text{if } l = \alpha \\ l & \text{otherwise} \end{cases}$$

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

E-Send

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

E-React

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

E-Suspend

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

E-Spawn

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

E-Reset

$$\frac{}{\begin{array}{c}\langle a, Q[\textbf{return}\ ()], \sigma, \rho\rangle \xrightarrow{\tau} \\ \langle a, \textbf{idle}, \sigma, \rho\rangle\end{array}}$$

E-NewAP

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

E-Register

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

E-Init

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

E-Lift

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

E-Nu

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

E-Par

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

E-Struct

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

Fig. 8. Operational semantics (2)

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 fresh access point name p and an access point with empty queues for each role. Rule E-Register allows an actor to evaluate **register** p p M in order to register with access point p to take part as role p: the rule creates an initialisation token ι, storing a mapping from ι to the callback M in the local initialisation environment, and appending ι to the potential participant set for p in p. Finally, E-Init establishes a session when idle participants are registered for all roles: in this case, the initialisation tokens are discarded; a session name restriction and empty queue is created; and each initialisation callback is invoked in the context of the newly-created session.

The remaining rules are administrative: E-LiftM allows β-reduction under an evaluation context; E-Nu allows reduction under a name restriction, hiding bound names; E-Par allows reduction under parallel composition; and E-Struct allows reduction modulo structural congruence.

3 METATHEORY

In this section we prove type soundness and progress for Maty; our event-based setting also allows us to show global progress. Following Scalas and Yoshida [46] we begin by showing a type semantics

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	::=	$s : \mathsf{p} \uparrow \mathsf{q} :: \ell \mid s : \mathsf{p} \downarrow \mathsf{q} :: \ell \mid \mathsf{end}(s, \mathsf{p})$

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

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

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

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

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

Fig. 9. Labelled transition system on runtime type environments

for sets of local types, and show how this LTS can be used to specify type-level properties that can be used to prove our metatheoretical results.

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 in the typing of configurations, and are also used 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 $s[\mathsf{p}] : S$; and finally session queue types $s : Q$. Queue types mirror the structure of queue entries and are a triple $(\mathsf{p}, \mathsf{q}, \ell(A))$. We include structural congruence on queue types to match structural congruence on queues, and extend this to runtime environments.

Labelled transition system on environments. Figure 9 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 preservation: it ensures that communication does not introduce type errors (for example, exchanging values of incompatible payload types). Intuitively a safety property on runtime environments 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 3.1 (Safety property). φ is a *safety property* of runtime type environments Δ if:

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

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

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

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

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

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

$$\text{T-Actor}\ \frac{\Gamma;\Delta_1 \vdash \mathcal{T} \qquad \Gamma;\Delta_2 \vdash \sigma \qquad \Gamma;\Delta_3 \vdash \rho}{\Gamma;\, \Delta_1, \Delta_2, \Delta_3, a \vdash \langle a, \mathcal{T}, \sigma, \rho \rangle} \qquad \text{T-EmptyQueue}\ \frac{}{\Gamma;\, s : \epsilon \vdash s \triangleright \epsilon}$$

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

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

$$\text{TA-Empty}\ \frac{}{\{(\mathsf{p}_i : S_i)_i\}\ \cdot \vdash S}$$

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

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

$$\text{TT-Idle}\ \frac{}{\Gamma;\, \cdot \vdash \mathbf{idle}} \qquad \text{TT-Sess}\ \frac{\Gamma \mid S \triangleright M : \mathbf{1} \triangleleft \mathsf{end}}{\Gamma;\, s[\mathsf{p}] : S \vdash (M)^{s[\mathsf{p}]}}$$

$$\text{TT-NoSess}\ \frac{\Gamma \mid \mathsf{end} \triangleright M : \mathbf{1} \triangleleft \mathsf{end}}{\Gamma;\, \cdot \vdash M}$$

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

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

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

$$\text{TI-Empty}\ \frac{}{\Gamma;\, \cdot \vdash \epsilon} \qquad \text{TI-Callback}\ \frac{\Gamma \mid S \triangleright M : \mathbf{1} \triangleleft \mathsf{end} \qquad \Gamma;\Delta \vdash \rho}{\Gamma;\Delta, \iota^+ : S \vdash \rho[\iota \mapsto M]}$$

Fig. 10. Runtime typing

(3) $\varphi(\Delta)$ and $\Delta \Longrightarrow \Delta'$ implies $\varphi(\Delta')$.

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

We henceforth assume that all other properties are safety properties.

3.1 Runtime typing

In order to reason about the metatheory we must firstly define an extrinsic [43] type system for configurations; note that this is used only for reasoning and need not be implemented as part of a typechecker. Figure 10 shows the runtime typing rules for the system.

Runtime typing rules. The runtime typing judgement $\Gamma;\, \Delta \vdash C$ can be read, "under type environment Γ and runtime type environment Δ, configuration C is well typed". The typing rules are also implicitly parameterised by a safety property on sets of local types φ; we omit this from the rules to avoid clutter, but sometimes write $\Gamma;\, \Delta \vdash_\varphi C$ in statements of metatheoretical properties where we want to be specific about the property φ rather than considering the largest safety property.

We have three rules for name restrictions: read bottom-up, T-APName types an access point name restriction by adding p to both the type environment (such that it can be referenced in terms)

and to the runtime environment (to ensure the access point must appear in the configuration). Rule T-InitName types an initialisation token by adding initialisation tokens of both polarities to the runtime type environment. The typing rule for session name restrictions is key to the generalised multiparty session typing approach introduced by Scalas and Yoshida [46]: the type environment Δ' consists of a set of session channel endpoints $\{s[\mathrm{p}_i]\}_i$ with session types S_{p_i}, along with a session queue $s : Q$. Environment Δ' must be safe.

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 $\{(\mathrm{p}_i : S_i)_i\}\ \Delta \vdash \chi$ 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 $\langle a, \mathcal{T}, \sigma, \rho\rangle$ and makes use of three auxiliary judgements. The thread state typing judgement $\Gamma; \Delta \vdash \mathcal{T}$ states that the **idle** state is always well typed (TT-Idle); a thread $(M)^{s[\mathrm{p}]}$ is well typed if given a singleton runtime environment $s[\mathrm{p}] : S$, term M is typable with session precondition S, return type $\mathbf{1}$, and post condition end; and a non-session thread M is typable if it has session pre- and postconditions end and return type $\mathbf{1}$. In turn this ensures that all session actions are used, or the thread suspends. The handler typing judgement ensures that the stored handlers match the types in the runtime environments, and the initialisation state typing judgement 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.

3.2 Properties

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

3.2.1 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 3.2 (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* $C \longrightarrow \mathcal{D}$, *then there exists some* Δ' *such that* $\Delta \Longrightarrow^{?} \Delta'$ *and* $\Gamma; \Delta' \vdash \mathcal{D}$.

3.2.2 Progress. The next major property we can show is *progress*, which states that if a configuration is typable using a property that guarantees session type reduction, then the configuration can either make progress, or is in a position where no actors are involved in a session and no further sessions can be established. We start by classifying a *canonical form* for configurations.

Definition 3.3 (Canonical form). A configuration C is in *canonical form* if it can be written:

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

Every well typed configuration can be written in canonical form; the result follows from the structural congruence rules and Theorem 3.2.

Proposition 3.4 (Canonical forms). *If* $\Gamma; \Delta \vdash C$ *then there exists a* $\mathcal{D} \equiv C$ *where* $\Gamma; \Delta \vdash \mathcal{D}$ *and* $\mathcal{D}$ *is in canonical form.*

Next, we define a progress property on runtime environments.

Definition 3.5 (Progress). A runtime environment Δ *satisfies progress*, written $\mathrm{prog}(\Delta)$, if $\Delta \Longrightarrow^{*} \Delta' \not\Longrightarrow$ implies that $\Delta' = \cdot$.

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. It helps to define configuration contexts $\mathcal{G} ::= [\,] \mid (\nu\alpha)\mathcal{G} \mid \mathcal{G} \parallel C$.

Lemma 3.6 (Thread Progress). *Let $C = \mathcal{G}[\langle a, \mathcal{T}, \sigma, \rho\rangle]$. If $\cdot;\cdot \vdash C$ then either $\mathcal{T} = \mathbf{idle}$, or there exist $\mathcal{G}', \mathcal{T}', \sigma', \rho'$ such that $C \longrightarrow \mathcal{G}'[\langle a, \mathcal{T}', \sigma', \rho'\rangle]$.*

We can finally show that if we require all sets of session types to satisfy a progress property, a non-reducing closed configuration cannot be blocked on any session communication. The proof relies the observation that the runtime typing rules require that each session endpoint must be contained in precisely one actor either to be used by the active thread or suspended as a handler. By Lemma 3.6 we need only consider the case where all actors are idle; the progress property on runtime environments ensures that each actor could reduce by E-React. Thus, a non-reducing configuration cannot contain any sessions.

Theorem 3.7 (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:*

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

3.2.3 Global Progress. The progress theorem shows that session typing can rule out deadlocks. However, we can show a stronger result: in the absence of general recursion, the system enjoys *global progress*: any 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 [33] or a natural number recursor as in System T (c.f. [19]).

Let $\Gamma \vdash^{\mathsf{f}} V : A$, $\Gamma \mid S \rhd^{\mathsf{f}} M : A \triangleleft T$, 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 key *session progress* theorem shows that for every session, any reduction in its associated session typing environment can be (eventually) reflected by a session reduction in the configuration.

Definition 3.8 (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 $s[\mathrm{p}] : S$ where $S \neq \mathsf{end}$. Note that we can always use garbage collection congruences to eliminate inactive sessions.

Theorem 3.9 (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 proceeds by describing a labelled transition for term reduction; standard techniques such as $\top\top$-lifting [31] show this is strongly normalising and thus that there exists a finite reduction sequence until the term reduces to either a value or **suspend** V for some handler V. Global progress then follows as a consequence of showing an operational correspondence between the term LTS and configurations, along with similar reasoning to that of Theorem 3.7.

We define the set of *active sessions* as the set of names of sessions typable under active environments, more precisely the homomorphic extension of the following operation over configurations:

$$\mathrm{activeSessions}((\nu s : \Delta)C) = \begin{cases} \{s\} \cup \mathrm{activeSessions}(C) & \text{if } \mathrm{active}(\Delta) \\ \mathrm{activeSessions}(C) & \text{otherwise} \end{cases}$$

Since (by Theorem 3.2) 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.

```
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})
```

Fig. 11. Shop example with session switching

Corollary 3.10 (Global Progress). *If* $\cdot; \cdot \vdash^{\mathsf{f}}_{prog} C$, *then for every* $s \in activeSessions(C)$, $C \equiv (\nu s)\mathcal{D}$ *for some* $\mathcal{D}$, *and* $\mathcal{D} \xrightarrow{\tau}{}^{*} \xrightarrow{s}$.

4 EXTENSIONS

In this section we show to extend Maty with three extensions. We first show a straightforward extension to allow actor-level state to be shared between sessions. Next, we show how to allow actors to proactively switch between sessions, and illustrate it on the warehouse example shown by Neykova and Yoshida [38]. Finally we show how our approach can be integrated with Erlang-style supervision hierarchies along with cascading session failures [14, 36].

4.1 State

We can straightforwardly extend Maty to support state that can be shared between different sessions (e.g., to record stock levels in a shop, or a transaction log). We give an overview here but give the full system in Appendix A. The key change is to modify the function type to $A_1 \xrightarrow[B]{S,T} A_2$, which can be read as, "A function from A_1 to A_2, with session pre-condition S, post-condition T, working with a state of type B", and record the type of the state in the computation typing judgement: $\Gamma \mid B \mid S \triangleright M : A \triangleleft T$. We can then introduce **get** and **set** V constructs to manipulate the state; we also need to adjust the **spawn** construct to take an initial state:

$$\text{T-Get}\ \frac{}{\Gamma \mid A \mid S \triangleright \mathbf{get} : A \triangleleft S} \qquad \text{T-Set}\ \frac{\Gamma \vdash V : A}{\Gamma \mid A \mid S \triangleright \mathbf{set}\ V : \mathbf{1} \triangleleft S} \qquad \text{T-Spawn}\ \frac{\Gamma \mid B \mid \mathsf{end} \triangleright M : \mathbf{1} \triangleleft \mathsf{end} \qquad \Gamma \vdash V : B}{\Gamma \mid A \mid S \triangleright \mathbf{spawn}\ M\ V : \mathbf{1} \triangleleft S}$$

The main change to the semantics is to extend the actor configuration to $\langle a, \mathcal{T}, \sigma, \rho, V\rangle$ in order to record the current state, which can then be used to implement the E-Get and E-Set rules:

$$\text{E-Get} \quad \langle a, \mathcal{M}[\mathbf{get}], \sigma, \rho, V\rangle \xrightarrow{\tau} \langle a, \mathcal{M}[\mathbf{return}\ V], \sigma, \rho, V\rangle$$
$$\text{E-Set} \quad \langle a, \mathcal{M}[\mathbf{set}\ W], \sigma, \rho, V\rangle \xrightarrow{\tau} \langle a, \mathcal{M}[\mathbf{return}\ ()], \sigma, \rho, W\rangle$$

All metatheoretical properties (preservation, progress, and global progress) continue to hold.

4.2 Switching Between Sessions

Until now we have considered scenarios where an actor is involved in *independent* sessions. In this section we show how to extend Maty so that it is possible for communication in one session

Modified syntax

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

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

$$\textsc{T-Suspend}_? \ \frac{\Gamma \vdash V : \mathrm{Handler}(S^?)}{\Gamma \mid S^? \triangleright \mathbf{suspend}_? \ V : A \triangleleft T} \qquad \textsc{T-Suspend}_! \ \frac{\Sigma(\underline{s}) = (S^!, A) \qquad \Gamma \vdash V : A \xrightarrow{S^!, \mathrm{end}} \mathbf{1}}{\Gamma \mid S^! \triangleright \mathbf{suspend}_! \ \underline{s} \ V : B \triangleleft T}$$

$$\textsc{T-Become} \ \frac{\Sigma(\underline{s}) = (T, A) \qquad \Gamma \vdash V : A}{\Gamma \mid S \triangleright \mathbf{become} \ \underline{s} \ V : \mathbf{1} \triangleleft S}$$

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

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

$$\textsc{TH-SendHandler} \ \frac{\Gamma; \Delta \vdash \sigma \qquad \Sigma(\underline{s}) = (S^!, A) \qquad (\Gamma \vdash V_i : A \xrightarrow{S^!, \mathrm{end}} \mathbf{1})_i}{\Gamma; \Delta, (s_i[\mathrm{p}_i] : S^!)_i \vdash \sigma, \underline{s} \mapsto (s_i[\mathrm{p}_i], V_i)_i}$$

$$\textsc{TR-Empty} \ \frac{}{\Gamma \vdash \epsilon} \qquad \textsc{TR-Request} \ \frac{\Gamma \vdash \theta \qquad \Sigma(\underline{s}) = (S^!, A) \qquad \Gamma \vdash V : A}{\Gamma \vdash \theta \cdot (\underline{s}, V)}$$

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

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

Fig. 12. Maty-⇄: Modified syntax, typing, and reduction rules

to be triggered by another session. We call this extension Maty-⇄. Neykova and Yoshida [38] demonstrate their session actors model using a warehouse scenario. Their scenario is similar to our shop example from Section 1, but allows the shop to maintain a long-running session with a supplier and request delivery of an item if it runs out of stock. We can describe the Restock session with the following simple local types:

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

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

The key difference to our original example is that the sessions are *no longer independent*: the order message to the supplier (in the Restock session) is triggered only *as a consequence* of a buy message in the session with a customer. 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.

Figure 11 shows the extension of the shop example with the ability to switch into the restocking session; the new constructs are shaded. We set $\Sigma = \underline{\text{Restock}} \mapsto (\text{ShopRestock}, [\text{ItemID}])$ to show that we can suspend as, and switch into the Restock session. 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.

Our extension to allow session switching is shown in Figure 12. 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: $\mathbf{suspend}_? V$ (which, as before, installs a message handler and suspends an actor) and $\mathbf{suspend}_! \underline{s} V$, which suspends a session in a *send* state, installing a function taking a payload of the given type. Finally we introduce a $\mathbf{become} \underline{s} V$ 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 4.1 (Preservation). *Preservation (as defined in Theorem 3.2) 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 thread progress continues to hold as before, we obtain a weaker version of progress where non-reducing configurations can contain send-suspended sessions.

Theorem 4.2 (Weak 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:*

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

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

4.3 Supervision & Cascading Failure

A large reason for the success of actor languages is their support for the *let-it-crash* philosophy: if an actor encounters an error then it should crash and be restarted by a *supervisor* actor. Until now we have not considered the possibility of failure. If an actor has crashed, then it cannot send any further messages, so we need some mechanism to ensure sessions do not get 'stuck' due to a failure. Our solution is based on the *affine sessions* approach [36], in particular its adaptation to the multiparty setting [20, 28] and the asynchronous formulation introduced by Fowler et al. [14]. The key idea behind this approach is that a role can be marked as *cancelled*, meaning that it cannot take part in any future sessions. Trying to receive from a cancelled participant, when there are no remaining messages from that participant in the session queue, raises an exception; exceptions then cause an actor to crash and propagate the failure.

Figure 13 shows the additional syntax, typing rules, and reduction rules needed for supervision and cascading failure; we call this extension Maty-$\lightning$. Concretely we make actors addressable, meaning that **spawn** will return a process identifier (PID) of type Pid. We introduce two additional constructs: **monitor** V M monitors the actor referred to by PID V and installs a callback M to be evaluated should the actor crash; and **raise**, which when evaluated 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 $\mathbf{suspend}\ V \triangleq \mathbf{suspend}\ V\ \mathbf{raise}$ to propagate the failure.

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

$$\begin{array}{l}\mathsf{shopSupervisor}(\mathit{custAP}, \mathit{staffAP}) \triangleq \\ \quad \mathbf{let}\ \mathit{shopPid} = \mathbf{spawn}\ \mathsf{shop}(\mathit{custAP}, \mathit{staffAP})\ \mathbf{in} \\ \quad \mathbf{monitor}\ \mathit{shopPid}\ (\mathsf{shopSupervisor}(\mathit{custAP}, \mathit{staffAP}))\end{array}$$

Here, the shopSupervisor spawns the shop actor and monitors the resulting PID. Whenever the spawned shop actor encounters an exception, it will propagate the cancellation to all sessions in

Syntax

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

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

$$\textsc{T-Spawn}\ \frac{\Gamma \mid \mathsf{end} \triangleright M : \mathbf{1} \triangleleft \mathsf{end}}{\Gamma \mid S \triangleright \mathbf{spawn}\ M : \mathrm{Pid} \triangleleft S} \qquad \textsc{T-Suspend}\ \frac{\Gamma \vdash V : \mathrm{Handler}(S^?) \quad \Gamma \mid \mathsf{end} \triangleright M : \mathbf{1} \triangleleft \mathsf{end}}{\Gamma \mid S^? \triangleright \mathbf{suspend}\ V\ M : A \triangleleft T}$$

$$\textsc{T-Monitor}\ \frac{\Gamma \vdash V : \mathrm{Pid} \quad \Gamma \mid \mathsf{end} \triangleright M : \mathbf{1} \triangleleft \mathsf{end}}{\Gamma \mid S \triangleright \mathbf{monitor}\ V\ M : \mathbf{1} \triangleleft S} \qquad \textsc{T-Raise}\ \frac{}{\Gamma \mid S \triangleright \mathbf{raise} : A \triangleleft T}$$

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

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

$$\begin{array}{lrcl}
\textsc{E-Spawn} & \langle a, \mathcal{M}[\mathbf{spawn}\ M], \sigma, \rho, \omega\rangle & \xrightarrow{\tau} & (\nu b)(\langle a, \mathcal{M}[\mathbf{return}\ b], \sigma, \rho, \omega\rangle \parallel \langle b, M, \epsilon, \epsilon, \epsilon\rangle) \\
\textsc{E-Suspend} & \langle a, (\mathcal{E}[\mathbf{suspend}\ V\ M])^{s[\mathrm{p}]}, \sigma, \rho, \omega\rangle & \xrightarrow{\tau} & \langle a, \mathbf{idle}, \sigma[s[\mathrm{p}] \mapsto (V, M)], \rho, \omega\rangle \\
\textsc{E-Monitor} & \langle a, \mathcal{M}[\mathbf{monitor}\ b\ M], \sigma, \rho, \omega\rangle & \xrightarrow{\tau} & \langle a, \mathcal{M}[\mathbf{return}\ ()], \sigma, \rho, \omega \cup \{(b, M)\}\rangle \\
\textsc{E-InvokeM} & \langle a, \mathbf{idle}, \sigma, \rho, \omega \cup \{(b, M)\}\rangle \parallel \lightning b & \xrightarrow{\tau} & \langle a, M, \sigma, \rho, \omega\rangle \parallel \lightning b \\
\textsc{E-Raise} & \langle a, \mathcal{E}[\mathbf{raise}], \sigma, \rho, \omega\rangle & \xrightarrow{\tau} & \lightning a \parallel \lightning \sigma \parallel \lightning \rho \\
\textsc{E-RaiseS} & \langle a, (\mathcal{E}[\mathbf{raise}])^{s[\mathrm{p}]}, \sigma, \rho, \omega\rangle & \xrightarrow{\tau} & \lightning a \parallel \lightning s[\mathrm{p}] \parallel \lightning \sigma \parallel \lightning \rho \\
\textsc{E-CancelMsg} & s \triangleright (\mathrm{p}, \mathrm{q}, \ell(V)) \cdot \delta \parallel \lightning s[\mathrm{q}] & \xrightarrow{\tau} & s \triangleright \delta \parallel \lightning s[\mathrm{q}] \\
\textsc{E-CancelAP} & (\nu\iota)(p(\chi[\mathrm{p} \mapsto \widetilde{\iota'} \cup \{\iota\}]) \parallel \lightning \iota) & \xrightarrow{\tau} & p(\chi[\mathrm{p} \mapsto \widetilde{\iota'}]) \\
\textsc{E-CancelH} & \multicolumn{3}{l}{\langle a, \mathbf{idle}, \sigma[s[\mathrm{p}] \mapsto (V, M)], \rho, \omega\rangle \parallel s \triangleright \delta \parallel \lightning s[\mathrm{q}] \xrightarrow{\tau} \langle a, M, \sigma, \rho, \omega\rangle \parallel s \triangleright \delta \parallel \lightning s[\mathrm{q}] \parallel \lightning s[\mathrm{p}] \quad \text{if } \mathsf{messages}(\mathrm{q}, \mathrm{p}, \delta) = \emptyset}
\end{array}$$

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

$$(\nu s)(\lightning s[\mathrm{p}_1] \parallel \cdots \parallel \lightning s[\mathrm{p}_n] \parallel s \triangleright \epsilon) \parallel \mathcal{C} \equiv \mathcal{C} \qquad (\nu a)(\lightning a) \parallel \mathcal{C} \equiv \mathcal{C}$$

Syntactic sugar

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

Fig. 13. Maty-$\lightning$: Modified syntax and reduction rules

which it is still involved; the failure will then be detected by the shopSupervisor which will restart the actor and monitor it again. The restarted shop actor will then re-register with the access points and can then take part in subsequent sessions.

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

Reduction rules. The first three reduction rules are straightforward adaptations of the base rules: E-React discards the failure callback once a message is received; E-Spawn returns the PID of a spawned actor; and E-Suspend stores the failure handling computation along with the handler.

The next two rules handle process monitoring: E-Monitor stores a failure handler for actor b in ω, whereas E-InvokeM invokes the failure handler once b has failed (i.e., if there is a configuration $\lightning b$). Rules E-Raise and E-RaiseS signify exceptions. In both cases the rules cancel the actor name,

Runtime syntax

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

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

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

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

$$\textsc{T-Actor}\ \frac{\Gamma; \Phi_1 \vdash \mathcal{T} \quad \Gamma; \Phi_2 \vdash \sigma \quad \Gamma; \Phi_3 \vdash \rho \quad \forall (b, M) \in \omega.\ \Gamma \vdash b : \mathsf{Pid} \wedge \Gamma \mid \mathsf{end} \triangleright M : \mathbf{1} \triangleleft \mathsf{end}}{\Gamma; \Phi_1, \Phi_2, \Phi_3, a \vdash \langle a, \mathcal{T}, \sigma, \rho, \omega \rangle}$$

$$\textsc{TH-Handler}\ \frac{\Gamma \vdash V : \mathsf{Handler}(S^{?}) \quad \Gamma \mid \mathsf{end} \triangleright M : \mathbf{1} \triangleleft \mathsf{end} \quad \Gamma; \Phi \vdash \sigma}{\Gamma; \Phi, s[\mathsf{p}] : S^{?} \vdash \sigma[s[\mathsf{p}] \mapsto (V, M)]}$$

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

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

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

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

Fig. 14. Maty-$\lightning$: Modified configuration typing rules and type LTS

any roles currently suspended in the handler environment σ, as well as any pending initialisation tokens. In the case of E-RaiseS, the currently-active session name is also cancelled.

The final three rules describe the action of zapper threads on the remainder of the system. Rule E-CancelMsg discards a message from a queue when the receiver has been cancelled (and can therefore never receive the message). Rule E-CancelAP removes a cancelled initialisation token from an access point (as well as garbage collecting the associated name restriction). Finally, rule E-CancelMsg handles the case where an actor is waiting on a message that will never arrive. We define $\mathsf{messages}(\mathsf{p}, \mathsf{q}, \delta)$ to be the set of messages in δ that are sent from p to q; more specifically $\mathsf{messages}(\mathsf{p}, \mathsf{q}, \delta) = \{\ell(V) \mid (\mathsf{r}, \mathsf{s}, \ell(V)) \in \delta \wedge \mathsf{p} = \mathsf{r} \wedge \mathsf{q} = \mathsf{s}\}$; we extend the definition to queue types (i.e., $\mathsf{messages}(\mathsf{p}, \mathsf{q}, \mathrm{Q})$) similarly. In this case, if an actor playing role p in session s is waiting for a message from q and there are no further messages to process, then the failure continuation is executed and the failure propagates to $s[\mathsf{p}]$.

The additional structural congruences handle 'garbage collection': the first eliminates sessions where all participants have been cancelled, and the second eliminates a cancelled actor that has no references in the remainder of the system.

Metatheory. Figure 14 shows the necessary modifications to the configuration typing rules and type LTS. We extend runtime type environments to *cancellation-aware* environments Φ that include an additional entry of the form $s[\mathsf{p}] : \lightning$, denoting that endpoint $s[\mathsf{p}]$ has been cancelled. Read bottom-up, rule T-ActorName is modified to add an actor name into the term type environment. Rules T-ZapActor, T-ZapRole, and T-ZapTok type zapper threads. Rule T-Actor is modified to ensure that all stored monitor callbacks do not perform session communication, and TH-Handler is modified to store the cancellation callback should receiving a message fail.

We also need to extend the type LTS to account for the possibility of failure; we take a similar approach to Barwell et al. [4]. Rules Lbl-ZapMsg and Lbl-ZapRecv mirror the effects of E-CancelMsg and E-CancelH at the type level. 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 we

include this functionality as a separate relation since (unlike the other two rules) it is unnecessary for determining behavioural properties of types.

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

Theorem 4.3 (Preservation ($\longrightarrow$, Maty-$\lightning$)). *If* $\Gamma;\ \Phi \vdash C$ *with* $safe(\Phi)$ *and* $C \longrightarrow \mathcal{D}$, *then there exists some* Φ' *such that* $\Phi \Rrightarrow \Phi'$ *and* $\Gamma;\ \Phi' \vdash \mathcal{D}$.

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

Theorem 4.5 (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:*

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

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 4.6 (Global Progress (Maty-$\lightning$)). *If* $\cdot;\ \cdot \vdash^{\mathrm{f}}_{prog_{\lightning}} C$, *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* $s \notin activeSessions(\mathcal{D}_2)$.

5 IMPLEMENTATION AND CASE STUDIES

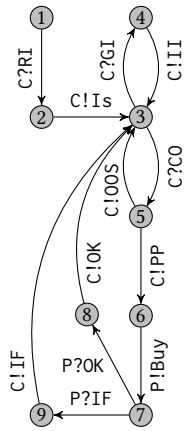
5.1 Implementation

We have implemented a practical toolchain for Maty-style event-driven actor programming in Scala. Our toolchain adopts the API generation approach of Scribble [24, 25]:

(1) The user specifies global types in the Scribble protocol description language [52].
(2) Our toolchain internally uses Scribble to *validate* a global type according to the MPST-based safety conditions, *project* it to local types for each role, and generate a *communicating finite state machine* (CFSM) [5] from each local type.
(3) From each CFSM, the toolchain generates a typed API for the user to implement that protocol role as an event-driven Maty actor in native Scala.

To illustrate, consider the Shop role in the main protocol of our running example (Fig. 2b). Fig. 15 depicts the CFSM for Shop and summarises the main types and operations in the generated Scala API; we abbreviate the CFSM labels for brevity. Fig. 16 gives a user implementation of one handler in the Shop actor using the generated API. It also uses **become** to implement the extended version of the example with session switching (Sec. 4.2); this code can be compared with Fig. 4 and Fig. 11.

The toolchain generates one or more Scala types for each state in the CFSM. In the API summary in Fig. 15, we colour the non-blocking such *state types* blue (where the user directly performs a send or suspend action), and the blocking state types (input events) red. Non-blocking state types offer methods for performing the I/O and handler suspend actions with typing constraints corresponding



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	`sendCustomerItems(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	`sendCustomerItemInfo(pay): S3Suspend`
5	S5	`sendCustomerProcessingPayment(): S6`
		`sendCustomerOutOfStock(): S3Suspend`
6	S6	`sendPaymentProcessorBuy(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	`sendCustomerOK(pay): S3Suspend`
9	S9	`sendCustomerInsufficientFunds(pay): S3Suspend`

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

```
def custReqHandler[T: S1orS3](d: DataS, s: T): Done.type = { // d represents actor state
    s match {
        case c: S1 => c match {
            case RequestItems(sid, pay, succ) => // pay is message payload; succ is successor state
                succ.sendCustomerItems(d.summary()).suspend(d, custReqHandler[S3]) }
        case c: S3 => c match {
            case GetItemInfo(sid, pay, succ) =>
                succ.sendCustomerItemInfo(d.lookupItem(pay)).suspend(d, custReqHandler[S3])
            case Checkout(sid, pay, succ) =>
                if (d.inStock(pay)) {
                    succ.sendCustomerProcessingPayment()
                        .sendPaymentProcessorBuy(d.total(pay))
                        .suspend(d, paymentResponseHandler)
                } else {
                    val sus = succ.sendCustomerOutOfStock()
                    d.staff match { // d.staff: LOption[R1] -- a "frozen" instance of state type R1
                        case x: LSome[R1] => ibecome(d, x, restockHndlr) // R1 is Restock state type
                        case _: LNone => throw new RuntimeException // error handling
                    }
                    sus.suspend(d, custReqHandler[S3])
                }
}   }   }
```

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

to that state in the CFSM (and thus the local type). The return type of these methods corresponds to the successor state type, allowing session actions to be chained. Input state types are traits that are implemented by case classes generated for each input message option. The API calls the user handler with an instance of the appropriate case class when an input event occurs; the case class object carries an instance of the successor state type. For brevity, Fig. 15 omits the type annotations for the session identifier `sid` and message payload value `pay` parameters (they are as in Fig. 1a). The special `Done.type` is a type for which the user can only obtain a value (to return from the handler) by performing a `suspend` or completing a protocol.

Starting from a handler for a new session initiation, the user is guided through the protocol by the typed API to construct a Maty actor with compliant handlers for every session event. Fig. 16

takes state `S1` after initiation, where the shop receives `RequestItems` from the customer. It further handles state `S3`, where the shop receives either `GetItemInfo` or `Checkout`. `S1orS3` is user-defined Scala trait that uses implicits to serve as a union type of `S1` and `S3`, akin to the notion of branch subtyping [17]. We summarise the relationship with our formalism as follows:

- We have designed the generated APIs to avoid full inversion of control [49, 53]): our implementation supports direct-style communication by exposing state types and their (output) operations to the user via *explicit* objects (e.g., `s` and `succ`).
- A related design point is that of *linearity*: the basic usage contract of our API design is that every instance of a state type should be used linearly, i.e., by performing *exactly one* session operation. Following other works [6, 24, 40, 45, 50], our APIs currently enforce linearity *dynamically*: using a state object *more than once* yields an error, and the API asserts that every provided state object is used *at least once* when a handler returns. That said, the API does offer some static assistance; e.g., the `Done` type requires `suspend` to be invoked at least once. API designs with a *fully static* treatment of linearity are possible with various trade-offs; we hypothesise for example that features such as path-dependent types could strengthen the static guarantees.
- For user convenience, our toolchain supports an *inline* version of **become**, as demonstrated above. 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" state type instances as a type `LOption[S]` and resume them later by an inline `ibecome`. The trade-off is this entity must also be treated linearly, which our current framework checks dynamically.

Altogether, our toolchain ensures safe execution of Maty actors, handling multiple heterogeneously-typed sessions concurrently. Note that dynamic linearity errors are subsumed by our treatment of failures. The runtime of our toolchain 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.

5.2 Case Studies

We have used our toolchain to implement a number of use cases to test the expressiveness of Maty and its programming model for event-driven multiparty session actors. This includes our shop running example, which as discussed subsumes the Warehouse use case of Neykova and Yoshida [38]. We highlight some points from other use cases below.

Robot coordination. We have reimplemented the interaction protocol of a real-world use case from Actyx AG[1] [13], who develop control software for factories. As described by Fowler et al. [13], "The use case captures a scenario where multiple robots on a factory floor acquire parts from a warehouse that provides access through a single door. Robots negotiate with the door to gain entry into the warehouse and obtain the part they require." We give a sequence diagram for the interactions between the Robot, Door and Warehouse actors in Appendix B.

Each Robot actor establishes a separate, concurrent multiparty session with the Door and Warehouse actor. A key point is that the event-driven actor model of Maty enables the implementation of the Door and Warehouse each as a *single* actor that can safely handle the concurrent interleavings of events across *any number* of separate Robot sessions. By contrast, the standard programming model of multiparty session π-calculi (e.g., [8, 22]) can only support such use cases by effectively spawning a parallel instance of the Door process for each Robot client (which introduces the considerable further complexity of synchronising the state of these Door processes).

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,

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

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. 1.3).

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

Chat server. This use case [12] involves an arbitrary number of Clients using a Registry to create new chat Rooms and lookup existing ones, and to dynamically join and leave any chat Room. We implement this scenario by modelling each Client, the Registry and each Room as separate actors. We enact Room creation by dynamically spawning new Room actors (with fresh access points), and allow any Client to establish sessions with the Registry or any Room asynchronously from all other Clients. See Appendix B for the use case sequence diagram.

Our current toolchain uses a core version of Scribble that does not support the notion of *subsessions* [11, 12] nor the `par` construct for parallel composition within a global type. Instead we decompose the Client-Registry and the Client-Room interactions into separate sessions. We note, however, that Maty's support for event-driven processing of concurrent sessions again allows us to handle the decomposed sessions in a natural manner within a single Client/Room actor. Moreover, the use case does not involve any interaction between the Registry and Rooms (beyond Room spawning) nor any synchronisation between the parallel global types (beyond overall actor termination), so our implementation remains close to the original.

A key point is the use of **become** in the Room actor to switch between Client sessions in order to broadcast chat messages to all Clients currently in that Room.

6 RELATED WORK

Session-typed functional programming. Session types were first investigated as part of a functional language design by Gay and Vasconcelos [18]. Wadler [51] later introduced a core calculus, *GV*, and described a translation from GV into a session-typed process calculus *CP* whose types are precisely the propositions of classical linear logic. Lindley and Morris [30] provided GV with a semantics and showed semantics-preserving translations between CP and GV in both directions.

Our flow-sensitive effect type system has similarities to Atkey's *parameterised monads* [3], which were used in earlier implementations (e.g. [42]). This form of effect system is particularly useful in typing actor-style programming, since unlike channel-based communication we use session types to govern the actions that a process performs, rather than the type of a channel endpoint.

Event-driven session types. Several works have investigated how inversion of control can be used in conjunction with session typing. Zhou et al. [53] introduce a multiparty type discipline that supports statically-checked *refinements* on communicated data, implemented in F⋆; to avoid needing to reason about linearity, users implement callbacks for each send and receive action in a protocol. A similar approach is used by Miu et al. [34] for session-typed web applications, and by Thiemann [49] in Agda [39]. In contrast, our approach only requires that the program yields control to the event loop when an actor needs to receive (as is done in idiomatic actor frameworks such as Akka or the Erlang/OTP `gen_server` behaviour).

Hu et al. [23] and Kouzapas et al. [27] introduced a session calculus with a session set type, arrival predicates, and a session typecase operation. Their approach provides the low-level constructs to implement an event loop; in contrast our work encodes an event loop directly in the operational

semantics, and includes first-class event handlers. Viering et al. [50] make use of event-driven programming in a framework for fault-tolerant session-typed distributed programming. Their programming model involves inversion of control on *output* events as well as inputs, and delegates the decision between output choices to the event loop; consequently, their progress depends on exhaustiveness of selection decision predicates, which their system does not verify. That is in contrast to Maty's support for safe direct-style session programming of non-blocking actions; our global progress is also stronger as it ensures possible progress on *every session in the system*. The above works are formalised as process calculi as opposed to a programming language design.

Behavioural types for actor languages. Mostrous and Vasconcelos [35] were first to investigate session typing for actors: they used Erlang's unique reference generation and selective receive construct to impose a channel-based communication discipline. The approach has never been implemented, and does not scale beyond binary session types. Francalanza and Tabone [16] implement a binary session typing system in Elixir, also using pre- and post-conditions on module-level functions. Their approach is limited in expressiveness as it is only possible to reason about the interaction between pairs of participants.

Our approach is inspired by the programming methodology put forward by Neykova and Yoshida [38] and implemented in Erlang by Fowler [12], but our language design allows for *static* checking and is formalised. Neykova and Yoshida [37] show how causality information in global types can optimise process restarts in Erlang supervision hierarchies, showing that protocol-guided recovery can lead to speedups over naïve Erlang recovery strategies without jeopardising soundness. Again this work relies on dynamic checking, but it would be interesting to see whether their failure handling strategies could be incorporated in our work.

Harvey et al. [20] introduce EnsembleS, which also uses a flow-sensitive effect type system to enforce session typing (with explicit connection actions [25]) in an actor language. Their main focus is using MPSTs to ensure safe *adaptive* systems, where each actor can discover other actors and safely replace their behaviour. Each EnsembleS actor can only take part in a single session at a time, ruling out many of the collaborative tasks (e.g., servers) that can benefit from our approach.

Mailbox types [10] (inspired by earlier work on behavioural types for typestate [9, 41]) capture the expected contents of a mailbox as a commutative regular expression. Mailbox types guarantee, for example, to ensure that a process consumes every message that it receives; Fowler et al. [13] design and implement a functional programming language with mailbox types and show that the type discipline scales to idiomatic actor style applications. Although both mailbox types and session types share the same aims, they solve different problems: in particular session types are better suited for structured interactions between a known number of participants, whereas mailbox types are better when the participants are not necessarily known but message ordering is unimportant.

Scalas et al. [47] introduce a behavioural typing discipline that includes dependent function types. Their resulting language makes it possible to check that functions satisfy interaction patterns written in a type-level DSL; the toolchain can then check whether collections of behavioural types satisfy properties (e.g., liveness or termination). Their behavioural type discipline is different to session typing (e.g., supporting parameterised server interactions but not branching choice). While both approaches have advantages, ours is deliberately session-based to support structured interactions between known participants; it is unclear how their actor API would scale to processes being involved in multiple session-style interactions.

7 CONCLUSION

Actor languages have been successful tools for constructing reliable and fault-tolerant distributed systems. Until now, however, it has been difficult to guarantee protocol conformance for *structured* actor interactions, leading to the possibility of communication mismatches and deadlocks.

In this paper we have addressed this problem using *multiparty session types*. We have proposed an actor language design that supports MPSTs by using a novel combination of a flow-sensitive effect system and first-class message handlers; notably, in contrast to previous work on session types for actor languages, our language can be checked *statically*, and actors can take part in *multiple sessions*. We have shown how session typing guarantees strong metatheoretical guarantees: preservation, progress, and (unusually for a MPST system) *global progress*. We have shown the extensibility of our approach through three extensions: state, support for proactively switching between sessions, and failure handling (with Erlang-style process supervision). Finally, we have shown an implementation in Scala using an API generation approach, and shown two case studies.

In future it would be interesting to: implement our approach in static typing tool for Erlang; to investigate whether our combination of first-class event handlers and effect typing could be used for other programming paradigms (e.g., for distributed workflow systems [48]); and investigate how the approach could be integrated with more sophisticated failure handling mechanisms (e.g., [37]).

REFERENCES

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

[2] Joe Armstrong. 2003. *Making reliable distributed systems in the presence of software errors.* Ph. D. Dissertation. Royal Institute of Technology, Stockholm, Sweden.

[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] Francesco Cesarini and Steve Vinoski. 2016. *Designing for Scalability with Erlang/OTP.* O'Reilly Media, Inc.

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

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

[10] 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.

[11] Romain Demangeon and Kohei Honda. 2012. Nested Protocols in Session Types. In *CONCUR (Lecture Notes in Computer Science, Vol. 7454)*. Springer, 272–286.

[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 Malcolm Hole. 2005. Subtyping for session types in the pi calculus. *Acta Informatica* 42, 2-3 (2005), 191–225. https://doi.org/10.1007/S00236-005-0177-Z

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

[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] Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. 2022. Multiparty GV: functional multiparty session types with certified deadlock freedom. *Proc. ACM Program. Lang.* 6, ICFP (2022), 466–495.

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

[28] 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.

[29] 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.

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

[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] Daniel Marino and Todd D. Millstein. 2009. A generic type-and-effect system. In *TLDI*. ACM, 39–50.

[33] 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.

[34] 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.

[35] 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.

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

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

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

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

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

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

[42] Riccardo Pucella and Jesse A. Tov. 2008. Haskell session types with (almost) no class. In *Haskell*. ACM, 25–36.

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

[44] 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.

[45] 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

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

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

[48] Jonas Spenger, Paris Carbone, and Philipp Haller. 2022. Portals: An Extension of Dataflow Streaming for Stateful Serverless. In *Onward!* ACM, 153–171.
[49] Peter Thiemann. 2023. Intrinsically Typed Sessions with Callbacks (Functional Pearl). *Proc. ACM Program. Lang.* 7, ICFP (2023), 711–739.
[50] 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.
[51] Philip Wadler. 2014. Propositions as sessions. *J. Funct. Program.* 24, 2-3 (2014), 384–418.
[52] 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.
[53] Fangyi Zhou, Francisco Ferreira, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. 2020. Statically verified refinements for multiparty protocols. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 148:1–148:30.

Appendices

APPENDIX CONTENTS

A DETAILS OF STATE EXTENSION

Figure 17 shows the full extension of Maty with state. The typing rules for functions and function applications are modified in order to record the type of the state on which the actor operates. We also modify the configuration typing rules to account for the stored state, and ensure that the currently-executing thread operates on a compatible state type.

Extended syntax

$$\begin{array}{llll} \text{Types} & A, B & ::= & \cdots \mid A_1 \xrightarrow[B]{S,T} A_2 \\ \text{Computations} & M, N & ::= & \cdots \mid \textbf{get} \mid \textbf{set}\ V \mid \textbf{spawn}\ M\ V \\ \text{Configurations} & \mathcal{C}, \mathcal{D} & ::= & \langle a, \mathcal{T}, \sigma, \rho, V \rangle \end{array}$$

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

$$\textsc{T-Abs}\ \frac{\Gamma, x : A_1 \mid B \mid S \triangleright M : A_2 \triangleleft T}{\Gamma \vdash \lambda x.M : A_1 \xrightarrow[B]{S,T} A_2} \qquad \textsc{T-App}\ \frac{\Gamma \vdash V : A_1 \xrightarrow[B]{S,T} A_2 \qquad \Gamma \vdash W : A_1}{\Gamma \mid B \mid S \triangleright V\ W : A_2 \triangleleft T}$$

$$\textsc{T-Get}\ \frac{}{\Gamma \mid A \mid S \triangleright \textbf{get} : A \triangleleft T} \qquad \textsc{T-Set}\ \frac{\Gamma \vdash V : A}{\Gamma \mid A \mid S \triangleright \textbf{set}\ V : \textbf{1} \triangleleft S} \qquad \textsc{T-Spawn}\ \frac{\Gamma \mid B \mid T \triangleright M : \textbf{1} \triangleleft \mathsf{end} \qquad \Gamma \vdash V : B}{\Gamma \mid A \mid S \triangleright \textbf{spawn}\ M\ V : \textbf{1} \triangleleft S}$$

Modified configuration typing rules $\boxed{\{A\}\ \Gamma; \Delta \vdash \mathcal{T}}$ $\boxed{\Gamma; \Delta \vdash \mathcal{C}}$

$$\textsc{TT-Idle}\ \frac{}{\{A\}\ \Gamma; \cdot \vdash \textbf{idle}} \qquad \textsc{TT-NoSess}\ \frac{\Gamma \mid A \mid S \triangleright M : \textbf{1} \triangleleft \mathsf{end}}{\{A\}\ \Gamma; s[\mathrm{p}] : S \vdash (M)^{s[\mathrm{p}]}} \qquad \textsc{TT-NoSess}\ \frac{\Gamma \mid A \mid \mathsf{end} \triangleright M : \textbf{1} \triangleleft \mathsf{end}}{\{A\}\ \Gamma; \cdot \vdash M}$$

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

Modified reduction rules

$$\begin{array}{lrcl} \textsc{E-Spawn} & \langle a, \mathcal{M}[\textbf{spawn}\ M\ W], \sigma, \rho, V \rangle & \longrightarrow & \langle a, \mathcal{M}[\textbf{return}\ ()], \sigma, \rho, V \rangle \parallel \langle b, M, \epsilon, \epsilon, W \rangle \\ \textsc{E-Get} & \langle a, \mathcal{M}[\textbf{get}], \sigma, \rho, V \rangle & \longrightarrow & \langle a, \mathcal{M}[\textbf{return}\ V], \sigma, \rho, V \rangle \\ \textsc{E-Set} & \langle a, \mathcal{M}[\textbf{set}\ W], \sigma, \rho, V \rangle & \longrightarrow & \langle a, \mathcal{M}[\textbf{return}\ ()], \sigma, \rho, W \rangle \end{array}$$

Fig. 17. Extension of Maty with state

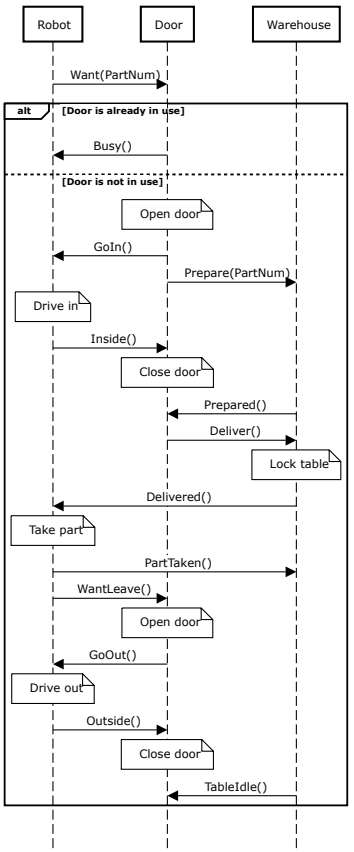
B DETAILS OF CASE STUDY PROTOCOLS

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;
  }
}
```

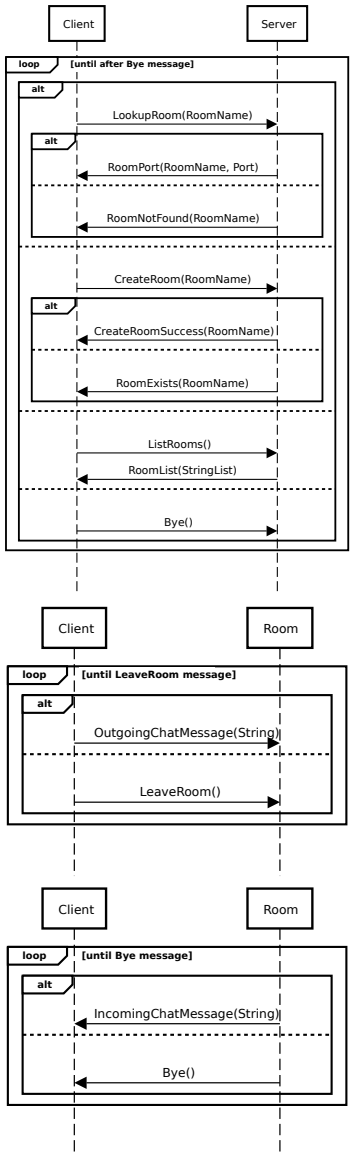


B.2 Chat Server

```
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 PROOFS

C.1 Preservation

We begin with some unsurprising auxiliary lemmas.

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

Proof. By induction on the derivation of $\Gamma_1, x : A \mid S \triangleright M : B \triangleleft T$. □

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

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

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

Proof. A standard induction on the derivation of $M \longrightarrow_M N$, 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* $\Gamma;\ \Delta \vdash C$ *and* $C \equiv \mathcal{D}$ *then there exists some* $\Delta' \equiv \Delta$ *such that* $\Gamma;\ \Delta' \vdash \mathcal{D}$.

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

Lemma C.10 (Preservation (Configuration reduction)). *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* $\Gamma;\ \Delta' \vdash \mathcal{D}$.

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

Case E-Send.

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

Assumption:

$$\dfrac{\dfrac{\dfrac{\Gamma \mid S \triangleright \mathcal{E}[\mathsf{q} \,!\, \ell(V)] : \mathbf{1} \triangleleft \mathsf{end}}{\Gamma; s[\mathsf{p}] : S \vdash (\mathcal{E}[\mathsf{q} \,!\, \ell(V)])^{s[\mathsf{p}]}} \qquad \Gamma; \Delta_2 \vdash \sigma \qquad \Gamma; \Delta_3 \vdash \rho}{\Gamma;\ s[\mathsf{p}], \Delta_2, \Delta_3 \vdash \langle a, (\mathcal{E}[\mathsf{q} \,!\, \ell(V)])^{s[\mathsf{p}]}, \sigma, \rho\rangle} \qquad \Gamma;\ s : Q \vdash s \triangleright \delta}{\Gamma;\ s[\mathsf{p}] : S, \Delta_2, \Delta_3, s : Q \vdash \langle a, (\mathcal{E}[\mathsf{q} \,!\, \ell(V)])^{s[\mathsf{p}]}, \sigma, \rho\rangle \parallel s \triangleright \delta}$$

By Lemma C.2 we have that $\Gamma \mid \mathsf{q} \oplus \{\ell_i(A_i) : T_i\}_{i \in I} \triangleright \mathsf{q} \,!\, \ell_j(V) : \mathbf{1} \triangleleft T_j$ and therefore that $S = \mathsf{q} \oplus \{\ell_i(A_i) : T_i\}_{i \in I}$.

Since $\Gamma \mid T_j \triangleright \mathbf{return}\ () : \mathbf{1} \triangleleft T_j$, we can show by Lemma C.3 we have that $\Gamma \mid T_j \triangleright \mathcal{E}[\mathbf{return}\ ()] : \mathbf{1} \triangleleft \mathsf{end}$.

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:

$$\dfrac{\dfrac{\dfrac{\Gamma \mid T_j \triangleright \mathcal{E}[\mathbf{return}\ ()] : \mathbf{1} \triangleleft \mathsf{end}}{\Gamma; s[\mathsf{p}] : T_j \vdash (\mathcal{E}[\mathbf{return}\ ()])^{s[\mathsf{p}]}} \qquad \Gamma; \Delta_2 \vdash \sigma \qquad \Gamma; \Delta_3 \vdash \rho}{\Gamma;\ s[\mathsf{p}] : T_j, \Delta_2, \Delta_3 \vdash \langle a, (\mathcal{E}[\mathbf{return}\ ()])^{s[\mathsf{p}]}, \sigma, \rho\rangle} \qquad \Gamma;\ s : Q \cdot (\mathsf{p}, \mathsf{q}, \ell_j(A_j)) \vdash s \triangleright \delta \cdot (\mathsf{p}, \mathsf{q}, \ell_j(V))}{\Gamma;\ s[\mathsf{p}] : T_j, \Delta_2, \Delta_3, s : Q \cdot (\mathsf{p}, \mathsf{q}, \ell_j(B_j)) \vdash \langle a, (\mathcal{E}[\mathbf{return}\ ()])^{s[\mathsf{p}]}, \sigma, \rho\rangle \parallel s \triangleright \delta \cdot (\mathsf{p}, \mathsf{q}, \ell_j(V))}$$

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.

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

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

Let **D** be the following derivation:

$$\dfrac{\Gamma;\cdot \vdash \textbf{idle} \qquad \dfrac{\dfrac{(\Gamma, x_i : B_i \mid S_i \triangleright M_i : \mathbf{1} \triangleleft \mathsf{end})_{i\in I}}{\Gamma \vdash \textbf{handler}\ \mathsf{q}\ \{(\ell_i(x_i) \mapsto M_i)_{i\in I}\} : \mathsf{Handler}(S^?)} \qquad \Gamma;\Delta_2 \vdash \sigma}{\Gamma;\Delta_2, s[\mathsf{p}] : S^? \vdash \sigma[s[\mathsf{p}] \mapsto \textbf{handler}\ \mathsf{q}\ \{\overrightarrow{H}\}]} \qquad \Gamma;\Delta_3 \vdash \rho}{\Gamma;\ \Delta_2, \Delta_3, s[\mathsf{p}] : S^? \vdash \langle a, \textbf{idle}, \sigma[s[\mathsf{p}] \mapsto \textbf{handler}\ \mathsf{q}\ \{\overrightarrow{H}\}], \rho\rangle}$$

Assumption:

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

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 $\Gamma, x : B_j \mid S_j \triangleright M : \mathbf{1} \triangleleft \mathsf{end}$.
By Lemma C.1, $\Gamma \mid S_j \triangleright M\{V/x_j\} : \mathbf{1} \triangleleft \mathsf{end}$.
Let $\mathbf{D}'$ be the following derivation:

$$\dfrac{\dfrac{\Gamma \mid S_j \triangleright M\{V/x_j\} : \mathbf{1} \triangleleft \mathsf{end}}{\Gamma; s[\mathsf{p}] : S_j \vdash (M\{V/x_j\})^{s[\mathsf{p}]}} \qquad \Gamma;\Delta_2 \vdash \sigma\Gamma;\Delta_3 \vdash \rho}{\Gamma;\ \Delta_2, \Delta_3, s[\mathsf{p}] : S_j \vdash \langle a, (M\{V/x_j\})^{s[\mathsf{p}]}, \sigma, \rho\rangle}$$

Recomposing:

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

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.

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

Assumption:

$$\dfrac{\dfrac{\Gamma \mid S \triangleright \mathcal{E}[\textbf{suspend}\ V] : \mathbf{1} \triangleleft \mathsf{end}}{\Gamma; s[\mathsf{p}] : S \vdash (\mathcal{E}[\textbf{suspend}\ V])^{s[\mathsf{p}]}} \qquad \Gamma;\Delta_2 \vdash \sigma \qquad \Gamma;\Delta_3 \vdash \rho}{\Gamma;\ s[\mathsf{p}] : S, \Delta_2, \Delta_3 \vdash \langle a, (\mathcal{E}[\textbf{suspend}\ V])^{s[\mathsf{p}]}, \sigma, \rho\rangle}$$

By Lemma C.2 we have that:

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

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

Recomposing:

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

as required.

Case E-Spawn.

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

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

Assumption:

$$\dfrac{\dfrac{\Gamma \mid S \triangleright \mathcal{E}[\textbf{spawn}\ M] : \mathbf{1} \triangleleft \mathsf{end}}{\Gamma; s[\mathrm{p}] : S \vdash (\mathcal{E}[\textbf{spawn}\ M])^{s[\mathrm{p}]}} \qquad \Gamma;\Delta_2 \vdash \sigma \qquad \Gamma;\Delta_3 \vdash \rho}{\Gamma;\, \Delta_2, \Delta_3, s[\mathrm{p}] : S \vdash \langle a, (\mathcal{E}[\textbf{spawn}\ M])^{s[\mathrm{p}]}, \sigma, \rho\rangle}$$

By Lemma C.2:

$$\dfrac{\Gamma \mid \mathsf{end} \triangleright M : \mathbf{1} \triangleleft \mathsf{end}}{\Gamma \mid S \triangleright \textbf{spawn}\ M : \mathbf{1} \triangleleft S}$$

By Lemma C.3, $\Gamma \mid S \triangleright \mathcal{E}[\textbf{return}\ ()] : \mathbf{1} \triangleleft \mathsf{end}$.

Thus, recomposing:

$$\dfrac{\dfrac{\dfrac{\Gamma \mid S \triangleright \mathcal{E}[\textbf{return}\ ()] : \mathbf{1} \triangleleft \mathsf{end}}{\Gamma; s[\mathrm{p}] : S \vdash (\mathcal{E}[\textbf{return}\ ()])^{s[\mathrm{p}]}} \quad \begin{array}{l}\Gamma;\Delta_2 \vdash \sigma\\ \Gamma;\Delta_3 \vdash \rho\end{array}}{\Gamma;\, \Delta_2, \Delta_3, s[\mathrm{p}] : S \vdash \langle a, (\mathcal{E}[\textbf{return}\ ()])^{s[\mathrm{p}]}, \sigma, \rho\rangle} \qquad \dfrac{\dfrac{\Gamma \mid \mathsf{end} \triangleright M : \mathbf{1} \triangleleft \mathsf{end}}{\Gamma;\cdot \vdash M} \qquad \Gamma;\cdot \vdash \epsilon \qquad \Gamma;\cdot \vdash \epsilon}{\Gamma;\, \cdot \vdash \langle a, M, \epsilon, \epsilon\rangle}}{\Gamma;\, \Delta_2, \Delta_3, s[\mathrm{p}] : S \vdash \langle a, (\mathcal{E}[\textbf{return}\ ()])^{s[\mathrm{p}]}, \sigma, \rho\rangle \parallel \langle a, M, \epsilon, \epsilon\rangle}$$

Case E-Reset.

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

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

Assumption:

$$\dfrac{\dfrac{\Gamma \mid \mathsf{end} \triangleright \textbf{return}\ () : \mathbf{1} \triangleleft \mathsf{end}}{\Gamma; s[\mathrm{p}] : \mathsf{end} \vdash (\textbf{return}\ ())^{s[\mathrm{p}]}} \qquad \Gamma;\Delta_2 \vdash \sigma \qquad \Gamma;\Delta_3 \vdash \rho}{\Gamma;\, \Delta_2, \Delta_3, s[\mathrm{p}] : \mathsf{end} \vdash \langle a, (\textbf{return}\ ())^{s[\mathrm{p}]}, \sigma, \rho\rangle}$$

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

$$\frac{\Gamma; \cdot \vdash \mathbf{idle} \qquad \Gamma; \Delta_2 \vdash \sigma \qquad \Gamma; \Delta_3 \vdash \rho}{\Gamma;\ \Delta_2, \Delta_3 \vdash \langle a, \mathbf{idle}, \sigma, \rho\rangle}$$

as required.
Case E-NewAP.

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

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

$$\frac{\dfrac{\Gamma \mid T \triangleright \mathcal{E}[\mathbf{newAP}_{(\mathrm{p}_i:S_i)_{i\in I}}] : \mathbf{1} \triangleleft \mathsf{end}}{\Gamma; s[\mathrm{p}] : T \vdash (\mathcal{E}[\mathbf{newAP}_{(\mathrm{p}_i:S_i)_{i\in I}}])^{s[\mathrm{p}]}} \qquad \begin{array}{l}\Gamma; \Delta_2 \vdash \sigma \\ \Gamma; \Delta_3 \vdash \rho\end{array}}{\Gamma;\ \Delta_2, \Delta_3 \vdash \langle a, (\mathcal{E}[\mathbf{newAP}_{(\mathrm{p}_i:S_i)_{i\in I}}])^{s[\mathrm{p}]}, \sigma, \rho\rangle}$$

By Lemma C.2:

$$\frac{\varphi \text{ is a safety property} \qquad \varphi((\mathrm{p}_i : S_i)_{i\in I})}{\Gamma \mid T \triangleright \mathbf{newAP}_{(\mathrm{p}_i:S_i)_{i\in I}} : \mathsf{AP}((\mathrm{p}_i : S_i)_{i\in I}) \triangleleft T}$$

By Lemma C.3, $\Gamma, c : \mathsf{AP}((\mathrm{p}_i : S_i)_{i\in I}) \mid T \triangleright \mathcal{E}[\mathbf{return}\ c] : \mathbf{1} \triangleleft \mathsf{end}$.
Let $\Gamma' = \Gamma, c : \mathsf{AP}((\mathrm{p}_i : S_i)_{i\in I})$.
By Lemma C.4, since c is fresh we have that $\Gamma'; \Delta_2 \vdash \sigma$ and $\Gamma'; \Delta_3 \vdash \rho$.
Recomposing:

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

as required.
Case E-Register.

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

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

$$\frac{\dfrac{\Gamma \mid T \triangleright \mathcal{E}[\mathbf{register}\ p\ \mathrm{p}_j\ M] : \mathbf{1} \triangleleft \mathsf{end}}{\Gamma; s[\mathrm{q}] : T \vdash (\mathcal{E}[\mathbf{register}\ p\ \mathrm{p}_j\ M])^{s[\mathrm{q}]}} \qquad \begin{array}{l}\Gamma; \Delta_2 \vdash \sigma \\ \Gamma; \Delta_3 \vdash \rho\end{array}}{\Gamma;\ \Delta_2, \Delta_3, s[\mathrm{q}] : T \vdash \langle a, (\mathcal{E}[\mathbf{register}\ p\ \mathrm{p}_j\ M])^{s[\mathrm{q}]}, \sigma, \rho\rangle}$$

Assumption:

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

By Lemma C.2:

$$\dfrac{\Gamma \vdash c : \mathsf{AP}((\mathsf{p}_i : S_i)_i) \qquad \Gamma \mid S_j \triangleright M : \mathbf{1} \triangleleft \mathsf{end}}{\Gamma \mid T \triangleright \mathbf{register}\ c\ \mathsf{p}_j\ M : \mathbf{1} \triangleleft T}$$

By Lemma C.3, $\Gamma \mid T \triangleright \mathcal{E}[\mathbf{return}\ ()] : \mathbf{1} \triangleleft \mathsf{end}$.
Now, let $\mathbf{D}'$ be the following derivation:

$$\dfrac{\dfrac{\Gamma \mid T \triangleright \mathcal{E}[\mathbf{return}\ ()] : \mathbf{1} \triangleleft \mathsf{end}}{\Gamma; s[\mathsf{q}] : T \vdash (\mathcal{E}[\mathbf{return}\ ()])^{s[\mathsf{q}]}} \qquad \Gamma;\Delta_2 \vdash \sigma \qquad \dfrac{\Gamma \mid S_j \triangleright M : \mathbf{1} \triangleleft \mathsf{end} \qquad \Gamma;\Delta_3 \vdash \rho}{\Gamma;\Delta_3, \iota^+ : S_j \vdash \rho[\iota^+ \mapsto M]}}{\Gamma;\ \Delta_2, \Delta_3, s[\mathsf{q}] : S, \iota^+ : S_j \vdash \langle a, (\mathcal{E}[\mathbf{return}\ ()])^{s[\mathsf{q}]}, \sigma, \rho\rangle}$$

Finally, we can recompose:

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

as required.
Case E-Init.

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

For each actor composed in parallel we have:

$$\dfrac{\Gamma;\cdot \vdash \mathbf{idle} \qquad \Gamma;\Delta_{i_2} \vdash \sigma_i \qquad \dfrac{\Gamma \mid S_i \triangleright M_i : \mathbf{1} \triangleleft \mathsf{end} \qquad \Gamma;\Delta_{i_3} \vdash \rho}{\Gamma;\Delta_{i_3}, \iota_i^+ : S_i \vdash \rho_i[\iota_{\mathsf{p}_i} \mapsto M_i]}}{\Gamma;\ \Delta_{i_2}, \Delta_{i_3}, \iota_i^+ : S_i, a_i \vdash \langle a_i, \mathbf{idle}, \sigma_i, \rho_i\rangle}$$

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

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

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

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

$$\dfrac{\dfrac{\Gamma \mid S_i \triangleright M_i : \mathbf{1} \triangleleft \mathsf{end}}{\Gamma; s[\mathrm{p}_i] : S_i \vdash (M_i)^{s[\mathrm{p}_i]}} \qquad \Gamma; \Delta_{i_2} \vdash \sigma_i \qquad \Gamma; \Delta_{i_3} \vdash \rho_i}{\Gamma;\ \Delta_{i_2}, \Delta_{i_3}, s[\mathrm{p}_i] : S_i \vdash \langle a, (M_i)^{s[\mathrm{p}_i]}, \sigma_i, \rho_i\rangle}$$

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

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

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

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

(⟶) If $\Gamma;\ \Delta \vdash \mathcal{C}$ *with safe*(Δ) *and* $\mathcal{C} \longrightarrow \mathcal{D}$, *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.2 Progress

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

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

Functional reduction satisfies progress.

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

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

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

The functional reduction result allows us to show thread progress.

Lemma 3.6 (Thread Progress). *Let $C = \mathcal{G}[\langle a, \mathcal{T}, \sigma, \rho\rangle]$. If $\cdot; \cdot \vdash C$ then either $\mathcal{T} = \mathbf{idle}$, or there exist $\mathcal{G}', \mathcal{T}', \sigma', \rho'$ such that $C \longrightarrow \mathcal{G}'[\langle a, \mathcal{T}', \sigma', \rho'\rangle]$.*

Proof. If $\mathcal{T} = \mathbf{idle}$ then the theorem is satisfied, so consider the cases where $\mathcal{T} = M$ or $\mathcal{T} = (M)^{s[\mathsf{p}]}$. 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 communication or concurrency construct. Of these:

- **spawn** N can reduce by E-Spawn
- **suspend** V can reduce by E-Suspend
- $\mathbf{newAP}_{(\mathsf{p}_i:S_i)_i}$ can reduce by E-NewAP

Next, consider **register** p 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 $M = \mathsf{q} \,!\, \ell(V)$. It cannot be the case that $\mathcal{T} = \mathsf{q} \,!\, \ell(V)$ 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 $\mathcal{T} = (\mathsf{q} \,!\, \ell(V))^{s[\mathsf{p}]}$ for some s, p. Again since the initial runtime typing environment is empty, it must be the case that s is ν-bound and so by T-SessionName and T-EmptyQueue/T-ConsQueue there must be some session queue $s \triangleright \delta$. The thread must therefore be able to reduce by E-Send. □

Theorem 3.7 (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:*

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

Proof. By Proposition 3.4 C can be written in canonical form:

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

By repeated applications of Lemma 3.6, either the configuration can reduce or all threads are idle:

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

Additional Syntax

$$\text{Labels} \quad \mathscr{L} \quad ::= \quad \tau \mid \mathsf{spawn}(M) \mid \mathsf{send}(\mathsf{p}, \ell, V) \mid \mathsf{newAP}(a) \mid \mathsf{register}(p, \mathsf{p}, M)$$

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

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. 18. LTS Semantics for Computations

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

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

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

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

□

C.3 Global Progress

The overview of the global progress proof is as follows:

- We design a labelled transition system semantics for term reduction (Figure 18).
- We argue that our LTS is strongly-normalising up to **suspend** (Proposition C.12).
- 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.3.1 LTS Semantics.

Figure 18 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.12 (Strong Normalisation (LTS)). *If* $\Psi \mid S \triangleright^{\mathrm{f}} M : A \triangleleft$ *end then there exists some finite reduction sequence such that either:*

- $M \overset{\mathscr{L}_1}{\hookrightarrow} \cdots \overset{\mathscr{L}_n}{\hookrightarrow} V$ *for some* V*; or*
- $M \overset{\mathscr{L}_1}{\hookrightarrow} \cdots \overset{\mathscr{L}_n}{\hookrightarrow} \mathcal{E}[\textbf{\textit{suspend}}\ V]$ *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.13 (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:

Lemma C.14 (Term reduction under contexts). *Let* $C = \mathcal{G}[\langle a, \mathcal{Q}[\mathcal{E}[M]], \sigma, \rho\rangle]$. *If* $\Gamma; \Delta \vdash C$ *and* $M \overset{\tau}{\hookrightarrow} N$, *then* $C \longrightarrow \mathcal{G}[\langle a, \mathcal{Q}[\mathcal{E}[N]], \sigma, \rho\rangle]$.

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

Lemma C.15 (Simulation). *Suppose* $\Psi; \cdot \vdash^{\mathrm{f}} C$ *where* $C = \mathcal{G}[\langle a, \mathcal{Q}[M], \sigma, \rho\rangle]$. *If* $M \overset{\mathscr{L}}{\hookrightarrow} N$, *then there exist some* $\mathcal{G}'$, σ', *and* ρ' *such that* $C \longrightarrow \mathcal{G}'[\langle a, \mathcal{Q}[M'], \sigma', \rho'\rangle]$.

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

Case LTS-Spawn.

Assumption:

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

By Lemma C.13, E-Spawn, and E-Lift,

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

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

Case LTS-Send.

Assumption:

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

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

Case LTS-NewAP.

Assumption:

$$\frac{}{\textbf{newAP}_{(\mathrm{p}_i:S_i)_i} \overset{\mathsf{newAP}(a)}{\hookrightarrow} \textbf{return}\ a}$$

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

Case LTS-Register.
Assumption:

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

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

Case LTS-Beta.
Immediate by Lemma C.14.

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

Lemma C.16 (Determinism (Term Reduction)). *Suppose* $\Psi; \Delta \vdash^{\mathsf{f}} C$ *where* $C = \mathcal{G}[\langle a, \mathcal{Q}[M], \sigma, \rho\rangle]$. *If:*

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

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-Send, E-Spawn, E-NewAP, and E-Register. □

Lemma C.17 (Reflection). *Suppose* $\Psi; \Delta \vdash^{\mathsf{f}} C$ *where* $C = \mathcal{G}[\langle a, \mathcal{Q}[M], \sigma, \rho\rangle]$.
If $C \longrightarrow \mathcal{G}'[\langle a, \mathcal{Q}[N], \sigma', \rho'\rangle]$ *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.16, the reduction from C must be unique, and will be the one specified by Lemma C.15. □

Proposition C.18 (Operational Correspondence).

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

- *If* $M \overset{\mathscr{L}}{\hookrightarrow} N$, *then there exist some* $\mathcal{G}'$, σ', *and* ρ' *such that* $C \longrightarrow \mathcal{G}'[\langle a, \mathcal{Q}[N], \sigma', \rho'\rangle]$.
- *If* $C \longrightarrow \mathcal{G}'[\langle a, \mathcal{Q}[N], \sigma', \rho'\rangle]$ *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.15 and C.17. □

Lemma C.19. *If* $\cdot; \cdot \vdash (\nu s : \Delta)C$ *and* $C \xrightarrow{\tau} \mathcal{D}$, *then* $\cdot; \cdot \vdash (\nu s : \Delta)\mathcal{D}$.

Proof. A straightforward corollary of Theorem 3.2. □

Theorem 3.9 (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 $\cdot; s[\mathrm{p}_1] : S_1, \ldots, s[\mathrm{p}_n] : S_n \vdash^{\mathrm{f}} C$ and thus by the linearity of Δ_s alongside rule T-Actor we have some set of actors:

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

such that for each role p_j for $j \in 1..n$, either:

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

Consider the subset of actors where $\mathcal{T}_i \neq \mathbf{idle}$, i.e., $\mathcal{T}_i = N_i$ or $\mathcal{T}_i = (N_i)^{s'[\mathrm{p}_j]}$ for some N_i. In this case, for each actor, by Proposition C.12 we have that $N_i \xhookrightarrow{\mathscr{L}_1} \cdots \xhookrightarrow{\mathscr{L}_n} N_i'$ where either $N_i' = \mathbf{return}\ ()$, or $N_i' = E[\mathbf{suspend}\ V]$ for some value V. By Proposition C.18, 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:

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

By Lemma C.19 we have that $\cdot; \cdot \vdash^{\mathrm{f}}_{\mathrm{prog}} (\nu s : \Delta_s)\mathcal{D}$ and thus it remains the case that $\Delta \Longrightarrow$. Thus by similar reasoning to Theorem 3.7 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 4

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

D.1 Maty-$\rightleftarrows$

D.1.1 *Preservation.*

Theorem 4.1 (Preservation). *Preservation (as defined in Theorem 3.2) continues to hold in Maty-$\rightleftarrows$.*

Proof. Preservation of typing under structural congruence follows straightforwardly.

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

Case E-Suspend$_!$-1.

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

Case E-Suspend$_!$-2.

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

Assumption:

$$\dfrac{\dfrac{\Gamma \mid S \triangleright \mathcal{E}[\mathbf{suspend}_! \ \underline{\mathrm{s}} \ V] : \mathbf{1} \triangleleft \mathsf{end}}{\Gamma; s[\mathrm{p}] : S \vdash (\mathcal{E}[\mathbf{suspend}_! \ \underline{\mathrm{s}} \ V])^{s[\mathrm{p}]}} \qquad \dfrac{\Gamma; \Delta_1 \vdash \sigma \qquad \Sigma(\underline{\mathrm{s}}) = (S^!, A) \qquad (\Gamma \vdash W_i : A \xrightarrow{S^!,\mathsf{end}} \mathbf{1})_i}{\Gamma; \Delta_1, (s_i[\mathrm{q}_i] : S^!)_i \vdash \sigma[\underline{\mathrm{s}} \mapsto (s_i[\mathrm{q}_i], W_i)_i]} \qquad \begin{array}{l}\Gamma; \Delta_2 \vdash \rho \\ \Gamma \vdash \Delta_3\theta\end{array}}{\Gamma; \Delta_1, \Delta_2, \Delta_3, s[\mathrm{p}] : S, (s_i[\mathrm{q}_i] : S^!)_i, a \vdash \langle a, \mathbf{idle}, \sigma[\underline{\mathrm{s}} \mapsto (s_i[\mathrm{q}_i], W_i)_i], \rho, \theta\rangle}$$

Consider the subderivation $\Gamma \mid S \triangleright \mathcal{E}[\mathbf{suspend}_! \ \underline{\mathrm{s}} \ V] : \mathbf{1} \triangleleft \mathsf{end}$. By Lemma C.2 there exists a subderivation:

$$\dfrac{\Sigma(\underline{\mathrm{s}}) = (S^!, A) \qquad \Gamma \vdash V : A \xrightarrow{S^!,\mathsf{end}} \mathbf{1}}{\Gamma \mid S^! \triangleright \mathbf{suspend}_! \ \underline{\mathrm{s}} \ V : \mathbf{1} \triangleleft \mathsf{end}}$$

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

Recomposing:

$$\dfrac{\dfrac{}{\Gamma; \cdot \vdash \mathbf{idle}} \qquad \dfrac{\Gamma; \Delta_1 \vdash \sigma \qquad \Sigma(\underline{\mathrm{s}}) = (S^!, A) \qquad (\Gamma \vdash W_i : A \xrightarrow{S^!,\mathsf{end}} \mathbf{1})_i \qquad \Gamma \vdash V : A \xrightarrow{S^!,\mathsf{end}} \mathbf{1}}{\Gamma; \Delta_1, (s_i[\mathrm{q}_i] : S^!)_i, s[\mathrm{p}] : S^! \vdash \sigma[\underline{\mathrm{s}} \mapsto (s_i[\mathrm{q}_i], W_i)_i \cdot (s[\mathrm{p}], V)]} \qquad \begin{array}{l}\Gamma; \Delta_2 \vdash \rho \\ \Gamma \vdash \Delta_3\theta\end{array}}{\Gamma; \Delta_1, \Delta_2, \Delta_3, s[\mathrm{p}] : S, (s_i[\mathrm{q}_i] : S^!)_i, a \vdash \langle a, (\mathcal{E}[\mathbf{suspend}_! \ \underline{\mathrm{s}} \ V])^{s[\mathrm{p}]}, \sigma[\underline{\mathrm{s}} \mapsto (s_i[\mathrm{q}_i], W_i)_i \cdot (s[\mathrm{p}], V)], \rho, \theta\rangle}$$

as required.

Case E-Become.

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

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

$$\dfrac{\dfrac{\Gamma \mid S \triangleright \mathcal{E}[\mathbf{become} \ \underline{\mathrm{s}} \ V] : \mathbf{1} \triangleleft \mathsf{end}}{\Gamma; \cdot \vdash \mathcal{E}[\mathbf{become} \ \underline{\mathrm{s}} \ V]} \qquad \Gamma; \Delta_1 \vdash \sigma \qquad \Gamma; \Delta_2 \vdash \rho \qquad \Gamma \vdash \Delta_3\theta}{\Gamma; \Delta_1, \Delta_2, \Delta_3, a \vdash \langle a, \mathcal{T}, \sigma, \rho, \theta\rangle}$$

By Lemma C.2 we have:

$$\frac{\Sigma(\underline{s}) = (T, A) \qquad \Gamma \vdash V : A}{\Gamma \mid S \triangleright \textbf{become}\ \underline{s}\ V : \mathbf{1} \triangleleft S}$$

By Lemma C.3 we can show that $\Gamma \mid S \triangleright \mathcal{E}[\textbf{return}\ ()] : \mathbf{1} \triangleleft \text{end}$.
Recomposing:

$$\frac{\dfrac{\Gamma \mid S \triangleright \mathcal{E}[\textbf{return}\ ()] : \mathbf{1} \triangleleft \text{end}}{\Gamma; \cdot \vdash \mathcal{E}[\textbf{return}\ ()]} \qquad \Gamma; \Delta_1 \vdash \sigma \qquad \Gamma; \Delta_2 \vdash \rho \qquad \dfrac{\Gamma \vdash \Delta\theta \qquad \Sigma(\underline{s}) = (S^!, A) \qquad \Gamma \vdash V : A}{\Gamma \vdash \Delta_3\theta \cdot (\underline{s}, V)}}{\Gamma; \Delta_1, \Delta_2, \Delta_3, a \vdash \langle a, \mathcal{E}[\textbf{return}\ ()], \sigma, \rho, \theta \cdot (\underline{s}, V)\rangle}$$

as required.

Case E-Activate.

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

Assumption:

$$\frac{\dfrac{}{\Gamma; \cdot \vdash \textbf{idle}} \qquad \dfrac{\Sigma(\underline{s}) = (S^!, A) \qquad \Gamma \vdash V : A \xrightarrow{S^!,\text{end}} \mathbf{1} \qquad (\Gamma \vdash V_i : A \xrightarrow{S^!,\text{end}} \mathbf{1})_i}{\Gamma; \Delta_1, s[\mathrm{p}] : S^!, (s_i[\mathrm{p}_i] : S^!)_i \vdash \sigma, \underline{s} \mapsto (s[\mathrm{p}], V) \cdot (s_i[\mathrm{p}_i], V_i)_i} \qquad \Gamma; \Delta_2 \vdash \rho \qquad \dfrac{\Gamma \vdash \Delta_3\theta \quad \Sigma(\underline{s}) = (S^!, A) \quad \Gamma \vdash W : A}{\Gamma \vdash \Delta_3(\underline{s}, W) \cdot \theta}}{\Gamma; \Delta_1, \Delta_2, s[\mathrm{p}] : S^!, (s_i[\mathrm{p}_i] : S^!)_i, \Delta_3, a \vdash \langle a, \textbf{idle}, \sigma[\underline{s} \mapsto (s[\mathrm{p}], V) \cdot (s_i[\mathrm{p}_i], V_i)_i], \rho, (\underline{s}, W) \cdot \theta\rangle}$$

Recomposing:

$$\frac{\dfrac{\dfrac{\Gamma \vdash V : A \xrightarrow{S^!,\text{end}} \mathbf{1} \qquad \Gamma \vdash W : A}{\Gamma \mid S^! \triangleright V\ W : \mathbf{1} \triangleleft \text{end}}}{\Gamma; s[\mathrm{p}] : S^! \vdash (V\ W)^{s[\mathrm{p}]}} \qquad \dfrac{\Gamma; \Delta_1 \vdash \sigma \qquad \Sigma(\underline{s}) = (S^!, A) \qquad (\Gamma \vdash V_i : A \xrightarrow{S^!,\text{end}} \mathbf{1})_i}{\Gamma; \Delta_1, (s_i[\mathrm{p}_i] : S^!)_i \vdash \sigma, \underline{s} \mapsto (s_i[\mathrm{p}_i], V_i)_i} \qquad \Gamma; \Delta_2 \vdash \rho \qquad \Gamma \vdash \Delta_3\theta}{\Gamma; \Delta_1, \Delta_2, s[\mathrm{p}] : S^!, (s_i[\mathrm{p}_i] : S^!)_i, \Delta_3, a \vdash \langle a, (V\ W)^{s[\mathrm{p}]}, \sigma[\underline{s} \mapsto (s_i[\mathrm{p}_i], V_i)_i], \rho, \theta\rangle}$$

as required. □

D.1.2 *Progress.*

Theorem 4.2 (Weak progress (Maty-⇄)). *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:*

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

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

Proof. The proof follows that of Theorem 3.7. Thread progress (Lemma 3.6) holds as before, since we can always evaluate **become** by E-Become, and we can always evaluate **suspend**$_!$ by E-Suspend-!$_1$ or E-Suspend-!$_2$.

Following the same reasoning as Theorem 3.7 we can write C in canonical form, where all threads are idle:

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

However, there are now *three* places each role endpoint $s[\mathrm{p}]$ can be used: either by TT-Sess to run a term in the context of a session or by TH-Handler to record a receive-suspended session

type as before, but now also by TH-SendHandler to record a send-suspended session type. As before, the former is impossible as all threads are idle, so now we must consider the cases for TH-Handler.

Following the same reasoning as Theorem 3.7, 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-$\lightning$

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

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

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

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

Theorem 4.3 (Preservation ($\longrightarrow$, Maty-$\lightning$)). *If $\Gamma;\ \Phi \vdash C$ with $\mathit{safe}(\Phi)$ and $C \longrightarrow \mathcal{D}$, 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 $C \longrightarrow \mathcal{D}$, concentrating on the new rules rather than the adapted rules (which are straightforward changes to the existing proof).

Case E-Monitor.

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

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

Assumption:

$$\frac{\dfrac{\Gamma \mid S \triangleright \mathcal{E}[\mathbf{monitor}\ b\ M] : \mathbf{1} \triangleleft \mathsf{end}}{\Gamma; \cdot \vdash \mathcal{E}[\mathbf{monitor}\ b\ M]} \qquad \Gamma; \Phi_1 \vdash \sigma \qquad \Gamma; \Phi_2 \vdash \rho}{\Gamma;\ \Phi_1, \Delta_2, a \vdash \langle a, \mathcal{E}[\mathbf{monitor}\ b\ M], \sigma, \rho, \omega\rangle}$$

where $\forall (b, N) \in \omega.\ \Gamma \vdash b : \mathsf{Pid} \ \wedge\ \Gamma \mid \mathsf{end} \triangleright N : \mathbf{1} \triangleleft \mathsf{end}$.

By Lemma C.2, we know:

$$\frac{\Gamma \vdash b : \mathsf{Pid} \qquad \Gamma \mid \mathsf{end} \triangleright M : \mathbf{1} \triangleleft \mathsf{end}}{\Gamma \mid S \triangleright \mathbf{monitor}\ b\ M : \mathbf{1} \triangleleft S}$$

By Lemma C.3 we know $\Gamma \mid S \triangleright \mathcal{E}[\mathbf{return}\ ()] : \mathbf{1} \triangleleft \mathsf{end}$.

Recomposing:

$$\frac{\dfrac{\Gamma \mid S \triangleright \mathcal{E}[\mathbf{return}\ ()] : \mathbf{1} \triangleleft \mathsf{end}}{\Gamma; \cdot \vdash \mathcal{E}[\mathbf{return}\ ()]} \qquad \Gamma; \Phi_1 \vdash \sigma \qquad \Gamma; \Phi_2 \vdash \rho}{\Gamma;\ \Phi_1, \Delta_2, a \vdash \langle a, \mathcal{E}[\mathbf{return}\ ()], \sigma, \rho, \omega \cup (b, N)\rangle}$$

noting that $\omega \cup (b, N)$ is safe since $\Gamma \vdash b : \mathsf{Pid}$ and $\Gamma \mid S \triangleright \mathcal{E}[\mathbf{return}\ ()] : \mathbf{1} \triangleleft \mathsf{end}$, as required.

Case E-InvokeM.

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

Assumption:

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

where $\forall (a', M) \in \omega \cup \{(b, N)\}.\ \Gamma \vdash b : \mathsf{Pid} \ \wedge\ \Gamma \mid \mathsf{end} \triangleright M : \mathbf{1} \triangleleft \mathsf{end}$.
Recomposing:

$$\dfrac{\dfrac{\dfrac{\Gamma \mid \mathsf{end} \triangleright M : \mathbf{1} \triangleleft \mathsf{end}}{\Gamma; \cdot \vdash M} \qquad \Gamma; \Phi_1 \vdash \sigma \qquad \Gamma; \Phi_2 \vdash \rho}{\Gamma;\ \Phi_1, \Phi_2, a \vdash \langle a, M, \sigma, \rho, \omega\rangle} \qquad \dfrac{}{\Gamma;\ b \vdash \lightning b}}{\Gamma;\ \Phi_1, \Phi_2, a, b \vdash \langle a, M, \sigma, \rho, \omega\rangle \parallel \lightning b}$$

as required.

Case E-Raise.

Similar to E-RaiseS.

Case E-RaiseS.

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

$$\dfrac{\dfrac{\Gamma \mid S \triangleright \mathcal{E}[\mathbf{raise}] : \mathbf{1} \triangleleft \mathsf{end}}{\Gamma; s[\mathsf{p}] : S \vdash (\mathcal{E}[\mathbf{raise}])^{s[\mathsf{p}]}} \qquad \Gamma; \Phi_1 \vdash \sigma \qquad \Gamma; \Phi_2 \vdash \rho}{\Gamma;\ \Phi_1, \Phi_2, s[\mathsf{p}] : S, a \vdash \langle a, (\mathcal{E}[\mathbf{raise}])^{s[\mathsf{p}]}, \sigma, \rho, \omega\rangle}$$

where $\forall (b, M) \in \omega.\ \Gamma \vdash b : \mathsf{Pid} \ \wedge\ \Gamma \mid \mathsf{end} \triangleright M : \mathbf{1} \triangleleft \mathsf{end}$.

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

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:

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

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:

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

Recomposing:

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

as required.

Case E-CancelH.

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

Let **D** be the following derivation:

$$\dfrac{\Gamma;\cdot \vdash \mathbf{idle} \qquad \dfrac{\dfrac{T = \mathsf{q}\,\&\{\ell_i(x_i) \mapsto S_i\}_i \qquad \Gamma \vdash V : \mathsf{Handler}(T)}{\Gamma \mid \mathsf{end} \triangleright M : \mathbf{1} \triangleleft \mathsf{end} \qquad \Gamma; \Phi_1 \vdash \sigma}}{\Gamma;\, \Phi_1, s[\mathsf{p}] : T \vdash \sigma[s[\mathsf{p}] \mapsto (V, M)]} \qquad \Gamma; \Phi_2 \vdash \rho}{\Gamma;\, \Phi_1, \Phi_2, s[\mathsf{p}] : T, a \vdash \langle a, \mathbf{idle}, \sigma[s[\mathsf{p}] \mapsto (V, M)], \rho, \omega\rangle}$$

Assumption:

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

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

$$\dfrac{\dfrac{\Gamma \mid \mathsf{end} \triangleright M : \mathbf{1} \triangleleft \mathsf{end}}{\Gamma; \cdot \vdash M} \qquad \Gamma; \Phi_1 \vdash \sigma \qquad \Gamma; \Phi_2 \vdash \rho}{\Gamma;\ \Phi_1, \Phi_2, a \vdash \langle a, M, \sigma, \rho, \omega \rangle}$$

Then we can construct the remaining derivation:

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

Finally, we need to show environment reduction:

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

as required. □

D.2.2 Progress. We first need to define a canonical form that takes zapper threads into account.

Definition D.2 (Canonical form (Maty − ↯)). A Maty − ↯ configuration C is in *canonical form* if it can be written:

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

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

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

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

Proof. Zapping a role may prevent Lbl-Recv from firing, but in this case would enable either a Lbl-ZapRecv 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.5 (Thread Progress). *Let* $C = \mathcal{G}[\langle a, \mathcal{T}, \sigma, \rho \rangle]$. *If* $\cdot; \cdot \vdash C$ *then either:*

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

Proof. As with Lemma 3.6 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.6 (Progress (Maty-↯)). *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:*

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

Proof. The reasoning is similar to that of Theorem 3.7. By Lemma D.3, C can be written in canonical form:

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

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

By repeated applications of Lemma D.5, either the configuration can reduce or all threads are idle:

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

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

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

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

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

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

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

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

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

as required. □

D.2.3 Global Progress.

Lemma D.7 (Session Progress (Maty-↯)). *If $\cdot;\cdot \vdash^{\mathrm{f}}_{prog} (\nu s : \Delta_s)C$, 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 3.9, except we must account for failed sessions arising as a consequence of reduction. □

Theorem D.8 (Global Progress (Maty-$\not\downarrow$)). *If* $\cdot; \cdot \vdash^{\mathsf{f}}_{prog_{\not\downarrow}} C$, *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.7. □