Abstract

Human fallibility, unpredictable operational environments, and the heterogeneity (and corresponding resource constraints) of hardware devices are driving in the need for software to be able to adapt as seen in the Internet of Things or national telecommunication networks. Unfortunately, mainstream programming languages do not readily allow a software component to sense and respond to its operating environment, by discovering, replacing, and communicating with other software components that are not part of the original system design, while maintaining static correctness guarantees. In particular, if a new component is discovered at runtime, there is no guarantee that its communication behaviour is compatible with existing components.

We address this problem by using multiparty session types with explicit connection actions, a type formalism used to model distributed communication protocols. By associating session types with software components, the discovery process can check protocol compatibility and, when required, correctly replace components. Moreover, the use of session types throughout the software system design guarantees the correctness of all communication, whether or not it is adaptive.

We present the design and implementation of Ensembles, the first actor-based language with adaptive features and a static session type system. We apply it to a case study based on an adaptive DNS server. Finally, we formalise the type system of Ensembles and prove the safety of well-typed programs, making essential use of recent advances in non-classical multiparty session types.

1 Introduction

The era of single monolithic stand-alone computers has long been replaced by a landscape of heterogeneous and distributed computers and software applications. Embracing the current landscape, technologies such as the IoT [57], self-driving cars [56], or autonomous networks [7] bring the new challenge of needing to successfully operate in face of ever-changing environments, technologies, devices, and human errors, necessitating the need to adapt.

Here, we define dynamic self-adaptation—hereafter referred to as adaptation—as the ability of a software component to sense and respond to its operating environment, by discovering, replacing, and communicating with other software components at runtime that are not part of the original system design [6, 53]. There are many examples of adaptive systems, as well as the mechanisms of adaptation they leverage, such as discovery [37], modularisation [26], and...
dynamic code loading and migration [12, 23]. Commercially, Steam’s in-home streaming system\(^2\) enables video games to dynamically transfer their input/output across a range of devices. Academically, RE\(^\times\) [51] enables software to self-assemble predefined components, using machine learning to reconfigure the software in response to environmental changes.

Despite strong interest in adaption and substantial work on the mechanisms of adaptation, current programming languages either lack the capabilities to ensure that adaptation can be achieved safely and correctly, or they check correctness dynamically, resulting in runtime overheads which may not be acceptable for resource-constrained devices.

Specifically, if an adaptive system discovers new software components at runtime, these components must interact with the system in a purposeful manner. In concurrent and distributed systems, such interaction goes beyond a simple function call / return expressed with standard types and type systems: interaction involves complex communication protocols that constrain the sequence and type of data exchanged. For example, knowing that two components communicate integers and strings does not describe if or when they will be sent or received. In spite of growing interest in the topic, for example, the recent formation of the United Nations group considering creative adaptation\(^3\), mainstream programming languages do not support the specification and verification of communication protocols in concurrent and distributed systems. In turn, errors are discovered late in the development process and potentially after deployment.

Even where all components are known statically, communication safety cannot be guaranteed; as an example, the RE\(^\times\) system’s programming language specifies sequential call / return interfaces for components, but not communication protocols for concurrent components. The adaptation in the Steam in-home streaming system is even more limited, being restricted to detection of input/output devices from a set of compatible possibilities. In both cases, the adaptive aspects of the software have been defined and designed ahead of time, as opposed to being composed on-demand at runtime, leaving no scope for extending the system via runtime discovery and replacement.

This situation brings us to a key research question:

**RQ:** Can a programming language support static (compile-time) verification of safe runtime dynamic self-adaptation, i.e., discovery, replacement and communication?

The problem of static verification of safe communication is addressed by (multiparty) session types [29–31]. Multiparty session types (MPSTs) are a type formalism used to specify the type, direction and sequence of communication actions between two or more software components. As well as providing formal communication safety guarantees, session types offer a mechanism by which developers can guarantee that software conforms to predefined communication protocols, rather than risking costly errors manifesting themselves at runtime. However, until now, neither session type theory nor language implementations have supported static verification of runtime discovery and replacement of software.

There is already some work in the literature on adaptation and session types, but it does not answer our research question. We discuss related work in §6, but in brief, the state-of-the-art has some combination of the following limitations: theory for a formal model such as the \(\pi\)-calculus [11, 13, 18, 19], rather than a real-world programming language; omission of some aspects of adaptation, such as runtime discovery [32]; or verification by runtime monitoring [21, 48, 49], as opposed to static checking.

\(^2\) http://store.steampowered.com/streaming/  
\(^3\) https://www.itu.int/en/ITU-T/focusgroups/an/Pages/default.aspx
As an answer to our research question, we implement EnsembleS, the first actor language with support for MPSTs to provide (compile-time) verification of safe dynamic runtime adaptation, i.e., software discovery, replacement, and communication. We formalise our language and type system, which is the first formalisation of software discovery, replacement and communication in MPST-based actors.

Key to our approach is the use of the actor paradigm [27], for its process addressability, modularity, and explicit message passing, alongside explicit connection actions [32] in multiparty session types, which allow discovered actors to be invited into a session.

Paper contributions and structure. Our specific contributions are the following.

1. **EnsembleS and its compiler** (§ 3): we present an actor language, EnsembleS, which supports safe adaptable applications using MPSTs. Our framework supports:
   - MPST specifications, both standard and using explicit connection actions (§ 3.3);
   - MPSTs to provide guarantees of protocol behaviour compliance in runtime software discovery (§ 3.4);
   - automatic generation of application code from MPSTs, separating the protocol design from application implementation (§ 3.2)

2. **An adaptive DNS case study** (§ 4): using MPSTs and runtime discovery to show safe dynamic self-adaptation can be achieved in a non-trivial software service

3. **A core calculus for EnsembleS** (§ 5): we formalise the statics and dynamics of EnsembleS. We prove type preservation (Thm. 10) and progress (Thm. 19), which ensure safety and deadlock-freedom for well-typed programs.

The core calculus makes several technical contributions: it is the first actor-based calculus with statically-checked MPSTs; it is the first calculus to provide a language design and semantics for MPSTs with explicit connection actions; and it is the first to integrate exception handling with MPSTs in a functional core language, which has previously only been studied in the binary setting. We make essential use of non-classical multiparty session types [54].

### 2 Multiparty Session Types

Multiparty session types [31] are a type formalism used to describe communication protocols in concurrent and distributed systems. A MPST describes communication among multiple software components or participants, by specifying the type and the direction of data exchanged, which is given as a sequence of send and receive actions.
Multiparty Session Types for Safe Runtime Adaptation in an Actor Language

We first introduce MPSTs (formalised in §5) via Scribble [58], a specification language for communicating protocols based on the theory of multiparty session types. We start with a global type, which describes the interactions among all communicating participants. Using the Scribble tool, a global protocol can be validated, guaranteeing its correctness, and then projected for each participant. Projection returns a local type, which describes communication actions from the viewpoint of that participant.

**Bookstore example.** To illustrate MPSTs we will present the classic Bookstore (also known as Two-Buyer) example, written in Scribble. Consider three communicating participants, two buyers Buy1 and Buy2, and one seller Sell. These are the roles in our Bookstore example, given in Fig. 1 (left). Buy1 wishes to buy a book from Sell; they send the title of the book of type string to Sell (line 3). Next, Sell sends the price of the book of type int to Buy1 (line 4). At this stage, Buy1 invites Buy2 to share the cost of the book, by sending them a quote of type int that Buy2 should pay (line 5). It is Buy2’s internal choice (line 6) to either agree (line 7), or quit the protocol (line 11). After agreement, both Buy1 and Buy2 transfer their quote to Sell (lines 8 and 9, respectively).

Projecting the Bookstore global protocol into each of the communicating participants Sell, Buy1, and Buy2, returns their local protocols, respectively Bookstore_Sell, Bookstore_Buy1 and Bookstore_Buy2. Fig. 1 shows the local protocol for Sell; we omit Buy1 and Buy2 as they are similar. Note that the local protocol only includes actions relevant to Sell.

**Explicit connection actions.** The Bookstore protocol assumes that all roles are connected at the start of the session. This is undesirable when a participant is only needed for part of a session, or the identity of a participant depends on data exchanged in the protocol.

Consider Figure 2, which details the protocol for an online shopping service, inspired by the travel agency protocol detailed by Hu and Yoshida [32]. The protocol is organised as three subprotocols: OnlineStore, the entry-point; Browse, where the customer repeatedly requests quotes for items; and Deliver, where the store requests delivery from a courier.

In contrast to Bookstore, each connection must be established explicitly (note that connect replaces from when initiating a connection). The customer begins by logging onto the store, and proceeds to browse by sending an item name and receiving a price. After receiving the price, the customer can continue browsing by recursively re-invoking the Browse protocol; exit the protocol by sending a quit message and disconnecting; or request a delivery by invoking the Deliver protocol. To request a delivery, the customer sends their address to the store, which connects to a courier and forwards the address. The courier sends the store a tracking number, which is forwarded to the customer.
Note in particular that Courier is only involved in the Deliver subprotocol. The store can therefore choose which courier to use based on, for example, the weight of the item or the customer’s location. Furthermore, it is not necessary to involve the courier if the customer does not choose to make a purchase.

3 EnsembleS: an Actor Language for Runtime Adaptation

To explore the communication safety guarantees provided by session types in adaptable applications, we present EnsembleS, a new actor-based language, based on Ensemble [24, 25]. EnsembleS actors are addressable, single-threaded entities with share-nothing semantics, and communicate via message passing. However, differently from the classic definition of the actor model [1, 28], the communication model in EnsembleS is channel-based. EnsembleS supports both static and dynamic topologies:

- **Static Topologies**: All participants are present at the start of the session and remain involved for the duration of the session. This is based on traditional MPSTs [31].

- **Dynamic Topologies**: Participants can connect and disconnect during a session. This builds on the more recent idea of explicit connection actions [32].

3.1 EnsembleS: basic language features

An EnsembleS actor has its own private state and a single thread of control expressed as a behaviour clause. The code within this clause is repeated until explicitly told to stop. Every actor executes within a stage, which represents a memory space; many stages may exist per physical machine. EnsembleS supports reference-counted garbage collection and exceptions.

Actors have share-nothing semantics—i.e., they share no state. They communicate via message passing along half-duplex, simply-typed channels.

Fig.3 shows a simple EnsembleS program which defines, instantiates and connects two actors, one of which sends linearly increasing values to the other. The program defines two interfaces Isnd and Ircv, declaring an output and input channel respectively. The boot clause (lines 19–23) is executed first and creates an instance of each actor (lines 20–21), using the appropriate constructor (lines 7 and 13, respectively). This creates and begins executing new threads for each actor, which follow the logic of the relevant behaviour clause. Next, the boot clause binds the actor’s channels together (line 22, discussed later in §3.3). Once bound, the sender actor sends the contents of value on its channel, increments it, and goes back to the beginning of its behaviour loop (lines 8–11). Conversely, the receiver actor waits for a message, and once received, binds the message to data, displays it, and returns to the top of its behaviour loop (lines 14–18).

EnsembleS applications are compiled to Java source code, and then to custom Java class files for use with a custom VM [10]. These applications can be executed on desktop, parallel accelerators (e.g. GPUs), Raspberry Pi, Lego NXT, and Tmote Sky hardware platforms, and

```java
3 type Isnd is interface(out integer output)
4 type Ircv is interface(in integer input)
5
6 stage home { 7 actor sender presents Isnd {
8     value = 1;
9     constructor() {} 10     behaviour { 11     send value on output;
12
13 actor receiver presents Ircv { 14     constructor() {} 15     behaviour { 16     receive data from input;
17     printString("received: "); 18
19 boot { 20 s = new sender(); 21 r = new receiver(); 22 establish topology(s, r);
23 } 24 }
```
use a range of networking technologies. Additionally, EnsembleS inherits Ensemble’s native support for runtime software adaptation actions [25]:

- **Discover**: The ability to locate an arbitrary actor or stage reference at runtime, given an interface and query.
- **Install**: Given an actor type, the ability to spawn it at a specified stage.
- **Migrate**: The ability for an executing actor to move to another stage and continue execution.
- **Replace**: The ability to replace an executing actor A by a new instantiation of actor B, the latter continuing at the same stage as A, if A and B have the same interface.
- **Interact**: Given an actor reference (either spawned, discovered, or communicated), the ability to connect to its channels at runtime and then communicate.

We focus on the underlined actions and apply session types to guarantee communication safety. The reason for this choice is that discover, replace and interact are actions that modify how actors operate, whereas the other actions, install and migrate, affect where actors operate, but not their behaviour.

### 3.2 Session types in EnsembleS

A session type in EnsembleS represents a communication protocol for an actor, i.e., a local protocol (or local session type) validated and projected from a global session type.

We extend the StMungo [40, 41] tool to generate EnsembleS template code that supports session types. Fig. 4 shows an overview of the actor template code generation from a global session type, and Fig. 5 shows an example of the generated code.

First, a developer defines a global session type in Scribble [58] (Fig. 4, first stage). The Scribble tool checks that the protocol is well-formed and valid according to MPST theory and projects the global protocol into local protocols for each participant (Fig. 4, second stage). For each local protocol, the StMungo tool produces (Fig. 4, third stage) i) the session type, ii) the interface and type definitions, and iii) the actor template. The generated code is parsed by the EnsembleS compiler, producing executable code (Fig. 4, fourth stage).

Let us now look at the **Buy1** local protocol, given in Fig. 1. Following the code generation process in Fig. 4, the EnsembleS template items i), ii) and iii) for **Buy1** correspond respectively to the code blocks starting in lines 3, 14, and 24 in Fig. 5.

The **Buy1** local protocol is translated as an EnsembleS session type in Fig. 5 (lines 3–12). It shows a sequence of send and receive actions (lines 4–6), followed by a choice at **Buy2** (lines 7–12), which determines the next set of communication actions.

Following session type specifications, EnsembleS channels define both the payload type and the session that this channel expects to interact with (lines 14–21, Fig. 5). The EnsembleS compiler uses this information to ensure that the session of each channel matches the session associated with the actor it is connected to.

An actor may follow a session type (line 24, Fig. 5). This tells the EnsembleS compiler that the logic within the behaviour clause of that actor must follow the communication protocol defined in the session.
It is important to note that the code generation in Fig. 4 is optional and the Ensembles typechecker is independent of this process.

### 3.3 Channel connections: static and dynamic

If an actor follows a session type, then its channel connections must be 1-1. This is the standard linearity requirement for session types: if there are multiple senders on one channel, then their messages can interfere and it is not possible to statically check that the session is followed correctly. Ensembles avoids this problem by using a single channel for each message type between each pair of participants. For example, in Fig. 1, each of the three actors communicates strings and integers with both of the other actors. Because channels are unidirectional, each actor therefore has 8 channels: 2 to send strings and 2 to send integers to both other actors, and similarly 4 channels for receiving.

**Static connections.** When using session types with static topologies, and all actors in the session are known from the beginning of the application, Ensembles provides the topology keyword to create the connections between the specified session actors (line 22, Fig. 3; line 51, Fig. 5). As there is a channel for each message type between each pair of actors, the topology is uniquely determined. A compile-time error is generated if the number of actors specified is insufficient, if the sessions that they follow do not compose, or if two or more actors follow the same session.

**Dynamic connections.** Ensembles supports reconfigurable channels and dynamic connections, via link and unlink statements. The link statement takes two references to actors which follow sessions (line 6, Fig. 6). It connects all of the channels of the two specified actors such that the channel and actor’s sessions match. If the sessions are incompatible, or if the channels are already bound, a compile-time error is generated. The unlink statement accepts a session type and disconnects all channels owned by the encompassing actor which have been defined with the specified session type (line 9, Fig. 6).
Figure 7 Session type-based replacement

3.4 Adaptation via discovery and replacement

ENSEMBLES supports runtime discovery of local or remote actor instances. As an example, in a sensor network, it may be desirable to connect to a sensor which has a battery level above a certain threshold. The ENSEMBLES query language allows the user to define a query on non-functional properties (such as battery level, signal strength, or name), as well as the channels exposed by an actor’s interface. This ensures that any discovered actor has the correct number and type of channels, and satisfies user’s preferences. To ensure that the discovered actor also obeys a declared protocol, ENSEMBLES uses session types in the discovery process. The green box in Fig. 6 shows how a session is used in the actor discovery process, and the yellow box shows how such actors are connected together. Runtime discovery does not appear in the session because it does not affect the behaviour of an application directly.

ENSEMBLES also supports the replacement of executing actors, much like the hot-code swapping in Erlang [12]. The new actor must present the same interface as it takes over the channels of the actor being replaced at the location it was executing. Replacement happens at the beginning of an actor’s behaviour loop. Replacement has many uses, such as updating, changing, or extending some of the functionalities of existing software, and is particularly useful in embedded systems [33, 34]. The existing and new actors must follow the same session type, guaranteeing that replacement will not break existing actor interactions.

Fig. 7 shows an example of a main actor searching for actors of type slowA (line 32), and replacing them with new actors of type fastA (lines 35–37). slowA actors are located by defining a query (line 26) over user-defined properties, which are published (lines 16–18). The discovery process is the same as above, but now the discovered actors are used for replacement rather than just communication.
To illustrate the use of session types for adaptive programming, we consider a real-world case study: the domain name system (DNS). DNS is a hierarchical, globally distributed translation system that converts an internet host name (domain name) into its corresponding numerical Internet Protocol (IP) address [44].

The process begins by transmitting a domain name to one of many well-known root servers. This server either rejects bad requests, or provides the information to contact a zone server. The zone server may know the IP address of the domain name; if not it refers the request to another zone server. This process continues until either the IP address is returned, or the name cannot be found.

To develop an adaptive DNS example, we assume no a priori information about server location, and instead use explicit discovery to find root and zone servers based on session types and server properties. We use an existing Scribble description of DNS as a starting point [21]. To illustrate adaptation we focus on the client who is querying DNS.
Fig. 8 shows the session type for the client actor which asks DNS to resolve a domain name. The client first asks for a root server (lines 2–3), and then either is informed that the request is invalid (lines 24–25) or recursively queries zone servers (lines 7–22) until the IP address is found (lines 19–20), or an error is reported (lines 16–17). Based on this session, StMungo generates EnsembleS types and interface definitions and a skeleton actor. Minimally completing the generated skeleton produces the code in Fig. 9.

In this example, discovery is used to locate the root server (lines 21–25, in Fig. 9) and the zone server (line 37). In each case, the session for the relevant server is provided to ensure that the discovered actor follows the expected protocol. When either server is located, the client links with it (lines 27 and 39), enabling communication. When communication with the server is no longer required, the client unlinks explicitly (lines 34, 47, 51, 55, 62).

Although explicit discovery is used at the language level, there is nothing to prevent the implementation of discovery from caching the addresses of the root and zone servers. This does not affect the use of sessions in discovery or the safety they provide, as the type-based guarantees are still enforced. However, this would potentially improve performance of the system. Additionally, if a cached entry becomes stale, the full discovery process can again be used without code modification or degradation in trust.

A version of DNS which uses discovery allows the system to become more flexible and resilient to changing operational conditions, such as topology changes in the servers and their data. The use of sessions ensures compatibility with the discovered actors.

### 5 A Core Calculus for EnsembleS

In this section, we provide a formal characterisation of EnsembleS. In doing so, we show that our integration of adaptation with multiparty session types is safe, allowing adaptation while precluding communication mismatches.

**Relationship to implementation.** Although EnsembleS builds upon Ensemble by using the Mungo / StMungo toolchain, our core calculus aims to distil the essence of the interplay between adaptation and session-typed communication with explicit connection actions.

We concentrate on a functional core calculus rather than an imperative one: imperative variable binding serves only to clutter the formalism, and our fine-grain call-by-value representation can be thought of as an intermediate language.

Interfaces and unidirectional, simply-typed channels in EnsembleS are an implementation artifact: sending on a channel whose type changes is equivalent to sending on multiple channels with different types. Moreover, following theoretical accounts of multiparty session types [14, 31, 32], instead of having send and receive (resp. connect and accept) operations followed by branching (as done in Mungo and StMungo), we have unified send and receive constructs which communicate a label along with the message payload.

Since session typing is the interesting part of discovery, we omit properties and queries from the formalism; their inclusion is routine.

Finally, since they are important for adaptation and more interesting technically, we concentrate on dynamic topologies with explicit connection actions rather than static topologies.

#### 5.1 Syntax

**Definitions.** Figure 10 shows the syntax of Core EnsembleS terms and types. We let $u$ range over actor class names, and $D$ range over definitions; each definition $\textbf{actor } u \textbf{ follows } S \{ M \}$ specifies the class name, the session type followed by the actor, and the actor’s behaviour.
Syntax of Types and Terms

- Actor class names: \( u \)
- Actor definitions: \( D ::= \text{actor } u \text{ follows } S \{M\} \)
- Recursion Labels: \( l \)
- Behaviours: \( \kappa ::= M | \text{stop} \)
- Types: \( A, B ::= \text{Pid}(S) | 1 \)
- Values: \( V, W ::= x | () \)
- Actions: \( L ::= \text{return } V | \text{continue } l | \text{raise } \)
  \( \text{new } u | \text{self } | \text{replace } V \text{ with } \kappa | \text{discover } S \)
  \( \text{connect } \ell(V) \text{ to } W \text{ as } p | \text{accept from } p \{\ell(x_i) \mapsto M_i\}_i \)
  \( \text{send } \ell(V) \text{ to } p | \text{receive from } p \{\ell(x_i) \mapsto M_i\}_i \)
  \( \text{wait } p | \text{disconnect from } p \)
- Computations: \( M, N ::= \text{let } x \leftarrow M \text{ in } N | \text{try } L \text{ catch } M | l :: M | L \)

Syntax of Session Types

- Session Actions: \( \alpha, \beta ::= p!f(A) | p!f(A) | p?f(A) | p?f(A) | \# p \)
- Session Types: \( S, T, U ::= \Sigma_{\ell \in I}(\alpha_i, S_i) | \mu X.S | X | \# \downarrow p | \text{end} \)
- Communication Actions: \( \dagger ::= ! | ? \)
- Disconnection Actions: \( \ddagger ::= \# \uparrow | \# \downarrow \)

Figure 10 Syntax

Much like a class table in Featherweight Java [36], we assume a fixed mapping from class names to definitions.

Values. Since our calculus is inherently effectful, it is convenient to adopt a presentation in the style of fine-grain call-by-value [42]. In this setting, we have an explicit static stratification of values and computations, and an explicit evaluation order similar to A-normal form [20]. Values \( V, W \) describe data that has been computed, and for the sake of simplicity, consist of variables and the unit value. Other base values (such as integers or booleans) can be encoded or added straightforwardly.

Computations. The \text{let } x \leftarrow M \text{ in } N \text{ construct evaluates } M, \text{ binding its result to } x \text{ in } N. \text{ The calculus supports exception handling over a single }\text{action } L \text{ using } \text{try } L \text{ catch } M, \text{ where } M \text{ is evaluated if } L \text{ raises an exception, and labelled recursion using } l :: M, \text{ stating that inside term } M, \text{ a process can recurse to label } l \text{ using } \text{continue } l. \text{ Actions } L \text{ denote the basic steps of a computation. The }\text{return } V \text{ construct denotes a value.}

Concurrency and adaptation constructs. The \text{new } u \text{ construct spawns a new actor of class } u \text{ and returns its PID. The }\text{self } \text{ construct returns the current actor’s PID. An actor can replace the behaviour of itself or another actor } V \text{ using } \text{replace } V \text{ with } \kappa. \text{ An actor can discover other actors following a session type } S \text{ using the }\text{discover } S \text{ construct, which returns the PID of the discovered actor.}

Session communication constructs. An actor can connect to an actor } W \text{ playing role } p \text{ using } \text{connect } \ell(V) \text{ to } W \text{ as } p, \text{ sending a message with label } \ell \text{ and payload } V. \text{ An actor can accept a connection from another actor playing role } p \text{ using } \text{accept from } p \{\ell(x_i) \mapsto M_i\}_i, \text{ which allows an actor to receive a choice of messages; given a message with label } \ell_j, \text{ the payload is bound to } x_j \text{ in the continuation } N_j. \text{ Once connected, an actor can communicate using the }\text{send } \text{ and }\text{receive } \text{ constructs. An actor can disconnect from } p \text{ using } \text{disconnect from } p, \text{ and await the disconnection of an actor } p \text{ using } \text{wait } p.\)

Types. Types, ranged over by \( A, B \), include the unit type \( 1 \) and process IDs \text{Pid}(S); \text{ the parameter } S \text{ refers to the statically-known initial session type of the actor (i.e., the session type declared in the }\text{follows } \text{ clause of a definition). Unlike in channel-based session-typed
systems, process IDs themselves need not be linear: any number of actors can have a reference to another actor, but each actor may only be in a single session at a time. PIDs can be passed as payloads in session communications.

Session types. Session types are ranged over by $S, T, U$ and follow the formulation of Hu and Yoshida [32]. A session type can be a choice of actions, written $\Sigma_{i \in I}(\alpha_i . S_i)$, a recursive session type $\mu X.S$ binding recursion variable $X$ in continuation $S$, a recursion variable $X$, a disconnection action $\# p$, or the finished session $\text{end}$. The syntax of session types is more liberal than traditional ‘directed’ presentations in order to allow output-directed choices to send or connect to different roles.

Session actions $\alpha$ involve sending ($!$), receiving ($?$), connecting ($!!$), or accepting ($???$) a message $\ell(A)$ with label $\ell$ and type $A$; or awaiting another participant’s disconnection ($\# \uparrow$).

As well as disallowing self-communication, following Hu and Yoshida [32], we require the following syntactic restrictions on session types:

\begin{definition}[Syntactic validity] A choice type $S = \Sigma_{i \in I}(\alpha_i . S_i)$ is syntactically valid if:
\begin{enumerate}
\item it is an output choice, i.e., each $\alpha_i$ is a send or connection action; or
\item it is a directed input choice, i.e., $S = \Sigma_{i \in I}(p?\ell_i(A_i).S_i)$ or $S = \Sigma_{i \in I}(p??\ell_i(A_i).S_i)$; or
\item the choice consists of single wait action $\# \uparrow p . S$.
\end{enumerate}
\end{definition}

In the remainder of the paper, we assume that all session types are syntactically valid.

Session correlation. The most general form of explicit connection actions allows a participant to leave and re-join a session, or accept connections from multiple different participants. Such generality comes at a cost, since care must be taken to ensure that the same participant plays the role throughout the session.

To address this session correlation issue, Hu and Yoshida [32] propose two solutions: either augment global types with type-level assertions and check conformance dynamically, or adopt a lightweight syntactic restriction which requires that each local type may contain at most a single accept action as its top-level construct. We opt for the latter, enforcing the constraint as part of our safety property (§5.4.2), and by requiring that $\# \uparrow p$ does not have a continuation. As Hu and Yoshida [32] show, this design still supports the most common use cases of explicit connection actions.

Global types. Traditional MPST works [14, 31] use global types to describe the interactions between participants at a global level, which are then projected into local types; projectability ensures safety and deadlock-freedom.

Since we are using explicit connection actions, traditional approaches are insufficiently flexible as they do not account for certain roles being present in certain branches but not others. Following Scalas et al. [55] and subsequently non-classical MPSTs [54], we instead formulate our typing rules and safety properties using collections of local types.

It is, however, still convenient to write a global type and have local types computed programatically. Global types are defined as follows:

\[
\text{Global Actions } \pi ::= p \rightarrow q : \ell(A) \mid p \rightarrow q : \ell(A) \mid p \# q
\]

\[
\text{Global Types } G ::= \Sigma_{i \in I}(\pi_i . G_i) \mid \mu X.G \mid X \mid \text{end}
\]

Global actions $\pi$ describe interactions between participants: $p \rightarrow q : \ell(A)$ states that role $p$ sends a message with label $\ell$ and payload type $A$ to $q$. Similarly, $p \rightarrow q : \ell(A)$ states that $p$ connects to $q$ by sending a message with label $\ell$ and payload type $A$. The disconnection action $p \# q$ states that role $p$ disconnects from role $q$.

We can write the OnlineStore example from §2 as follows:
Customer → Store : login(String).μBrowse.
Customer → Store : item(String). Store → Customer : price(Int). Browse +
Customer → Store : address(String). Store → Courier : deliver(String).
Store#Customer . end +
Customer → Store : quit(1). Store#Customer . end

Although projectability in our setting does not necessarily guarantee safety and deadlock-
freedom, we show a projection algorithm, adapted from that of Hu and Yoshida [32], in
Appendix A. The resulting local types can then be checked for safety (§5.4.2).

Protocols and Programs. Terms do not live in isolation; they refer to a set of protocols,
and evaluate in the context of an actor. A protocol maps role names to local session types.

Definition 2 (Protocol). A protocol is a set \( \{ p_i : S_i \} \), mapping role names to session types.

As an example, consider the protocol for the online shop example:

\[
\begin{align*}
\text{Customer : Store!login(String).μBrowse.} \\
\text{Store!item(String). Store!price(Int). Browse} \\
\text{Store!address(String). Store!ref(Int). Store#end} \\
\text{Customer!logout(String).μBrowse.} \\
\text{Customer!item(String). Customer!price(Int). Browse} \\
\text{Customer!address(String). Courier!deliver(String). Courier!ref(Int). Store#end} \\
\text{Courier : Store!deliver(String). Store!ref(Int). Store#end}
\end{align*}
\]

We can now consider an implementation of a Store actor, which uses discovery to find a
courier. We write receive \( \ell(x) \) from \( p : M \) and accept \( \ell(x) \) from \( p : M \) as syntactic sugar for
receive from \( p \) \{ \( \ell(x) \rightarrow M \) \} and accept from \( p \) \{ \( \ell(x) \rightarrow M \) \} respectively, and write \( M : N \) as
syntactic sugar for let \( x \leftarrow M \) in \( N \) for a fresh variable \( x \). We assume the existence of a function
lookupPrice, and define CourierType as Store!? deliver(String). Store!ref(Int). Store#end.

actor Store follows ty(Store) {
  accept login(credits) from Customer;
  Browse ::
  receive from Customer {
    item(name) ⇒
      send price(lookupPrice(name)) to Customer;
      continue Browse
    address(addr) ⇒
      let pid ⇥ discover CourierType in
      connect deliver(addr) to pid as Courier;
      receive ref(r) from Courier;
      wait Courier;
      send ref(r) to Customer;
      disconnect from Customer
    quit() ⇒ disconnect from Customer
  }
}

A program consists of actor definitions, protocol definitions, and the ‘boot’ clause to be
run in order to set up initial actor communication.

Definition 3 (Program). An Ensembles program is a 3-tuple \( (\bar{D}, \bar{P}, M) \) of a set of
definitions, protocols, and an initial term to be evaluated.
<table>
<thead>
<tr>
<th>Definition typing</th>
<th>Value typing</th>
<th>Behaviour typing</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash D$</td>
<td>$\Gamma \vdash v : A$</td>
<td>${S} \Gamma \vdash \kappa$</td>
</tr>
</tbody>
</table>

Typing rules for computations:

**Functional Rules**

- $T\text{-Def}$: 
  \[
  \vdash \text{actor } u \text{ follows } S \{M\} \quad \Gamma \vdash x : A \quad \Gamma \vdash () : I \quad \{S\} \Gamma \vdash \text{stop} \quad \{S\} \Gamma \vdash M
  \]

**Actors / Adaptation Rules**

- $T\text{-New}$: 
  \[
  \{T\} \Gamma \vdash \text{sessionType}(u) = U \quad \{T\} \Gamma \vdash \text{new } u : \text{Pid}(U) \vdash S
  \]

- $T\text{-Self}$: 
  \[
  \{T\} \Gamma \vdash \text{self} : \text{Pid}(T) \vdash S
  \]

- $T\text{-Discover}$: 
  \[
  \{T\} \Gamma \vdash \text{discover } U : \text{Pid}(U) \vdash S
  \]

- $T\text{-Replace}$: 
  \[
  \Gamma \vdash V : \text{Pid}(U) \quad \{U\} \Gamma \vdash \kappa \quad \{T\} \Gamma \vdash S \vdash \text{replace } U \text{ with } \kappa \Gamma \vdash S
  \]

- $T\text{-Body}$: 
  \[
  \{S\} \Gamma \vdash M : A \vdash S'
  \]

**Figure 11** Typing rules (1)

In the context of a program, we write $\text{ty}(p)$ to refer to the session type associated with role $p$ as defined by the set of protocols. Given an actor definition $\text{actor } u \text{ follows } S \{M\}$, we define $\text{sessionType}(u) = S$ and $\text{behaviour}(u) = M$.

### 5.2 Typing rules

Figures 11 and 12 show the typing rules for ENSEMBLES. Value typing, with judgement $\Gamma \vdash V : A$, states that under environment $\Gamma$, value $V$ has type $A$. Judgement $\vdash D$ states that an actor definition $\text{actor } u \text{ follows } S \{M\}$ is well-typed if its body is typable under, and fully consumes, its statically-defined session type $S$. The behaviour typing judgement $\{S\} \Gamma \vdash \kappa$ states that given static session type $S$, behaviour $\kappa$ is well-typed under $\Gamma$. Specifically, $\text{stop}$ is always well-typed, $M$ is well-typed if it is typable under, and fully consumes, $S$.

#### 5.2.1 Term typing

The typing judgement for terms $\{T\} \Gamma \vdash M : A \vdash S'$ reads “in an actor following $T$, under typing environment $\Gamma$ and with current session type $S$, term $M$ has type $A$ and updates the session type to $S’$.” Note that the term typing judgement, reminiscent of parameterised monads [3], contains a session precondition $S$ and may perform some session communication actions to arrive at postcondition $S’$.

**Functional rules.** Rule $T\text{-Let}$ is a sequencing operation: given a construct $\text{let } x \leftarrow M \text{ in } N$ where $M$ has pre-condition $S$ and post-condition $S'$, and where $N$ has pre-condition $S'$ and post-condition $S''$, the overall construct has pre-condition $S$ and post-condition $S''$. 

---

[452]

[453]

[454]

[455]

[456]

[457]

[458]

[459]

[460]

[461]

[462]

[463]

[464]

[465]

[466]

[467]

[468]

[469]

[470]
We take an equi-recursive view of session types, identifying recursive sessions with their
ensures that the pre-condition must match the label stored in the environment,
Exception handling rules.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Session communication rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T)-\textsc{Raise} [{T} \Gamma \mid S \triangleright \text{raise}\cdot A \triangleleft S']</td>
<td>( T)-\textsc{Connect} [{T} \Gamma \mid \sum_{i \in I} (\alpha_i \cdot S_i) \triangleright \text{connect} \ell_j(V) \to W] as ( p_j : 1 \triangleleft S_j']</td>
</tr>
<tr>
<td>( T)-\textsc{Try} [{T} \Gamma \mid S \triangleright L : A \triangleleft S']</td>
<td>( T)-\textsc{Send} [{T} \Gamma \mid \sum_{i \in I} (\alpha_i \cdot S_i) \triangleright \text{send} \ell_j(V) \to p_j : 1 \triangleleft S_j']</td>
</tr>
<tr>
<td>( T)-\textsc{Accept} [{T} \Gamma \mid \sum_{i \in I} (q?!l_i(B_i) \cdot S_i) \triangleright \text{accept from} q {l_i(x_i) \mapsto M_i}_{i \in I} : A \triangleleft S]</td>
<td>( T)-\textsc{Recv} [{T} \Gamma \mid \sum_{i \in I} (q?!l_i(B_i) \cdot S_i) \triangleright \text{receive from} q {l_i(x_i) \mapsto M_i}_{i \in I} : A \triangleleft S]</td>
</tr>
<tr>
<td>( T)-\textsc{Wait} [{T} \Gamma \mid #q \cdot S \triangleright \text{wait} q : 1 \triangleleft S]</td>
<td>( T)-\textsc{Disconnect} [{T} \Gamma \mid #q \triangleright \text{disconnect from} q : 1 \triangleleft \text{end}]</td>
</tr>
</tbody>
</table>

\(\textbf{Figure 12} \) Typing rules (2)

Following Kouzapas et al. [40], we formalise recursion through annotated expressions:
term \( l : M \) states that \( M \) is an expression which can loop to \( l \) by evaluating \( \text{continue } l \).
We take an equi-recursive view of session types, identifying recursive sessions with their
unfolding \( \mu X.S = S\{\mu X.S/X\} \), and assume that recursion is guarded. Rule \( T\)-\textsc{Rec} extends
the typing environment with a recursion label defined at the current session type. Rule
\( T\)-\textsc{Continue} ensures that the pre-condition must match the label stored in the environment,
but has arbitrary type and any post-condition since the return type and post-condition
depend on the enclosing loop’s base case.

**Actor and adaptation rules.** Rule \( T\)-\textsc{New} states that creating an actor of class \( u \) returns
a PID parameterised by the session type declared in the class of \( u \). Rule \( T\)-\textsc{Self} retrieves a
PID for the current actor, parameterised by the statically-defined session type of the local
actor. Rule \( T\)-\textsc{Discover} states \( \text{discover } U \) returns a PID of type \( \text{Pid}(U) \). Finally, given a
behaviour \( \kappa \text{ typable under a static session type } U \), and a process ID with the matching static
type \( \text{Pid}(U) \), \( T\)-\textsc{Replace} allows replacement, and returns the unit type.

**Exception handling rules.** Figure 12 shows the rules for exception handling and session
communication. \( T\)-\textsc{Raise} denotes raising an exception; since it does not return, it can
have an arbitrary return type and post-condition. Rule \( T\)-\textsc{Try} types an exception handler
\( \text{try } L \text{ catch } M \) which acts over a single action \( L \). If \( L \) raises an exception, then \( M \) is raised
instead. Since \( L \) only scopes over a single action, the \( \text{try} \) and \( \text{catch} \) clauses have the same
pre- and post-conditions to allow the action to be retried if necessary.

**Session communication rules.** Rule \( T\)-\textsc{Connect} types a term \( \text{connect } \ell_j(V) \to W \) as \( p_j \).
Given the precondition is a choice type containing a branch \( p!!\ell_j(A_j) \cdot S_j' \), and the remote
actor reference is \( W \) of type \( \text{Pid}(S) \), the rule ensures that \( S \) is compatible with the type of
\( p_j \), and ensures that the label and payload are compatible with the session type. The session
We describe the semantics of $\Delta$, which is similar and advances the session type to $\Delta$. For each branch, the result type and postcondition must be typable under an environment extended with $i$. Term reduction follows a deterministic reduction relation on terms, Configuration reduction (1) follows the same pattern.

### 5.3 Operational semantics

We describe the semantics of $\text{ENSEMBLE}$ via a deterministic reduction relation on terms, and a nondeterministic reduction relation on configurations.

#### 5.3.1 Runtime syntax

Figure 13 shows the runtime syntax and the first part of the reduction rules for $\text{ENSEMBLE}$. The type is then advanced to $S'_i$. Rule $\text{T-SEND}$ follows the same pattern.

Given a session type $\Sigma_{i \in I} (p ? x_i (A_i)) \cdot S_i$, rule $\text{T-ACCEPT}$ types term $\text{accept from } p \{ \ell_i (x_i) \mapsto M_i \}_{i \in I}$, enabling an actor to accept connections with messages $\ell_i$, binding the payload $x_i$ in each continuation $M_i$. Like case expressions in functional languages, each continuation must be typable under an environment extended with $x_i : A_i$, under session type $S_i$, and each branch must have the same result type and postcondition. Rule $\text{T-RECV}$ is similar.

Rule $\text{T-WAIT}$ handles waiting for a participant $p$ to disconnect from a session, requiring a pre-condition of $\# \nmid p \cdot S$, returning the unit type and advancing the session type to $S$. Rule $\text{T-DISCONNECT}$ is similar and advances the session type to end.
Whereas static syntax and typing rules describe code that a user would write, runtime syntax arises during evaluation. We introduce two types of runtime name: \( s \) ranges over session names, which are created when a process initiates a session, and \( a \) ranges over actor names, which uniquely identify each actor once it has been spawned by new.

**Configurations.** Configurations, ranged over by \( C, D, \mathcal{E} \), represent the concurrent fragment of the language. Like in the \( \pi \)-calculus [43], name restrictions \((\nu n)C\) bind name \( n \) in \( C, C \parallel D \) denotes \( C \) and \( D \) running in parallel, and the \( \emptyset \) configuration denotes the inactive process.

Actors are represented at runtime as a 4-tuple \( \langle a, M, \sigma, \kappa \rangle \), where \( a \) is the actor’s runtime name; \( M \) is the term currently evaluating; \( \sigma \) is the connection state; and \( \kappa \) is the actor’s current behaviour. A connection state is either disconnected, written \( \bot \), or playing role \( p \) in session \( s \) and connected to roles \( q \), written \( s[p](q) \).

Following the work on Exceptional GV by Fowler et al. [22] and the work on affine sessions by Mostroun and Vasconcelos [46], a zapper thread \( zs[p] \) indicates that participant \( p \) in session \( s \) cannot be used for future communications, for example due to the actor playing the role in the session crashing due to an unhandled exception.

**Runtime typing environments.** Whereas \( \Gamma \) is an unrestricted typing environment used for typing values and configurations, we introduce \( \Delta \) as a linear runtime environment. Runtime environments can contain entries of type \( a : S \), stating that actor \( a \) has statically-defined session type \( S \), and entries of type \( s[p](q):S \), stating that in session \( s \), role \( p \) is connected to roles \( q \) and currently has session type \( S \).

**Evaluation contexts.** Due to our fine-grain call-by-value presentation, evaluation contexts \( E \) allow nesting only in the immediate subterm of a let expression. The top-level frame \( F \) can either be a hole, or a single, top-level exception handler. Pure contexts \( EP \) do not include exception handling frames.

To run a program, we place it in an initial configuration.

**Definition 4 (Initial configuration).** An initial configuration for an EnsembleS program with boot clause \( M \) is of the form \((\nu a)((a, M, \bot, \text{stop}))\).

### 5.3.2 Reduction rules

Term reduction \( \rightarrow_M \) is standard \( \beta \)-reduction, save for E-TryRaise which evaluates the failure continuation in the case of an exception. We consider four subcategories of configuration reduction rules: actor and adaptation rules; session communication rules; exception handling rules; and administrative rules.

**Actor / adaptation rules.** Given a fully-evaluated actor, E-LOOP runs the term specified by the actor’s behaviour. Rule E-New allows actor \( a \) to spawn a new actor of class \( u \) by creating a fresh runtime actor name \( b \) and a new actor process of the form \( (b, M, \bot, M) \) where \( M \) is the behaviour specified by \( u \), returning the process ID \( b \). Rules E-Replacement and E-REPLACESELF handle replacement by changing the behaviour of an actor, returning the unit value to the caller. Rule E-DISCOVER returns the process ID of an actor \( b \) if it has the desired static session type \( S \). Rule E-SELF returns the PID of the local actor.

**Session communication rules.** An actor begins a session by connecting to another actor while disconnected; such a case is handled by rule E-CONNINIT. Suppose we have a disconnected actor \( a \) evaluating a connection statement connect \( \ell_i(V) \) to \( b \) as \( p \), evaluating in parallel with a disconnected actor \( b \) evaluating an accept statement accept from \( p \) \((\ell_i(x_i) \mapsto \)


Configuration reduction (2) \[ \mathcal{C} \rightarrow \mathcal{D} \]

Session reduction rules

**E-CONNINIT**

\[
\begin{align*}
\langle a, E[F[\text{connect } \ell_j(V) \text{ to } b \text{ as } q]], \bot, \kappa_1 \rangle &\parallel \langle b, E'[\text{accept from } p \{ \ell_i(x_i) \mapsto M_i \}_{i \in I}], \bot, \kappa_2 \rangle \red\rightarrow \\
\langle (\nu s)\langle (a, E[\text{return } i]), s[p][q], \kappa_1 \rangle \parallel \langle b, E'[M_j(V/x_j)], s[q][p], \kappa_2 \rangle \rangle &\parallel \langle (\nu s)\langle (a, E[\text{return } i]), s[p][q], \kappa_1 \rangle \parallel \langle b, E'[M_j(V/x_j)], s[q][p], \kappa_2 \rangle \rangle
\end{align*}
\]

**E-CONN**

\[
\begin{align*}
\langle a, E[F[\text{connect } \ell_j(V) \text{ to } b \text{ as } q]], s[p][\tilde{r}], \kappa_1 \rangle &\parallel \langle b, E'[\text{accept from } p \{ \ell_i(x_i) \mapsto M_i \}_{i \in I}], \bot, \kappa_2 \rangle \red\rightarrow \\
\langle (\nu s)\langle (a, E[\text{return } i]), s[p][\tilde{r}, q], \kappa_1 \rangle \parallel \langle b, E'[M_j(V/x_j)], s[q][p], \kappa_2 \rangle \rangle &\parallel \langle (\nu s)\langle (a, E[\text{return } i]), s[p][\tilde{r}, q], \kappa_1 \rangle \parallel \langle b, E'[M_j(V/x_j)], s[q][p], \kappa_2 \rangle \rangle
\end{align*}
\]

**E-CONNFail**

\[
\begin{align*}
\langle a, E[\text{connect } \ell_j(V) \text{ to } b \text{ as } q], s[p][\tilde{r}], \kappa_1 \rangle &\parallel \langle b, E'[\text{raise from } p \{ \ell_i(x_i) \mapsto N_i \}_{i \in I}], \bot, \kappa_2 \rangle \red\rightarrow \\
\langle (\nu s)\langle (a, E[\text{return } i]), s[p][\tilde{r}, q], \kappa_1 \parallel \langle b, E'[\text{raise } i], s[q][p], \kappa_2 \rangle \rangle &\parallel \langle (\nu s)\langle (a, E[\text{return } i]), s[p][\tilde{r}, q], \kappa_1 \parallel \langle b, E'[\text{raise } i], s[q][p], \kappa_2 \rangle \rangle
\end{align*}
\]

**E-Disconn**

\[
\begin{align*}
\langle a, E[F[\text{wait } q]], s[p][\tilde{r}, q], \kappa_1 \rangle &\parallel \langle b, E'[\text{disconnect from } p], s[q][p], \kappa_2 \rangle \red\rightarrow \\
\langle (\nu s)\langle (a, E[\text{return } i]), s[p][\tilde{r}, q], \kappa_1 \rangle \parallel \langle b, E'[\text{return } i], s[q][p], \kappa_2 \rangle \rangle &\parallel \langle (\nu s)\langle (a, E[\text{return } i]), s[p][\tilde{r}, q], \kappa_1 \rangle \parallel \langle b, E'[\text{return } i], s[q][p], \kappa_2 \rangle \rangle
\end{align*}
\]

**E-Comm**

\[
\begin{align*}
\langle a, E[F[\text{send } \ell_j(V) \text{ to } q]], s[p][\tilde{r}], \kappa_1 \rangle &\parallel \langle b, E'[\text{receive from } p \{ \ell_i(x_i) \mapsto M_i \}_{i \in I}], s[q][\tilde{s}], \kappa_2 \rangle \red\rightarrow \\
\langle (\nu s)\langle (a, E[\text{return } i]), s[p][\tilde{r}], \kappa_1 \rangle \parallel \langle b, E'[M_j(V/x_j)], s[q][\tilde{s}], \kappa_2 \rangle \rangle &\parallel \langle (\nu s)\langle (a, E[\text{return } i]), s[p][\tilde{r}], \kappa_1 \rangle \parallel \langle b, E'[M_j(V/x_j)], s[q][\tilde{s}], \kappa_2 \rangle \rangle
\end{align*}
\]

**E-Complete**

\[
\begin{align*}
\langle (\nu s)\langle (a, \text{return } V, s[p][\emptyset]), \kappa_1 \rangle \rangle &\red\rightarrow \langle a, \text{return } V, \bot, \kappa \rangle
\end{align*}
\]

**Figure 14** Operational semantics (2)

---

Exception handling rules. Exception handling rules allow safe session communication in the presence of exceptions. Rule E-ComRAISE states that if an actor is attempting to communicate with a role no longer present due to an exception, then an exception should be raised. We write \( \text{sub}(M) = p \) if \( M \in \{ \text{send } \ell(V) \text{ to } p, \text{receive from } p \{ \ell_i(x_i) \mapsto N_i \}_{i \in I}, \text{wait } p, \text{disconnect from } p \} \). Rule E-FAILS states that if a connected actor encounters an unhandled exception, then a zapper thread will be generated for the current role, the actor will become disconnected, and the current evaluation context will be discarded. Rule E-FAILLoop restarts an actor encountering an unhandled exception.
Exception handling rules

E-CommRaise
\[ \text{subj}(M) = q \]
\[ \langle a, E[\text{raise}], s[p](\tilde{q}), \kappa \| \tilde{s}[q] \rangle \rightarrow \langle a, E[\text{raise}], s[p](\tilde{q}), \kappa \| \tilde{s}[p] \rangle \]

E-FailS
\[ \langle a, E[\text{raise}], s[p](\tilde{q}), \kappa \| \tilde{s}[p] \rangle \rightarrow \langle a, E[\text{raise}], \bot, \kappa \| \tilde{s}[p] \rangle \]

E-FailLoop
\[ \langle a, E[\text{raise}], \bot, M \rangle \rightarrow \langle a, E[\text{raise}], \bot, M \rangle \]

Administrative rules

E-LiftM
\[ M \rightarrow_{\text{M}} M' \]
\[ \langle a, E[M], \sigma, \kappa \rangle \rightarrow \langle a, E[M'], \sigma, \kappa \rangle \]

E-Equiv
\[ C \equiv C' \rightarrow D' \]
\[ D \equiv D' \rightarrow C \parallel D' \]

E-Par
\[ C \rightarrow C' \parallel D \rightarrow C' \parallel D' \]

E-Nu
\[ (\nu n) C \rightarrow (\nu n) D \]

Configuration equivalence
\[ \not\equiv \]
\[ C \parallel D \equiv D \parallel C \]
\[ (\nu s)(\tilde{s}[p_1] \| \cdots \| \tilde{s}[p_n]) \parallel C \equiv C \]

Configuration reduction (3)

Administrative rules. The remaining rules are administrative: E-LiftM allows term reduction inside an actor; E-Equiv allows reduction modulo structural congruence; E-Par allows reduction under parallel composition; and E-Nu allows reduction under name restrictions.

Configuration equivalence. Reduction includes configuration equivalence \( \equiv \), defined as the smallest congruence relation satisfying the axioms in Figure 15. The equivalence rules extend the usual \( \pi \)-calculus structural congruence rules with a ‘garbage collection’ equivalence, which allows us to discard a session where all participants have exited due to an error.

5.4 Metatheory

We now turn our attention to showing that session typing allows runtime adaptation and discovery while precluding communication mismatches and deadlocks.

5.4.1 Runtime typing

To reason about the metatheory, we introduce typing rules for configurations (Fig. 16): the judgement \( \Gamma; \Delta \vdash C \) states that configuration \( C \) is well-typed under term typing environment \( \Gamma \) and runtime environment \( \Delta \).

Rule T-Pid types actor name restriction \( (\nu a)C \) by adding a PID into the term environment, and extending the runtime typing environment \( a : S \); the linearity of the runtime typing environment therefore means that the system must contain precisely one actor with name \( a \).

Session name restrictions \( (\nu s)C \) are typed by T-Session. We follow the formulation of Scalas and Yoshida [54] which types multiparty sessions using a parametric safety property \( \varphi \); we discuss safety properties in more depth in Section 5.4.2. Let \( \Delta' \) be a runtime typing environment containing only mappings of the form \( s[p_i](\tilde{q}_i):S_i \). Assuming \( \Delta \) does not contain any mappings involving session \( s \) and \( \Delta' \) satisfies \( \varphi \), the rule states that \( C \) is typable under typing environment \( \Gamma \) and runtime typing environment \( \Delta, \Delta' \). It is sometimes convenient to annotate session \( \nu \)-binders with their environment, e.g., \( (\nu s : \Delta')C \).
We now prove that reduction preserves typability and thus that actors only perform communication actions specified in their session types. Due to our use of explicit connection actions, classical MPST approaches are too limited for our purposes. Our approach, following that of Sculav and Yoshida [54], is to introduce a labelled transition system (LTS) on local types, and specify a generic safety property based around local type reduction. The property can then refined; in our case, we will later specialise the property in order to prove progress.

### Preservation

We now prove that reduction preserves typability and thus that actors only perform communication actions specified in their session types. Due to our use of explicit connection actions, classical MPST approaches are too limited for our purposes. Our approach, following that of Sculav and Yoshida [54], is to introduce a labelled transition system (LTS) on local types, and specify a generic safety property based around local type reduction. The property can then refined; in our case, we will later specialise the property in order to prove progress.

#### Reduction on runtime typing environments

Figure 17 shows the LTS on runtime typing environments. There are two judgements: $\Delta \rightarrow_\nu \Delta'$, which handles reduction of individual local types, and a synchronisation judgement $\Delta \xrightarrow{\nu} \Delta'$.

Rule ET-Conn handles the reduction of role $p$, where the choice session type contains a connection action $q!!\ell_j(A_j)$. If $q$ has a statically-defined session type $\Sigma_{k \in K} (p??\ell_k(B_k), T_k)$ which can accept $\ell_j$ from from $p$, and the payload types match, reduction advances $p$'s session type, adds $q$ to $p$'s connected role set, and introduces an entry for $q$ into the environment.

The reduction emits a label $s:p \rightarrow q:\ell_j$.

Given a role $p$ connected to $q$ with a session choice containing a send or receive action $q!\ell_j(A_j), S_j$, rule ET-Act will emit a label $s:p,q:\ell_j(A_j)$ and advance the session type of $p$.

Rule ET-Wait handles the reduction of $#\uparrow q, S$ actions, $s[p](\tau, q):#\uparrow q, S$, where $p$ waits for $q$ to disconnect: the reduction emits label $p,q,#\downarrow$ and removes $q$ from $p$'s connected roles. Similarly, rule ET-Disconn handles disconnection, by emitting label $p,q,#\downarrow$ and removing
A runtime typing environment is safe, written \textit{safe}(\Delta), if \(\varphi(\Delta)\) for a safety property \(\varphi\).
Clause (1) ensures that communication actions between participants are compatible: if $p$ is sending a message with label $\ell$ and payload type $A$ to $q$, and $q$ is receiving from $p$, then the two roles must be connected, and $q$ must be able to receive $\ell$ with a matching payload.

Clause (2) states that if $p$ is connecting to a role $q$ with label $\ell$, then $q$ should not already be involved in the session, and should be able to accept from $p$ on message label $\ell$ with a compatible payload type. The requirement that $q$ is not already involved in the session rules out the correlation errors described in Section 5.2.1. Clause (3) handles recursion, and clause (4) requires that safety is preserved under environment reduction.

**Concretising the safety property.** In order to deduce that a runtime typing environment $\Delta$ is safe, we define $\varphi(\Delta) = \{\Delta' \mid \Delta \Rightarrow^* \Delta'\}$ and verify that $\varphi$ is a safety property by ensuring that it satisfies all clauses in Definition 5.

**Properties on protocols and programs.** It is useful to distinguish active and inactive session types, depending on whether their associated role is currently involved in a session, and identify the initiator of a session.

**Definition 6 (Active and Inactive Session Types).** A session type $S$ is inactive, written $\text{inactive}(S)$, if $S = \text{end}$ or $S = \Sigma_{i \in I} \{p_?\ell_i(A_i).S_i\}$. Otherwise, $S$ is active, written $\text{active}(S)$.

**Definition 7 (Initiator, unique initiator).** Given a protocol $P$, a role $p : S_p \in P$ is an initiator if $S_p = \Sigma_{i \in I}(\alpha_i.S_i)$, and each $\alpha_i$ is a connection action $q!!\ell_i(A_i)$. Role $p$ is a unique initiator of $P$ if inactive($S_q$) for all $q \in P \setminus \{p : S_p\}$.

A protocol is well-formed if it is safe and has a unique initiator.

**Definition 8 (Well-formed protocol).** A protocol $P = \{p_i : S_i\}_{i \in I}$ is well-formed if it has a unique initiator $q$ of type $S$ and $\text{safe}(s(q)(\emptyset); S)$ for any $s$.

By way of example, the online shopping protocol is well-formed: Customer is the protocol’s unique initiator, and it is straightforward to verify that $\text{safe}(s(\text{Customer})(\emptyset); \text{ty(Customer)})$.

**Definition 9 (Well-formed program).** A program $(\vec{D}, \vec{P}, M)$ is well-formed if:

1. For each actor definition $D = \text{actor u follows} S \{N\} \in \vec{D}$, there exists some role $p \in \vec{P}$ such that $\text{ty}(p) = S$, and $\{S\} \cdot \mid S \triangleright N : A \triangleleft \text{end}$
2. Each protocol $P \in \vec{P}$ is well-formed and has a distinct set of roles
3. The ‘boot clause’ $M$ is typable under the empty typing environment and does not perform any communication actions: $\{\text{end}\} \cdot \mid \text{end} \triangleright M : A \triangleleft \text{end}$

When discussing the metatheory, we only consider configurations defined with respect to a well-formed program. Specifically, we henceforth assume that each actor definition in the system follows a session type matched by a role in a given protocol, assume each role belongs to a single protocol, and assume that all protocols are well-formed.

We can now state our first main result: given a safe runtime environment, configuration reduction preserves typability. Details can be found in Appendix B.

We write $\mathcal{R}'$ for the reflexive closure of a relation $\mathcal{R}$.

**Theorem 10 (Preservation (Configurations)).** If $\Gamma : \Delta \vdash C$ with $\text{safe}(\Delta)$, and $C \rightarrow C'$, then there exists some $\Delta'$ such that $\Delta \Rightarrow^* \Delta'$ and $\Gamma : \Delta' \vdash C'$.

Preservation shows that each actor conforms to its session type, and that communication never introduces unsoundness due to mismatching payload types.
5.4.3 Progress

We now show a progress property, which shows that given protocols which satisfy a progress property, Ensembles configurations do not get stuck due to deadlocks.

We begin with some auxiliary definitions. A final runtime typing environment contains a single, disconnected role with session type end, reflecting the intuition that all connected roles will eventually disconnect from a protocol initiator.

Definition 11 (Final environment). An environment \( \Delta \) is final, written \( \text{end}(\Delta) \), \( \Delta = \{ s[p]\langle \emptyset \rangle \text{-} \text{end} \} \) for some \( s \) and \( p \).

So far, we have considered safe protocols, which ensure the absence of communication mismatches. We say that an environment satisfies progress if each active role can eventually perform an action, each potential send is eventually matched by a receive, and non-reducing environments are final. Let \( \text{roles}(\rho) \) denote the roles referenced in a synchronisation label (i.e., \( \text{roles}(\rho) = \{ p, q \} \) for \( \rho \in \{ s[p] \Rightarrow q::\ell, s[p] q::\ell, s[p] q#q \} \)).

Definition 12 (Progress (Runtime typing environments)). A runtime typing environment \( \Delta \) satisfies progress, written \( \text{prog}(\Delta) \), if:

- (Role progress) for each \( s[p]\langle \{ q_i \} \rangle \in \Delta \) s.t. \( \text{active}(S_i) \), \( \Delta \Rightarrow^* \Delta' \frac{\rho}{\rightarrow} \) with \( p \in \text{roles}(\rho) \)
- (Eventual comm.) if \( \Delta \Rightarrow^* \Delta' \xrightarrow{\rho} \Delta'' \xrightarrow{\rho} \frac{\rho}{\rightarrow}, \) with \( p \not\in \text{roles}(\rho) \)
- (Correct termination) \( \Delta \Rightarrow^* \Delta' \not\Rightarrow \) implies \( \text{end}(\Delta) \)

The online shopping example satisfies progress, since all roles will always eventually be able to fire an action once connected, and since all roles disconnect, the non-reducing final environment will be of the form \( s[\text{Customer}]\langle \emptyset \rangle \text{-} \text{end} \).

When considering progress results, we assume that all protocols satisfy progress. It is useful to define a configuration context \( \mathcal{G} \) as the one-hole context \( \mathcal{G} ::= [ ] | (\nu s)\mathcal{G} | \mathcal{G} \parallel \mathcal{C} \).

To help us state a progress property, we consider configurations in canonical form. We begin by defining a session, which consists of a session name restriction and all connected actors and zapper threads:

Definition 13 (Session). A configuration is a session \( S \) if it can be written:

\[
(\nu s)((a_1, M_1, s[p_1]\langle \{ q_1 \} \rangle, \kappa_1) \parallel \cdots \parallel (a_m, M_m, s[p_m]\langle \{ q_m \} \rangle, \kappa_m) \parallel \{ s[p_{m+1}] \parallel \cdots \parallel s[p_n] \})
\]

A canonical form consists of binders for all connected actor names, followed by binders for all disconnected actor names, followed by all sessions, followed by all disconnected actors.

Definition 14 (Canonical form). A configuration is in canonical form if it is either 0 or can be written: \( (\nu a_1 \cdots a_l)(\nu b_1 \cdots b_m)(S_1 \parallel \cdots \parallel S_n \parallel (b_1, M_1, 1, \kappa_1) \parallel \cdots \parallel (b_m, M_m, 1, \kappa_m)) \).

Every well-typed, closed configuration can be written in canonical form.

Lemma 15 (Canonical forms). If \( :: \vdash \mathcal{C} \), then \( \exists \mathcal{D} \equiv \mathcal{C} \) where \( \mathcal{D} \) is in canonical form.

To characterise configuration progress, we need three further definitions. An actor is terminated if it has reduced to a value or has an unhandled exception, and has the behaviour stop. An unmatched discover occurs when no other actors match a given session type. An actor is accepting if it is ready to accept a connection.

Definition 16 (Terminated actor, unmatched discover, accepting actor).

- An actor \( \langle a, M, \sigma, \kappa \rangle \) is terminated if \( M = \text{return} \) or \( M = \text{Ep}[\text{raise}] \), and \( \kappa = \text{stop} \).
An actor $\langle a, M, \sigma, \kappa \rangle$ which is a subconfiguration of $C$ has an unmatched discover if no other non-terminated actor in $C$ has session type $S$.

An actor $\langle a, M, \sigma, \kappa \rangle$ is accepting if $M = E[\text{accept from } p \{\ell_j(x_j) \rightarrow N_j\}]$ for some evaluation context $E$ and role $p$.

Unhandled exceptions will propagate through a session, progressively cancelling all roles. A failed session consists of only zapper threads.

Definition 17 (Failed session). We say that a session $S$ is a failed session, written failed($S$), if $S \equiv (\nu s)(\nu s[p_1] \parallel \cdots \parallel \nu s[p_n])$.

The key session progress lemma establishes the reducibility of each session which does not contain an unmatched discover and is typable under a reducible runtime typing environment.

Lemma 18 (Session Progress). If $\vdash C$ where $C$ does not contain an unmatched discover, $C \equiv G[S]$ and $S = (\nu s : \Delta)D$ with prog($\Delta$), and $S$ is not a failed session, then $C \rightarrow \rightarrow$.

There are several steps to proving Lemma 18. First, we introduce exception-aware reduction on runtime typing environments, which explicitly accounts for zapper threads at the type level, and show that exception-aware environments threads retain safety and progress. Second, we introduce flattenings, which show that runtime typing environments containing only unary output choices can type configurations blocked on communication actions, and that flattenings preserve environment reducibility. Finally, we show that configurations typable under flat, reducible typing environments can reduce. Full details can be found in Appendix B.

We can now show our second main result: in the absence of unmatched discovers, a configuration can either reduce, or it consists only of accepting and terminating actors.

Theorem 19 (Progress). Suppose $\vdash C$ where $C$ is in canonical form. If $C$ does not contain an unmatched discover, either $\exists D$ such that $C \rightarrow \rightarrow D$, or $C \equiv \nu b_1 \cdots \nu b_n(\nu b_1, N_1, \bot, \kappa_1) \parallel \cdots \parallel (\nu b_n, N_n, \bot, \kappa_n)$ where each $b_i$ is terminated or accepting.

The proof eliminates all failed sessions by the structural congruence rules; shows that the presence of sessions implies reducibility (Lem. 18); and reasons about disconnected actors.

In addition to each actor conforming to its session type (Thm. 10), Theorem 19 guarantees that the system does not deadlock. It follows that session types ensure safe communication.

Theorem 19 assumes the absence of unmatched discovers. This is not a significant limitation in practice, however, as unmatched discovers can be mitigated with timeouts, where a timeout would trigger an exception.

Related Work

Behavioural typing for actors. Mostroux and Vasconcelos [47] present the first theoretical account of session types in an actor language; their work effectively overlays a channel-based session typing discipline on mailboxes using Erlang’s reference generation capabilities.

Neykova and Yoshida [49] use MPSTs to specify communication in an actor system, implemented in Python. Fowler [21] implements similar ideas in Erlang, with extensions to allow subsessions [17] and failure handling. Neykova and Yoshida [50] later improve the recovery mechanism of Erlang by using MPSTs to calculate a minimal set of affected roles. The above works check multiparty session typing dynamically. We are first to both formalise and implement static multiparty session type checking for an actor language.

Active objects (AOs) [15] are actor-like concurrent objects where results of asynchronous method invocations are returned through futures. Bagherzadeh and Rajan [4] study order
types for an AO calculus, which characterise causality and statically rule out data races. In contrast to MPSTs, order types work bottom-up through type inference. Kamburjan et al. [39] apply an MPST-like system to Core ABS [38], a core AO calculus; they establish soundness via a translation to register automata rather than via an operational semantics.

de’Liguoro and Padovani [16] introduce mailbox types, a type system for first-class, unordered mailboxes. Their calculus generalises the actor model, since each process can be associated with more than one mailbox. Their type discipline allows multiple writers and a single reader for each mailbox, and ensures conformance, deadlock-freedom, and for many programs, junk-freedom. Our approach is based on MPSTs and is more process-centric.

Non-classical multiparty session types. MPSTs were introduced by Honda et al. [31]. Classical MPST theories are grounded in binary duality: safety follows as a consequence of consistency (pointwise binary duality of interactions between participants), and deadlock-freedom follows from projectability from a global type.

Unfortunately, classical MPSTs are restrictive: there are many protocols which are intuitively safe but not consistent. Scalas and Yoshida [54] introduced the first non-classical MPST theory. Instead of ensuring safety using binary duality, they define an LTS on local types and a safety property suitable for proving type preservation; since the type system is parametric in the safety property (inspired by Igarashi and Kobayashi [35] in the π-calculus), the property can be customised in order to guarantee different properties such as deadlock-freedom or liveness. A key contribution of our work is the use of non-classical MPSTs to provide the first concrete language design for explicit connection actions [32].

Adaptation. None of the above work considers adaptation. The literature on formal studies of adaptation is mainly based on process calculi, without programming language design or implementation. Bravetti et al. [8] develop a process calculus that allows parts of a process to be dynamically replaced with new definitions. Their later work [9] uses temporal logic rather than types to verify adaptive processes. Di Giusto and Pérez [18] define a session type system for the same process calculus, and prove that adaptation does not disrupt active sessions. Later, Di Giusto and Pérez [19] use an event-based approach so that adaptation can depend on the state of a session protocol. Anderson and Rathke [2] develop an MPST-like calculus and study dynamic software update providing guarantees of communication safety and liveness. Differently from our work, they do not consider runtime discovery of software components and do not provide an implementation.

Coppo et al. [13] consider self-adaptation, in which a system reconfigures itself rather than receiving external updates. They define an MPST calculus with self-adaptation and prove type safety. Castellani et al. [11] extend [13] to also guarantee properties of secure information flow, but neither of these works have been implemented. Dalla Preda et al. [52] develop the AIOCIJ system based on choreographic programming for runtime updates. Their work is implemented in the Jolie language [45], but they do not consider runtime discovery.

In this work we focus on correct communication in the absence of adversaries, and do not consider security. The literature on security and behavioural types is surveyed by Bartoletti et al. [5] and could provide a basis for future work on security properties.

7 Conclusion and Future Work

Modern computing increasingly requires software components to adapt to their environment, by discovering, replacing, and communicating with other components which may not be part
of the system’s original design. Unfortunately, up until now, existing programming languages
have lacked the ability to support adaptation both safely and statically. We therefore asked:

Can a programming language support static (compile-time) verification of safe
runtime dynamic self-adaptation, i.e., discovery, replacement and communication?

We have answered this question in the affirmative by introducing ENSEMBLES, an actor-
based language supporting adaptation, which uses multiparty session types to guarantee
communication safety, using explicit connection actions to invite discovered actors into a
session. We have demonstrated the safety of our system by proving type soundness theorems
which state that each actor follows its session type, and that communication does not
introduce deadlocks. Our formalism makes essential use of non-classical MPSTs.

Future work. Currently, actors only take part in a single session; it would be interesting to
generalise this assumption. In order to avoid session correlation errors, we require that each
role includes at most a single top-level `accept` construct (c.f. [32]). It would be interesting to
investigate the more general setting, which would likely require dependent types.

References

1 Gul A. Agha. ACTORS - a model of concurrent computation in distributed systems. MIT
2 Gabrielle Anderson and Julian Rathke. Dynamic software update for message passing
programs. In Ranjit Jhala and Atsushi Igarashi, editors, Programming Languages
and Systems - 10th Asian Symposium, APLAS 2012, Kyoto, Japan, December 11-13,
https://doi.org/10.1007/978-3-642-35182-2_15.
3 Robert Atkey. Parameterised notions of computation. Journal of Functional Programming,
4 Mehdi Bagherzadeh and Hridesh Rajan. Order types: static reasoning about message
5 Massimo Bartoletti, Ilaria Castellani, Pierre-Malo Denié­lou, Mariangiola Dezani-
Ciancaglini, Silvia Ghilezan, Jovanka Pantovic, Jorge A. Pérez, Peter Thiemann, Bernardo
Toninho, and Hugo Torres Vieira. Combining behavioural types with security analysis.
6 J. Baumann, F. Hohl, K. Rothermel, and M. Straßer. MOLE — Concepts of Mobile
7 G. Bouabene, C. Jelger, C. Tschudin, S. Schmid, A. Keller, and M. May. The autonomic
network architecture (ana). IEEE Journal on Selected Areas in Communications, 28(1):
8 Mario Bravetti, Cinzia Di Giusto, Jorge A. Pérez, and Gianluigi Zavattaro. Towards
the verification of adaptable processes. In Proceedings (Part I) of the 5th
International Symposium on Leveraging Applications of Formal Methods, Verification
and Validation (iSoLA), volume 7690 of Lecture Notes in Computer Science,
//doi.org/10.1007/978-3-642-34026-0_20.


Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.


REFERENCES


A Global types and projection

Syntax of types. Similar to the presentation of local types, the key difference between this presentation of global types and standard MPST systems is that instead of having branching of the form $r \rightarrow s \{\ell_i : G_i\}_i$, where a single role makes a choice, this presentation of global types provides arbitrary summations of global actions. The standard sending action is therefore $p \rightarrow q : \ell(A). G$ which involves role $p$ sending message $\ell(A)$ to $q$.

A key technical innovation is explicit connection actions ($p \rightarrow q : \ell(A)$) which establish a connection between two roles $p$ and $q$ by sending a label $\ell$. Conversely, disconnection $p \# q$ is a directed disconnection where $p$ disconnects from $q$, and $q$ waits for $p$’s disconnection.

Instead of assuming that all roles are connected at the start of the the session, in order to communicate, each role must be connected explicitly. This allows communication paths where one party isn’t involved (as in Hu and Yoshida’s travel agent example).

Projection. Global types can be projected at a role in order to obtain a local type. In standard MPST systems, a global type is well-formed if it is closed, contractive, and projectable at all roles. A well-formed global type is then guaranteed to be deadlock-free.

Projection in this system is quite different, in that projection does not guarantee deadlock-freedom in itself. Rather, projection produces local types, which can be validated separately as described in the main body of the paper. The inclusion of the $\Phi$ parameter allows the pruning of unguarded recursion variables. Projection on unary choices, recursive types, type variables, and session termination are standard. Disconnection ensures that the disconnecting role is unused in the remainder of the protocol.

Now consider the case of projecting $n$-ary choices for $n > 1$. Suppose we have $\Sigma_{i \in I}(G_i) |_{\Phi} r$. If $G_i |_{\Phi} r = \text{end}$ for all $G_i$, then the result of the projection is end (and similarly for a recursion variable $X$).

Otherwise, there will be some subset $J$ of $I$ such that each $L_j$ for $j \in J$ is a communication action, and each $L_k \in I \setminus J$ projects to either end or a recursion variable. For each $j \in J$, there is a further condition: the choice must consist of all send actions, or all receive actions. If the latter, then the receiver must be the same in all branches. This will ensure syntactically well-formed local types used in the paper.
Meta-level definitions

\[ \text{isOutput}(L) \triangleq L \in \{ p!\ell(A), p!!\ell(A) \} \]
\[ \text{isInput}(L) \triangleq L \in \{ p?\ell(A), p??\ell(A) \} \]

Projection

\[
G | r = G | \emptyset r \\
(p \to q : \ell(A).G) | \emptyset r = \begin{cases} 
q!\ell(A). (G | \emptyset r) & r = p \\
P?\ell(A). (G | \emptyset r) & r = q \\
G | \emptyset r & r \not\in \{p, q\} 
\end{cases}
\]
\[
(p \to q : \ell(A).G) | \emptyset r = \begin{cases} 
q!\ell(A). (G | \emptyset r) & r = p \\
P?\ell(A). (G | \emptyset r) & r = q \\
G | \emptyset r & r \not\in \{p, q\} 
\end{cases}
\]
\[
(p \# q . G) | \emptyset r = \begin{cases} 
\# | q & r = p \text{ and } G | \emptyset r = \text{end} \\
\# | p . (G | \emptyset r) & r = q \\
G | \emptyset r & r \not\in \{p, q\} 
\end{cases}
\]
\[
\sum_{i \in I} (G_i) | \emptyset r = \begin{cases} 
X & \forall i \in I, (G_i | \emptyset r) = X \\
\text{end} & \forall i \in I, (G_i | \emptyset r) = \text{end} \\
| J | > 1, | J | > 0, \text{ and} \\
\exists k \in I \setminus J, G_k | \emptyset r = \text{end or } X \in \Phi, \text{ and} \\
\forall j \in J, \text{isOutput}(L_j) \text{ either} \\
\exists p \forall j \in J, \text{isInput}(L_j) \land \text{subj}(L_j) = p 
\end{cases}
\]
\[
\mu X . G | \emptyset r = \begin{cases} 
\text{end} & G | \emptyset . X r = X' \text{ or end} \\
G | \Delta . X r & \text{otherwise} 
\end{cases}
\]
\[
X | \emptyset r = X \\
\text{end} | \emptyset r = \text{end}
\]

\[ \text{Figure 18 Global types and projection} \]
B Proofs for Section 5

B.1 Preservation

Lemma 20 (Preservation (Terms)). If \( \{ T \} \ \Gamma \vdash S \triangleright A \triangleleft S' \) and \( M \rightarrow_M N \), then \( \{ T \} \ \Gamma \vdash S \triangleright M \triangleright N \triangleright A \triangleleft S' \).

Proof. By induction on the derivation of \( M \rightarrow_M N \).

Lemma 21 (Preservation (Equivalence)). If \( \Gamma; \Delta \vdash C \) and \( C \equiv D \), then \( \Gamma; \Delta \vdash D \).

Proof. By induction on the derivation of \( C \equiv D \).

Lemma 22 (Substitution). If \( \{ T \} \ \Gamma, x : A \vdash S \triangleright M \triangleright B \triangleleft S' \) and \( \Gamma \vdash V : A \), then \( \{ T \} \ \Gamma \vdash S \triangleright M \{ V/x \} \triangleright B \triangleleft S' \).

Proof. By induction on the derivation of \( \{ T \} \ \Gamma, x : A \vdash S \triangleright M \triangleright B \triangleleft S' \).

Lemma 23 (Subterm typability). Suppose \( D \) is a derivation of \( \{ T \} \ \Gamma \vdash S \triangleright E[M] : A \triangleleft S' \). Then there exists some subderivation \( D' \) of \( D \) concluding \( \{ T \} \ \Gamma \vdash S \triangleright M : B \triangleleft S'' \) for some type \( B \) and local type \( S'' \), where the position of \( D' \) in \( D \) corresponds to that of the hole in \( E \).

Proof. By induction on the structure of \( E \).

Lemma 24 (Replacement). If:

1. \( D \) is a derivation of \( \{ U \} \ \Gamma \vdash S \triangleright E[M] : A \triangleleft T \)
2. \( D' \) is a subderivation of \( D \) concluding \( \{ U \} \ \Gamma \vdash S \triangleright M : B \triangleleft T' \), where the position of \( D' \) in \( D \) corresponds to that of the hole in \( E \)
3. \( \{ U \} \ \Gamma \vdash S \triangleright N : B \triangleleft T' \)

Then \( \{ U \} \ \Gamma \vdash S \triangleright E[N] : A \triangleleft T \).

Proof. By induction on the structure of \( E \).

Lemma 25 (Replacement (Pure contexts)). If:

1. \( D \) is a derivation of \( \{ U \} \ \Gamma \vdash S \triangleright E[M] : A \triangleleft T \)
2. \( D' \) is a subderivation of \( D \) concluding \( \{ U \} \ \Gamma \vdash S \triangleright M : B \triangleleft T' \), where the position of \( D' \) in \( D \) corresponds to that of the hole in \( E_p \)
3. \( \{ U \} \ \Gamma \vdash S' \triangleright N : B \triangleleft T' \)

Then \( \{ U \} \ \Gamma \vdash S \triangleright E_p[N] : A \triangleleft T \).

Proof. By induction on the structure of \( E_p \), noting that \( S \) is not constrained by exception handling frames.

Theorem 10 (Preservation (Configurations)). If \( \Gamma; \Delta \vdash \mathcal{C} \) with \( \text{safe(\Delta)} \), and \( \mathcal{C} \rightarrow \mathcal{C}' \), then there exists some \( \Delta' \) such that \( \Delta \Rightarrow^* \Delta' \) and \( \Gamma; \Delta' \vdash \mathcal{C}' \).

Proof. By induction on the derivation of \( \mathcal{C} \rightarrow \mathcal{C}' \). Where there is a choice of whether \( \sigma = \bot \) or \( \sigma = s[p](q) : S \), we show the latter case; the technique for proving the former case is identical.

Case E-Loop
\[(a, \text{return } V, \bot, M) \longrightarrow (a, M, \bot, M)\]

Assumption:

\[
\begin{array}{c}
a : \text{Pid}(T) \in \Gamma \\
S = T \lor S = \text{end}
\end{array}
\begin{array}{c}
\{ T \} \quad \Gamma \mid S \triangleright \text{return } V : B \triangleleft \text{end}
\end{array}
\begin{array}{c}
\{ T \} \quad \Gamma \mid M
\end{array}
\]

\[
\Gamma ; a : T \vdash (a, \text{return } V, \bot, M)
\]

Recomposing:

\[
\begin{array}{c}
a : \text{Pid}(T) \in \Gamma
\end{array}
\begin{array}{c}
\{ T \} \quad \Gamma \mid T \triangleright M : A \triangleleft \text{end}
\end{array}
\begin{array}{c}
\{ T \} \quad \Gamma \mid M
\end{array}
\]

\[
\Gamma ; a : T \vdash (a, M, \bot, M)
\]

as required.

**Case E-New**

\[(a, E[\text{new } u], \sigma, \kappa) \longrightarrow (\nu b)((a, E[\text{return } b], \sigma, \kappa) \parallel (b, M, \bot, M))\]

where \(b\) is fresh, and \(u.\text{behaviour} = M\).

Assumption:

\[
\begin{array}{c}
a : \text{Pid}(S) \in \Gamma
\end{array}
\begin{array}{c}
\{ S \} \quad \Gamma \mid S \triangleright E[\text{new } u] : A \triangleleft \text{end}
\end{array}
\begin{array}{c}
\{ S \} \quad \Gamma \mid \kappa
\end{array}
\]

\[
\Gamma ; a : S, s[p][q] : S' \vdash (a, E[\text{new } u], s[p][q], \kappa)
\]

By Lemma 23:

\[u.\text{sessionType} = T\]

\[
\begin{array}{c}
\{ S \} \quad \Gamma \mid S' \triangleright \text{new } u : \text{Pid}(T) \triangleleft S'
\end{array}
\]

By Lemma 24, \(\{ S \} \quad \Gamma, b : \text{Pid}(T) \mid T \triangleright \text{return } b : \text{Pid}(T) \triangleleft S'\).

Let \(\Gamma' = \Gamma, b : \text{Pid}(T)\). Note that by weakening, everything typable under \(\Gamma\) is also typable under \(\Gamma'\).

By T-Def:

\[
\begin{array}{c}
\{ T \} \mid T \triangleright M : B \triangleleft \text{end}
\end{array}
\begin{array}{c}
\vdash \text{actor } u \text{ follows } T \{ M \}
\end{array}
\]

Again by weakening, \(\{ T \} \quad \Gamma' \mid T \triangleright M : B \triangleleft \text{end}\).

Recomposing:

\[
\begin{array}{c}
a : \text{Pid}(S) \in \Gamma'
\end{array}
\begin{array}{c}
\{ S \} \quad \Gamma' \mid S \triangleright E[\text{return } b] : A \triangleleft \text{end}
\end{array}
\begin{array}{c}
\{ S \} \quad \Gamma' \mid \kappa
\end{array}
\]

\[
\begin{array}{c}
b : \text{Pid}(T) \in \Gamma'
\end{array}
\begin{array}{c}
\{ T \} \quad \Gamma' \mid T \triangleright M : B \triangleleft \text{end}
\end{array}
\begin{array}{c}
\{ T \} \quad \Gamma' \mid M
\end{array}
\]

\[
\Gamma' ; a : S, b : S \vdash (a, E[\text{return } b], s[p][q], \kappa) \parallel (b, M, \bot, M)
\]

\[
\Gamma ; a : S \vdash (\nu b)((a, E[\text{return } b], s[p][q], \kappa) \parallel (b, M, \bot, M))
\]

as required.
Case E-Replace

\[ \langle a, E[\text{replace } b \text{ with } \kappa'_{a}], \sigma, \kappa \rangle \parallel \langle b, M, \sigma, \kappa_2 \rangle \]

Assumption:

\[
\begin{align*}
\Gamma; a : \text{Pid}(U_a) \in \Gamma & \quad \{U_a\} \quad \text{if } S \triangleright E[\text{replace } b \text{ with } \kappa'_{a}]; A \triangleq \text{end} \quad \{U_a\} \quad \Gamma \vdash \kappa_1 \\
\Gamma; a : U_a, s[p](S); r \vdash \langle a, E[\text{replace } b \text{ with } \kappa'_{a}]), \sigma, s[p](r) \rangle & \quad \{U_b\} \quad \Gamma \vdash \text{pid}(U_b) \\
\Gamma; b : U_b, t[q](S); T \vdash \langle a, E[\text{replace } b \text{ with } \kappa'_{a}], \sigma, s[p](r) \rangle \parallel \langle b, M, t[q](S), \kappa_2 \rangle & \quad \{U_b, T\} \quad \Gamma \vdash \kappa_2
\end{align*}
\]

By Lemma 23:

\[
\{U_a\} \quad \Gamma \vdash \kappa_2 \\
\{U_b\} \quad \Gamma \vdash \text{pid}(U_b)
\]

By Lemma 24, \( \{U_a\} \quad \Gamma \vdash S \triangleright E[\text{return }()]]; A \triangleq \text{end}. \)

Rerecomposing:

\[
\begin{align*}
\Gamma; a : U_a, s[p](S); r \vdash \langle a, E[\text{return }()]], \sigma, s[p](r) \rangle & \quad \{U_a\} \quad \Gamma \vdash \kappa_1 \\
\Gamma; b : U_b, t[q](S); T \vdash \langle a, E[\text{return }()]], \sigma, s[p](r) \rangle \parallel \langle b, M, t[q](S), \kappa_2 \rangle & \quad \{U_b, T\} \quad \Gamma \vdash \kappa_2
\end{align*}
\]

as required.

Case E-ReplaceSelf

\[ \langle a, E[\text{replace } a \text{ with } \kappa'], \sigma, \kappa \rangle \longrightarrow \langle a, E[\text{return }()]], \sigma, \kappa' \rangle \]

Assumption:

\[
\begin{align*}
\Gamma; a : \text{pid}(T) \in \Gamma & \quad \{T\} \quad \Gamma \vdash S \triangleright E[\text{replace } a \text{ with } \kappa']]; A \triangleq \text{end} \quad \{T\} \quad \Gamma \vdash \kappa \\
\Gamma; a : T, s[p](q) : S \vdash \langle a, E[\text{replace } a \text{ with } \kappa'], s[p](q), \kappa \rangle & \quad \{T\} \quad \Gamma \vdash \text{pid}(T)
\end{align*}
\]

By Lemma 23:

\[
\{T\} \quad \Gamma \vdash \kappa' \\
\{T\} \quad \Gamma \vdash \text{pid}(T)
\]

\[
\{T\} \quad \Gamma \vdash S \triangleright \text{replace } a \text{ with } \kappa'; 1 \triangleq S
\]

(noting that \( \Gamma \vdash \text{pid}(T) \) because \( a : \text{pid}(T) \in \Gamma \), as per the T-CONNECTEDActor and T-DISCONNECTEDActor rules).

By Lemma 24, \( \{T\} \quad \Gamma \vdash S \triangleright E[\text{return }()]]; A \triangleq \text{end}. \)
Case E-Self

\[ a : \text{Pid}(T) \in \Gamma \quad \{T\} \quad \Gamma \vdash E[\text{return }()] : A \text{ and } \{T\} \Gamma \vdash \kappa' \]
\[ \Gamma; a : T, s[p](S) : T \vdash \langle a, E[\text{return }()], s[p](\kappa), \kappa' \rangle \]

as required.

Case E-Discover

\[ \langle a, E[\text{discover } T], \sigma_1, \kappa_1 \rangle \parallel \langle b, M, \sigma_2, \kappa_2 \rangle \rightarrow \langle a, E[\text{return } b], \sigma_1, \kappa_1 \rangle \parallel \langle b, M, \sigma_2, \kappa_2 \rangle \]

where \( b.S\text{essionType} = T \) and \( \neg(M = \text{return } V \land \kappa_2 = \text{stop}) \).

Assumption:

\[ a : \text{Pid}(S) \in \Gamma \]
\[ \{S\} \Gamma \vdash \text{discover } T : A \text{ and } \{S\} \Gamma \vdash \kappa_1 \]
\[ b : \text{Pid}(T) \in \Gamma \quad \{T\} \quad \Gamma \vdash M : B \text{ and } \{T\} \Gamma \vdash \kappa_2 \]
\[ \Gamma; a : S, b : T, s[p](T), s[p](\kappa) : T' \vdash \langle a, E[\text{discover } S], \sigma_1, \kappa_1 \rangle \parallel \langle b, M, \sigma_2, \kappa_2 \rangle \]

By Lemma 23:

\[ \{T\} \Gamma \vdash \text{discover } T : \text{Pid}(T) \triangleleft S' \]

Since \( b : \text{Pid}(T) \in \Gamma \), we can show \( \{T\} \Gamma \vdash \text{discover } T : \text{Pid}(T) \triangleleft S' \).

By Lemma 24, \( \{T\} \Gamma \vdash \text{return } b : A \triangleleft S' \).

Thus, recomposing:

\[ a : \text{Pid}(S) \in \Gamma \quad \{S\} \Gamma \vdash \text{return } b : A \text{ and } \{S\} \Gamma \vdash \kappa_1 \]
\[ b : \text{Pid}(T) \in \Gamma \quad \{T\} \quad \Gamma \vdash M : B \text{ and } \{T\} \Gamma \vdash \kappa_2 \]
\[ \Gamma; a : S, b : T, s[p](T), s[q](\kappa) : T' \vdash \langle a, E[\text{return } b], s[p](\kappa), \kappa_1 \rangle \parallel \langle b, M, s[q](\kappa), \kappa_2 \rangle \]

as required.

Case E-Self

\[ \langle a, E[\text{self}], \sigma, \kappa \rangle \rightarrow \langle a, E[\text{self}], \sigma, \kappa \rangle \]

Assumption:

\[ a : \text{Pid}(T) \in \Gamma \quad \{T\} \quad \Gamma \vdash E[\text{self}] : A \triangleleft S \]
\[ \Gamma; a : T, s[p](q) : S \vdash \langle a, E[\text{self}], s[p](q), \kappa \rangle \]

By Lemma 23:

REFERENCES
We can show:

\[ \Gamma \vdash a : \text{Pid}(T) \]

\[ \{ T \} | S \triangleright E[\text{return } a]:A \triangleleft \text{end} \]

By Lemma 24, \( \{ T \} | S \triangleright E[\text{return } a]:A \triangleleft \text{end} \)

Recomposing:

\[ a : \text{Pid}(T) \in \Gamma \quad \{ T \} | S \triangleright E[\text{return } a]:A \triangleleft \text{end} \]

\[ \Gamma; a : T, s[p](q); S \vdash \langle a, E[\text{return } a], s[p](q), \kappa \rangle \]

as required.

**Case E-ConnInit**

\[ \langle a, E[F[\textbf{connect } \ell_j(V) \text{ to } b \text{ as } q]], \bot, \kappa_1 \rangle \parallel \langle b, E'[\textbf{accept from } p \{ \ell_i(x_i) \mapsto M_i \}_{i \in I}], \bot, \kappa_2 \rangle \rightarrow \]

\[ (\nu s)(\langle a, E[\textbf{return } ()], s[p](q), \kappa_1 \rangle \parallel \langle b, E'[M_j[V/x_i]], s[q](p), \kappa_2 \rangle) \]

Assumption:

\[ a : \text{Pid}(U_a) \in \Gamma \quad S = U_a \lor S = \text{end} \]

\[ \{ U_a \} | S \triangleright E[\textbf{connect } \ell_j(V) \text{ to } b \text{ as } q]:A \triangleleft \text{end} \]

\[ b : \text{Pid}(U_b) \in \Gamma \quad T = U_b \lor T = \text{end} \]

\[ \{ U_b \} | T \triangleright E'[\textbf{accept from } p \{ \ell_i(x_i) \mapsto M_i \}_{i \in I}], B \triangleleft \text{end} \]

\[ \Gamma; a : U_a, b : U_b \vdash \langle a, E[\textbf{connect } \ell_j(V) \text{ to } b \text{ as } q], \bot, \kappa_1 \rangle \parallel \langle b, E'[\textbf{accept from } p \{ \ell_i(x_i) \mapsto M_i \}_{i \in I}], \bot, \kappa_2 \rangle \]

By Lemma 23, by case analysis on \( F \), we either have:

\[ q!\ell_j(A_j) \cdot S_j' \in \{ S_i \} \quad \Gamma \vdash b : T_b \quad \text{ty}(q) = T_b \]

\[ \{ U_a \} | \Sigma_{i \in I}(S_i) \triangleright \textbf{connect } \ell_j(V) \text{ to } b \text{ as } q; 1 \triangleleft S_j' \]

or

\[ q!\ell_j(A_j) \cdot S_j' \in \{ S_i \} \quad \Gamma \vdash b : T_b \quad \text{ty}(q) = T_b \]

\[ \{ U_a \} | \Sigma_{i \in I}(S_i) \triangleright \textbf{connect } \ell_j(V) \text{ to } b \text{ as } q; 1 \triangleleft S_j' \]

\[ \{ U_a \} | \Sigma_{i \in I}(S_i) \triangleright \textbf{try connect } \ell_j(V) \text{ to } b \text{ catch } 1 \triangleleft S_j' \]

Since the evaluation context will be discarded, WLOG we proceed assuming that \( F = [ ] \).

As \( b : \text{Pid}(U_b) \in \Gamma \), we have that \( T_b = U_b \) and therefore that \( \text{ty}(q) = U_b \).
Again by Lemma 23, \( \{U_b\} \vdash \Sigma_{i \in \Gamma} (p??\ell_k(C_k) \cdot T_k) \triangleright \text{accept from } p \{\ell_i(x_k) \mapsto M_k\}_{k \in K}:B' \triangleleft T' \).

By case analysis on \( F' \), we either have:

\[
\begin{align*}
\Gamma, x : C_k &\vdash T_k \triangleright M_k : B' \triangleleft T' \quad \forall k \in K \\
\{U_b\} \vdash \Sigma_{i \in \Gamma} (p??\ell_k(C_k) \cdot T_k) \triangleright \text{accept from } p \{\ell_i(x_k) \mapsto M_k\}_{k \in K}:B' \triangleleft T' \\
\end{align*}
\]

or

\[
\begin{align*}
\Gamma, x : C_k &\vdash T_k \triangleright M_k : B' \triangleleft T' \quad \forall k \in K \\
\{U_b\} \vdash \Sigma_{i \in \Gamma} (p??\ell_k(C_k) \cdot T_k) \triangleright \text{accept from } p \{\ell_i(x_k) \mapsto M_k\}_{k \in K}:B' \triangleleft T' \\
\{U_b\} \vdash M : A \triangleleft T' \\
\{U_b\} \vdash \text{try (accept from } p \{\ell_i(x_k) \mapsto M_k\}_{k \in K}\text{) catch } M : B' \triangleleft T'
\end{align*}
\]

Again, we consider the first case.
Thus, \( U_b = \text{accept from } p \{\ell_i(x_k) \mapsto M_k\}_{k \in K} \).

We now need to introduce the new runtime typing environment for the new session. We begin with a singleton runtime typing environment \( \{s[p](\emptyset) : U_a\} \), and recall that \( U_a = \{\Sigma_{i \in \Gamma}(S_i)\} \) and \( q!!l_j(A_j) \cdot S_j' \in \{S_i\} \).

Since (by global assumption) protocols are well-formed and so \( p \) is a unique initiator, we know that \( U_a = ty(p) \) and safe(\( \{s[p](\emptyset) : U_a\} \)).

As a consequence of safety, we know that \( C_j = A_j \).

We can then reduce on typing environments:

\[
\begin{align*}
\exists j \in I \cdot S_j \Rightarrow q!!l_j(A_j) \cdot S_j' \quad &\text{ty}(q) = p??\ell_k(C_k), T_k \quad j \in K \\
\Sigma_{i \in \Gamma}(S_i) &\overset{s[p](\emptyset) : s[q](p) : T_j}{\rightarrow} s[p](\emptyset) : s[q](p) : T_j \\
\end{align*}
\]

Since \( s[p](\emptyset) : \Sigma_{i \in \Gamma}(S_i) \rightarrow s[p](q) : S_j', s[q](p) : T_j \), it follows by safety that \( \text{safe}(s[p](q) : S_j', s[q](p) : T_j) \).

Noting that only top-level frames (i.e., \( F, F' \)) can contain exception-handling frames, \( E, E' \) are pure. Thus, we can show

\[
\begin{align*}
\{U_a\} \vdash S_j' \triangleright \text{return (\( A \)) : } A \triangleleft S_j' \quad \text{and so by Lemma 25, } \{U_a\} \vdash S_j' \triangleright E[\text{return (\( A \))}] : 1 \triangleleft \text{end.}
\end{align*}
\]

Similarly, we can show \( \{U_b\} \vdash T_j \triangleright M_j \{V/x_j\} : B' \triangleleft T' \) and so by Lemma 25, \( \{U_b\} \vdash T_j \triangleright E'[M_j \{V/x_j\}] : B \triangleleft \text{end.} \)

Recomposing:

\[
\begin{align*}
a : \text{Pid}(U_a) \in \Gamma &\quad b : \text{Pid}(U_b) \in \Gamma \\
\{U_a\} \vdash S_j' \triangleright \text{return (\( A \)) : } A \triangleleft S_j' &\quad \{U_b\} \vdash T_j \triangleright E'[M_j \{V/x_j\}] : B \triangleleft \text{end.} \\
\end{align*}
\]

as required.

Note: in the remaining communication cases, we consider the case where the top-level frame is empty, since the frame will be discarded in the result.

**Case E-Conn**
We can then show a reduction on typing environments:

\[ (a, E[\text{connect } \ell_j(V) \text{ to } b \text{ as } q], s[p](\bar{r}), \kappa_1) \parallel (b, E'[\text{accept from } p \{ \ell_i(x_i) \mapsto M_i \}_{i \in I}], \bot, \kappa_2) \rightarrow (a, E[\text{return } ()], s[p](\bar{r}), \kappa_1) \parallel (b, E'[M_j[V/x_j]], s[q](p), \kappa_2) \]

with \( j \in K \).

Assumption:

\[
\Gamma; a : U_a, b : U_b, s[p](\bar{r}) : S \vdash (a, E[\text{connect } \ell_j(V) \text{ to } b \text{ as } q], s[p](\bar{r}), \kappa_1) \parallel (b, E'[\text{accept from } p \{ \ell_i(x_i) \mapsto M_i \}_{i \in I}], \bot, \kappa_2)
\]

where \( j \in I \) and \( \text{safe}(a : U_a, b : U_b, s[p](\bar{r}) : S) \).

Consider the subderivation \( \{ U_a \} \Gamma \mid S \triangleright E[\text{connect } \ell_j(V) \text{ to } b \text{ as } q] : A \triangleleft \text{end} \).

By Lemma 23:

\[
\frac{\{ U_a \} \Gamma \mid \Pi_{i \in I}(S_i) \triangleright \text{connect } \ell_j(V) \text{ to } b \text{ as } q \vdash 1 \triangleleft S'_j}{q!!\ell_j(A_j) : S'_j \in \{ S_i \}, \Gamma \vdash b : \text{Pid}(T_b) \quad \text{ty}(q) = T_b}
\]

Since \( b : \text{Pid}(U_a) \in \Gamma \), we have that \( T_b = U_b \). We also deduce that \( S = \Pi_{i \in I}(S_i) \) where \( q!!\ell_j(A_j) : S'_j \in \{ S_i \} \).

Also by Lemma 23:

\[
\frac{\{ U_a \} \Gamma \mid \Pi_{i \in I}(S_i) \triangleright \text{connect } \ell_j(V) \text{ to } b \text{ as } q \vdash \text{accept from } p \{ \ell_i(x_i) \mapsto M_i \}_{k \in K} : B \triangleleft \text{end}}{(\{ U_a \} \Gamma \mid \Pi_{i \in I}(S_i) \triangleright \text{connect } \ell_j(V) \text{ to } b \text{ as } q \vdash \text{accept from } p \{ \ell_i(x_i) \mapsto M_i \}_{k \in K} : B \triangleleft \text{end})}
\]

Thus, \( U_b = \text{accept from } p \{ \ell_i(x_i) \mapsto M_k \}_{k \in K} \).

We can then show a reduction on typing environments:

\[
\frac{\exists j \in K, S_j = q!!\ell_j(A_j), S'_j \ni \text{ty}(q) = p?\ell_i(C_i) \vdash T_t \quad j \in K \quad C_j = A_j}{s[p](\bar{r}) : \Pi_{i \in I}(S_i) \triangleright s[p](\bar{r}), q, s[q](p) : T_j}
\]

(nothing that as a consequence of safety, we know that \( C_j = A_j \)).

Since \( s[p](\bar{r}) : \Pi_{i \in I}(S_i) \rightarrow s[p](\bar{r}), q, S'_j, s[q](p) : T_j \), it follows by safety that \( \text{safe}(s[p](\bar{r}), q, S'_j, s[q](p) : T_j) \).

Noting that only top-level frames (i.e., \( F, F' \)) can contain exception-handling frames, \( E, E' \) are pure. We can show \( \{ U_a \} \Gamma \mid S'_j \triangleright \text{return } () : A \triangleleft S'_j \) and so by Lemma 25, \( \{ U_a \} \Gamma \mid S'_j \triangleright E[\text{return } ()] : 1 \triangleleft \text{end} \).

Similarly, we can show \( \{ U_b \} \Gamma \mid T_j \triangleright M_j[V/x_j] : B' \triangleleft T' \) and so by Lemma 25, \( \{ U_b \} \Gamma \mid T_j \triangleright E'[M_j[V/x_j]] : B \triangleleft \text{end} \).

Recomposing:

\[
\frac{\{ U_a \} \Gamma \mid S'_j \triangleright E[\text{connect } \ell_j(V) \text{ to } b \text{ as } q] : A \triangleleft \text{end}}{a : \text{Pid}(U_a) \in \Gamma}
\]

\[
\frac{\{ U_a \} \Gamma \mid S'_j \triangleright E[\text{connect } \ell_j(V) \text{ to } b \text{ as } q] : A \triangleleft \text{end}}{b : \text{Pid}(U_b) \in \Gamma}
\]

\[
\frac{\{ U_a \} \Gamma \mid S'_j \triangleright E[\text{connect } \ell_j(V) \text{ to } b \text{ as } q] : A \triangleleft \text{end}}{\{ U_a \} \Gamma \mid S'_j \triangleright E'[M_j[V/x_j]] : B \triangleleft \text{end}}
\]

\[
\frac{\{ U_a \} \Gamma \mid S'_j \triangleright E[\text{connect } \ell_j(V) \text{ to } b \text{ as } q] : A \triangleleft \text{end}}{\{ U_a \} \Gamma \mid S'_j \triangleright E'[M_j[V/x_j]] : B \triangleleft \text{end}}
\]

with \( \text{safe}(a : U_a, b : U_b, s[p](\bar{r}), q, S'_j, s[q](p) : T_j) \) as required.
REFERENCES

Case E-Comm

Let $\Delta_1 = a : U_a, s[p(\bar{r})] : S_a$ and $\Delta_2 = b : U_b, s[q(\bar{s})] : S_b$.

Assumption:

$$a : \mathrm{Pid}(U_a) \in \Gamma \quad \{U_a\} \quad \vdash S_a \triangleright E[\text{send} \, \ell_j(V)] \to q : A \triangleq \text{end}$$

$$b : \mathrm{Pid}(U_b) \in \Gamma \quad \{U_b\} \quad \Gamma \vdash \kappa_1$$

$$\Gamma ; \Delta_1 \vdash (a, E[\text{send} \, \ell_j(V)] \to q, s[p(\bar{r})], \kappa_1)$$

$$\Gamma ; \Delta_2 \vdash (b, E'[\text{receive from} \, p \{\ell_i(x_i) \to M_i\} \in E] : B \triangleq \text{end}$$

where $j \in I$ and safe($\Delta_1, \Delta_2$).

Consider the subderivation $\{U_a\} \quad \vdash S_a \triangleright E[\text{send} \, \ell_j(V)] \to q : A \triangleq \text{end}$. By Lemma 23:

$$\{U_a\} \quad \vdash V : A_j \quad \{U_a\} \quad \vdash \ell_j(V) \to q : 1 \triangleq S_j$$

Next, consider the subderivation $\{U_b\} \quad \vdash S_b \triangleright E'[\text{receive from} \, p \{\ell_i(x_i) \to M_i\} \in E] : B \triangleq \text{end}$. Again by Lemma 23:

$$\{U_b\} \quad \vdash (x_k : B_k, T_k : B, T_k) \triangleright \text{receive from} \, p \{\ell_i(x_i) \to M_i\} \in E : A \triangleq T'$$

From the assumption we know that $j \in I$. As a result of the two subderivations above, we can refine our definitions of $\Delta_1$ and $\Delta_2$:

$\Delta_1 = a : U_a, s[p(\bar{r})] : \Sigma_{i \in I}(S_i)$, where $\varrho(\ell_j(A_j)) : S_j \in \{S_i\}_{i \in I}$

$\Delta_2 = b : U_b, s[q(\bar{s})] : \Sigma_{k \in K}(p？\ell_k(B_k), T_k)$

Since safe($\Delta_1, \Delta_2$), we have that $j \in K$, $B_j = A_j$.

Thus we can construct a reduction on typing environments:

$$s[p(\bar{r})] : \Sigma_{i \in I}(S_i) \triangleright s[p(\bar{r})] : S_j'$$

$$s[q(\bar{s})] : \Sigma_{k \in K}(p？\ell_k(B_k), T_k) \triangleright s[q(\bar{s})] : S_j'$$

$$a : U_a, b : U_b, s[p(\bar{r})] : \Sigma_{i \in I}(S_i), s[q(\bar{s})] : \Sigma_{k \in K}(p？\ell_k(B_k), T_k) \triangleright a : U_a, b : U_b, s[p(\bar{r})] : S_j', s[q(\bar{s})] : T_j$$

Let $\Delta' = a : U_a, b : U_b, s[p(\bar{r})] : S_j', s[q(\bar{s})] : T_j$.

Since safe($\Delta_1, \Delta_2$) and $\Delta_1, \Delta_2 \rightarrow \Delta'$, by the definition of safety, safe($\Delta'$).

Noting that only top-level frames (i.e., $F, F'$) can contain exception-handling frames, $E, E'$ are pure. By Lemma 25, $\{U_a\} \quad \Gamma \vdash S_j' \triangleright E[\text{return}()] : A \triangleq \text{end}$. By Lemma 22, $\{U_b\} \quad \Gamma \vdash T_j \triangleright M_j[V/x_j] : B \triangleq T'$. By Lemma 25, $\{U_b\} \quad \Gamma \vdash T_j \triangleright E'[M_j[V/x_j]] : B \triangleq \text{end}$. Letting $\Delta'_1 = a : U_a, s[p(\bar{r})] : S_j'$; and $\Delta'_2 = b : U_b, s[q(\bar{s})] : T_j$; recomposing:

$$a : \mathrm{Pid}(U_a) \in \Gamma \quad \{U_a\} \quad \vdash \kappa_1$$

$$\Gamma ; \Delta'_1 \vdash (a, E[\text{return}()] \triangleq s[p(\bar{r})], \kappa_1)$$

$$b : \mathrm{Pid}(U_b) \in \Gamma \quad \{U_b\} \quad \vdash \kappa_2$$

$$\Gamma ; \Delta'_2 \vdash (b, E'[M_j[V/x_j]] \triangleq s[q(\bar{s})], \kappa_2)$$
with \( \text{safe}(\Delta') \), as required.

**Case E-Disconn**

\[
\langle a, E[\text{wait } q], s[p](\bar{r}, q), \kappa_1 \rangle \parallel \langle b, E'[\text{disconnect from } p], s[q](p), \kappa_2 \rangle \longrightarrow \langle a, E[\text{return }()], s[p](\bar{r}), \kappa_1 \rangle \parallel \langle b, E'[\text{return }()], \bot, \kappa_2 \rangle
\]

Let \( \Delta = a : U_a, s[p](\bar{r}) : S, b : U_b, s[q](p) : T \).

Assumption:

<table>
<thead>
<tr>
<th>Pid((U_a) \in \Gamma )</th>
<th>{U_a} \parallel S \triangleright E[\text{wait } q] : B \triangleleft \text{end} \quad {U_a} \parallel \kappa_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a : U_a, s[p](\bar{r}, q) : S \vdash \langle a, E[\text{wait } q], s<a href="%5Cbar%7Br%7D">p</a>, \kappa_1 \rangle )</td>
<td>( \Gamma \vdash \kappa_1 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pid((U_b) \in \Gamma )</th>
<th>{U_b} \parallel T \triangleright E'[\text{disconnect from } p] : B \triangleleft \text{end} \quad {U_b} \parallel \kappa_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( b : U_b, s<a href="p">q</a> : T \vdash \langle b, E'[\text{disconnect from } p], s<a href="p">q</a>, \kappa_2 \rangle )</td>
<td>( \Gamma \vdash \kappa_2 )</td>
</tr>
</tbody>
</table>

By Lemma 23:

\[
\{U_a\} \parallel \#\downarrow q \cdot S' \triangleright \text{wait } q \cdot 1 \triangleleft S'
\]

Also by Lemma 23:

\[
\{U_b\} \parallel \#\downarrow p \triangleright \text{disconnect from } p \cdot 1 \triangleleft T'
\]

Thus, \( S = \#\downarrow q \cdot S' \) and \( T = \#\downarrow q \cdot 1 \triangleleft T' \)

We can show a reduction on runtime typing environments:

\[
\begin{align*}
    & \text{s}[p](\bar{r}, q) : \#\downarrow q \cdot S' & \rightsquigarrow \text{s}[p](\bar{r}) : S'
    \\
    & \text{s}[q](p) : \#\downarrow p & \rightsquigarrow S'
    \\
    & a : U_a, s[p](\bar{r}, q) : \#\downarrow q \cdot S', b : U_b, s[q](p) : \#\downarrow p & \longrightarrow a : U_a, s[p](\bar{r}) : S', b : U_b
\end{align*}
\]

Since \( \text{safe}(\Delta) \), and \( \Delta \implies \Delta' \), by the definition of safety, we have that \( \text{safe}(\Delta') \).

Noting that only top-level frames (i.e., \( F, F' \)) can contain exception-handling frames, \( E, E' \) are pure. We can show that \( \{U_a\} \parallel S' \triangleright \text{return }() : 1 \triangleleft S' \), so by Lemma 25, we have that \( \{U_a\} \parallel S' \triangleright E[\text{return }()] : A \triangleleft S' \).

By the same argument, \( \{U_b\} \parallel b \triangleright E'[\text{return }()] : 1 \triangleleft \text{end} \).

Thus we can show (by T-UNCONNECTEDACTOR, noting that end is a permissible precondition):

\[
\begin{align*}
    & \text{P} \text{id}(U_a) \in \Gamma \\
    & \{U_a\} \parallel \text{end} \triangleright E'[\text{return }()] : B \triangleleft \text{end} \quad \{U_a\} \parallel \kappa_2 \\
    & \Gamma ; b : U_a \vdash \langle b, E'[\text{return }()] \rangle, \bot, \kappa_2
\end{align*}
\]

Recomposing:

\[
\begin{align*}
    & a : \text{P} \text{id}(U_a) \in \Gamma \\
    & \{T\} \parallel S \triangleright E[\text{wait } q] : B \triangleleft \text{end} \quad \{T\} \parallel \kappa_1 \\
    & \Gamma : a : U_a, s[p](\bar{r}) : S \vdash \langle a, E[\text{return }()], s[p](\bar{r}), \kappa_1 \rangle \\
    & b : U_b \in \Gamma \\
    & \{U_b\} \parallel \text{end} \triangleright E'[\text{return }()] : B \triangleleft \text{end} \quad \{U_b\} \parallel \kappa_2 \\
    & \Gamma : b : U_b \vdash \langle b, E'[\text{return }()] \rangle, \bot, \kappa_2
\end{align*}
\]

\[
\begin{align*}
    & \Gamma : a : U_a, s[p](\bar{r}) : S, b : U_b \vdash \langle a, E[\text{return }()], s[p](\bar{r}), \kappa_1 \rangle \parallel \langle b, E'[\text{return }()] \rangle, \bot, \kappa_2
\end{align*}
\]
REFERENCES

with safe(a : U_a, s[p](r):S', b : U_b), as required.

Case E-Complete

\( (\nu s)(⟨a, \text{return } V, s[p](\emptyset), \kappa⟩) \rightarrow ⟨a, \text{return } V, \bot, \kappa⟩ \)

Assumption:

\[
\begin{array}{c}
a : \text{Pid}(T) \quad \{T\} \quad S \triangleright \text{return } V : A \triangleleft \text{end} \quad \{T\} \quad \Gamma \vdash \kappa \\
\Gamma; a : T, s[p](\emptyset) : S \triangleright ⟨a, \text{return } V, s[p](\emptyset), \kappa⟩ \\
\Gamma; \vdash (\nu s)(⟨a, \text{return } V, s[p](\emptyset), \kappa⟩)
\end{array}
\]

Since by T-Return, the pre- and post-conditions must match, it must be the case that \( S = \text{end} \). Thus we can show:

\[
\begin{array}{c}
a : \text{Pid}(T) \quad \{T\} \quad \text{end} \triangleright \text{return } V : A \triangleleft \text{end} \quad \{T\} \quad \Gamma \vdash \kappa \\
\Gamma; a : T \vdash ⟨a, \text{return } V, s[p](\emptyset), \kappa⟩
\end{array}
\]

as required.

Case E-CommRaise

\( ⟨a, E[M], s[p](\hat{r}), \kappa⟩ \parallel \hat{q} s[q] \rightarrow ⟨a, E[\text{raise}], s[p](\hat{r}), \kappa⟩ \parallel \hat{q} s[q] \)

where subj(M) = p.

The proof is by cases on M, where M must be a communication action. The cases have the same structure, so we show the case where \( M = \text{wait } q \).

Assumption:

\[
\begin{array}{c}
\{T\} \quad \Gamma \vdash \kappa \\
\Gamma; a : T, s[p](\hat{r}) : \# q.S' \triangleright ⟨a, E[\text{wait } q], s[p](\hat{r}), \kappa⟩ \\
\Gamma; a : T, s[p](\hat{r}) : \# q.S'[s[q](\hat{s})] : U \vdash ⟨a, E[\text{wait } q], s[p](\hat{r}), \kappa⟩ \parallel \hat{q} s[q]
\end{array}
\]

By Lemma 23, there exists some \( S'' \) such that:

\[
\{T\} \quad \Gamma \vdash \kappa \\
\Gamma; a : T, s[p](\hat{r}) : \# q.S'[s[q](\hat{s})] : U \vdash ⟨a, E[\text{wait } q], s[p](\hat{r}), \kappa⟩ \parallel \hat{q} s[q]
\]

By T-Raise, \text{raise} can have any precondition, return type, and postcondition. Therefore, by Lemma 25, \{T\} \quad \Gamma \vdash \kappa \\
\begin{array}{c}
\{T\} \quad \Gamma \vdash \kappa \\
\Gamma; a : T, s[p](\hat{r}) : \# q.S'[s[q](\hat{s})] : U \vdash ⟨a, E[\text{raise}], s[p](\hat{r}), \kappa⟩ \parallel \hat{q} s[q]
\end{array}
\]

as required.

Case E-FailS
since \( \text{raise} \) is typable under any precondition and postcondition, and has an arbitrary return type, it follows that 
\[
\{T\} \quad \Gamma \mid S \triangleright P[\text{raise}]:A \triangleleft \text{end} \quad \{T\} \quad \Gamma \vdash \kappa
\]
Recomposing:
\[
\begin{align*}
\{T\} \quad \Gamma \mid \text{end} \triangleright \text{raise}:A \triangleleft \text{end} \quad \{T\} \quad \Gamma \vdash \kappa \\
\Gamma; a : T, s[p][\tilde{q}]:S \vdash \langle a, P[\text{raise}], s[p][\tilde{q}], \kappa \rangle
\end{align*}
\]
\[
\begin{align*}
\Gamma; a : T, s[p][\tilde{q}]:S \vdash \langle a, \text{raise}, s[p][\tilde{q}], \perp, \kappa \rangle \parallel \tilde{s}[p]
\end{align*}
\]

Case E-FailLoop

Similar to E-LOOP.

Case E-LiftM

Immediate by Lemma 20.

Case E-Equiv

Immediate by Lemma 21.

Case E-Par

Immediate by the IH and rules E-Cong1 and E-Cong2.

Case E-Nu

Immediate by the IH, noting that the IH ensures that the resulting environment is also safe.

\section{Progress}

\subsection{Overview}

The progress proof requires several steps. We overview them below.

**Canonical forms** Canonical forms allow us to reason globally about a configuration by putting it in a structured form. Lemma 15 states that any well-typed, closed configuration can be put into canonical form.

**Exception-aware runtime typing environments** Runtime typing environments do not account explicitly for zapper threads. This makes sense when analysing protocols statically to check their safety and progress properties, but is inconvenient when reasoning about configurations.

The second step is to introduce exception-aware runtime typing environments which account explicitly for zapper threads and the propagation of exceptions, and an exception-aware runtime typing system (Definition 28). We then show that configurations including zapper threads are typable under the exception-aware runtime typing system (Lemma 32).
REFERENCES

We refine our notion of environment progress to include the possibility of failed sessions (Definition 35), and show that given a runtime environment $\Delta$ satisfying safety and progress, a derived exception-aware runtime environment $\Theta$ also satisfies safety and progress (Lemma 36).

Flattenings Hu & Yoshida’s formulation of multiparty session types relaxes the directedness constraint for output choices, but this relaxation makes it more difficult to reason about reduction of environments being reflected by configurations.

In this step, we introduce flattenings (Definitions 39 and 40), which restrict each top-level output choice to a single option, and show that if a configuration is ready (i.e., all actors are blocked on communication actions), then it is typable under a flattened environment (Lemma 42). We then show that flattenings preserve environment reducibility (Lemma 43).

Session progress The penultimate step is to show that sessions can make progress. A key result is Lemma 45, which states that a ready configuration typable under a flat exception-aware typing environment can reduce. The session progress lemma (Lemma 18) follows from Lemma 45 and the previous results.

Progress Finally, progress follows from Lemma 18 and case analysis on the disconnected actors.

C.2 Auxiliary Definitions

Let $\Psi$ range over typing environments containing only runtime names: $\Psi := \cdot | \Psi, a : \text{Pid}(S)$.

Term reduction satisfies a form of progress: a well-typed term is a value, can reduce, or is a communication or concurrency construct:

$\triangleright$ Lemma 26 (Progress (terms)). If $\{T\} \Psi \vdash M : A \triangleleft S'$, then either $M = \text{return} V$ for some value $V$; there exists some $N$ such that $M \rightarrow_M N$; or there exists some $E$ such that $M$ can be written $E[N]$ for some $E, N$ where $N$ is either raise or an adaptation, communication, or concurrency construct.

$\triangleright$ Lemma 27 (Session typability). If $\cdot ; \cdot \vdash G[S]$, then there exist $\Psi, \Delta$ such that $\Psi; \Delta \vdash S$ and $\Delta$ only consists of entries of the form $a : S$.

Proof. By induction on the structure of $G$, noting that entries of the form $s[p](\bar{q}):S$ are linear; by the definition of $S = (\nu s)C$, actors and zapper threads in $C$ must refer to only to $s$.

C.3 Canonical forms

$\triangleright$ Lemma 15 (Canonical forms). If $\cdot ; \cdot \vdash C$, then $\exists D \equiv C$ where $D$ is in canonical form.

Proof sketch. Due to Lemma 21, by induction on typing derivations we can move all actor name restrictions to the top level, followed by all session name restrictions, followed by all connected actors, followed by all zapper threads, followed by all disconnected actors.

Since the typing rules ensure each actor only participates in a single session, we can then group each actor and zapper thread according to its session, in order to arrive at a canonical form.

C.4 Exception-aware runtime typing environments

Due to E-RAISEP, zapper threads expose additional reductions to those allowed by standard environment reduction. In particular, given the presence of a zapper thread $\xi s[p]$, any other participant in the remainder of the session blocked on $p$ can reduce. In order to account for this, we introduce the notion of exception-aware runtime typing environments, environment reduction, and runtime typing.

$\triangleright$ Definition 28 (Exception-aware runtime typing environments, environment reduction, and runtime typing). An exception-aware runtime typing environment $\Theta$ is defined as follows:

$$\Theta := a : S | s[p](\bar{q}):S | s[p] : \xi$$

The exception-aware runtime typing relation $\Gamma; \Theta \vdash \cdot C$ is defined to be the standard runtime typing relation $\Gamma; \Delta \vdash C$ but with rule $T-ZAP$ defined as:
Proof. A straightforward induction on the derivation of \( \Delta \Rightarrow \Delta' \) with the addition of the following rules:

- **ET-Comfail**
  \[
  \exists j. \Gamma \vdash S_j = p \cdot t_j(A_j) \cdot S_j'
  \]

- **ET-Disconnfail**
  \[
  \Gamma \vdash S'[\mathcal{q}] : \# q \rightarrow s[p] : \delta, s[q] : \delta
  \]

- **ET-Waitfail**
  \[
  \Gamma \vdash S : \delta, s[q] : \delta \rightarrow s[p] : \delta
  \]

**Definition 29** (Zapped roles). Define the zapped roles of session

\[
S = (\nu s)\langle (a_1, M_1, s[p_1](\mathcal{q}_1), \kappa_1) \rangle \cdots \langle (a_m, M_m, s[p_m](\mathcal{q}_m), \kappa_m) \rangle \| s[p_{m+1}] \| \cdots \| s[p_n] \rangle
\]

as \( p_{m+1}, \ldots, p_n \).

**Definition 30** (Zapped environment). Given a runtime environment

\[
\Delta = a_1 : S_1, \ldots, a_i : S_i, s[p_1](\mathcal{r}_1) : T_1, \ldots, s[p_m](\mathcal{r}_m) : T_m, s[q_1](\mathcal{r}_1) : T'_1, \ldots, s[q_n](\mathcal{r}_n) : T'_n
\]

and a set of roles \( \mathcal{q} = q_1, \ldots, q_n \), the zapped environment \( \Theta = \text{zap}(\Delta, \mathcal{q}) \) is defined as:

\[
\Delta = a_1 : S_1, \ldots, a_i : S_i, s[p_1](\mathcal{r}_1) : T_1, \ldots, s[p_m](\mathcal{r}_m) : T_m, s[q_1](\mathcal{r}_1) : T'_1, \ldots, s[q_n](\mathcal{r}_n) : T'_n
\]

**Definition 31** (Failed environment). An exception-aware runtime typing environment \( \Theta \) is a failed environment, written \( \text{failed}(\Theta) \), if \( \Theta \) is of the form \( s[p_1] : \delta, \ldots, s[p_n] : \delta \).

**Lemma 32** (Zapped Typeability). If \( \Gamma; \Delta \vdash S \) where \( S = (\nu s) \Delta' C \), the zapped roles of \( S \) are \( \mathcal{p} \), and \( \Theta = \text{zap}(\Delta', \mathcal{p}) \), then \( \Gamma; \Delta, \Theta \vdash C \).

Proof. A straightforward induction on the derivation of \( \Gamma; \Delta \vdash C \), noting that the definition of sessions and T-Session ensures that \( \Delta \) only contains actor runtime names, and that the modified T-Zap rule is satisfied by the fact that \( \Theta = \text{zap}(\Delta, \mathcal{p}) \).

**Corollary 33.** If \( \vdash C \) where \( C \) is in canonical form, then \( \vdash C \).

Proof. Follows directly by applying Lemma 32 to each session.

**Corollary 34** (Exception-aware session typing). If \( \vdash \mathcal{S} \), then there exist \( \Psi, \Delta \) such that \( \Delta \) only consists of entries of the form \( a : S \) and \( \vdash \mathcal{S} \).

Definition 12 extends straightforwardly to exception-aware runtime typing environments.

**Definition 35** (Progress (exception-aware environments)). An exception-aware runtime typing environment \( \Theta \) satisfies progress, written \( \text{prog}(\Theta) \), if:

- (Role progress) for each \( s[p_1](\mathcal{q}_1) : S_1 \) \( \in \Theta \) such that \( \text{active}(S_1) \), it is the case that \( \Theta \rightarrow^* \Theta' \mathcal{q} \rightarrow \) with \( p \in \text{roles}(\gamma) \).
- (Eventual communication) if \( \Theta \rightarrow^* \Theta' \mathcal{q} \rightarrow \Theta'' \mathcal{q} \rightarrow \Theta''' \mathcal{q} \rightarrow \), then either \( \Theta' \mathcal{q} \rightarrow \Theta'' \mathcal{q} \rightarrow \Theta''' \mathcal{q} \rightarrow \), or \( \Theta' \rightarrow^* \), where \( p \notin \text{roles}(\gamma) \).
- (Correct termination) \( \Theta \rightarrow^* \Theta' \mathcal{q} \rightarrow \) implies either \( \text{end}(\Theta) \) or \( \text{failed}(\Theta) \).
REFERENCES

If a runtime typing environment is deadlock-free, then any corresponding zapped environment will be deadlock-free.

Lemma 36 (Exception-aware environments preserve safety and progress). Suppose \( \text{safe}(\Delta) \) and \( \text{prog}(\Delta) \) for some \( \Theta \).

If \( \Theta = \text{zap}(\Delta, \vec{p}) \) for some set of roles \( \vec{p} \), then \( \text{safe}(\Theta) \) and \( \text{prog}(\Theta) \).

Proof sketch.

Safety. Safety follows since exception-aware environments do not modify session types, but only replace them with zappers. Interaction with a zapper is vacuously safe, so it follows that if \( \text{safe}(\Delta) \) then \( \text{safe}(\Theta) \).

Progress. For (role progress), we know that any active role in \( \Delta \) will eventually reduce. Presence of a zapped role means that any other role in the session trying to reduce with the zapped role will itself be zapped due to ET-CommFail, ET-Disconnect, or ET-WaitFail; connections will not occur, but this does not matter since the accepting role will not be in the session.

For (eventual communication), we have that if \( \Delta \xrightarrow{\ast} \Delta' \xrightarrow{s[p]:t(A)} \Delta'' \), then \( \Delta' \xrightarrow{p} \Delta'' \xrightarrow{s[q]:t(A)} \Delta''' \), where \( p \notin \text{roles}(\vec{p}) \).

Suppose \( \Theta \xrightarrow{\ast} \Theta' \xrightarrow{s[p]:t(A)} \). If \( \Theta' \) contains zapped roles, either they are irrelevant to \( p \)'s reduction and we can use the original reduction sequence to show \( \Theta \xrightarrow{\ast} \Theta'' \xrightarrow{s[q]:t(A)} \Delta''' \), or they are relevant to reduction and the exception propagates, resulting in \( \Theta' \xrightarrow{p} \Theta'' \xrightarrow{s[q]} \Delta''' \).

For (correct termination), we know that \( \Delta \equiv \ast \) implies that \( \text{end}(\Delta) \). By (1), we know that each active role in \( \Theta \) will eventually reduce, so it follows that either \( \text{end}(\Theta) \) (if all communications occur successfully) or \( \text{failed}(\Theta) \) if a role has failed and the failure has propagated to all other participants.

C.5 Flattenings

Definition 37 (Input / Output-Directed Choices). A choice session type \( \Sigma_{i \in I}(\alpha_i \cdot S_i) \) is a output-directed if each \( \alpha_i \) is either of the form \( p!f(A_i) \).

A choice session type is input-directed if it is of the form \( \Sigma_{i \in I}(p\downarrow f(A_i)) \) for some \( \dagger \in \{?, ???\} \).

For convenience, we write \( S^* \) and \( \Sigma^*_{i \in I}(\alpha_i \cdot S_i) \) to denote an output-directed choice, and \( S^? \) and \( \Sigma^?_{i \in I}(\alpha_i \cdot S_i) \) to denote an input-directed choice.

Definition 38 (Output-flat session type). An output-directed session type \( \Sigma^?_{i \in I}(S_i \cdot S_i) \) is output-flat if it is a unary choice, i.e., can be written \( p!f(A) \cdot S \) or \( p!f(A) \cdot S \).

An exception-aware runtime typing environment \( \Theta \) is output-flat, written \( \text{flat}(\Theta) \), if each output-directed choice type in \( \Theta \) is output-flat.

We now define a flattening of an output choice. A session type is a flattening of an output sum if it is a unary sum consisting of one of the choices.

Definition 39 (Flattening (types)). A session type \( S' \) is a flattening of an output-directed choice \( S = \Sigma^?_{i \in I}(\alpha_i \cdot T_i) \) if \( S' = \alpha_j \cdot T_j \) for some \( j \in I \).

We can extend flattening to environments by flattening all output-directed choices.

Definition 40 (Flattening (environments)). Given an exception-aware runtime typing environment:

\[
\Theta = s[p_1(\bar{r}_1);S^*_1, \ldots, s[p_1(\bar{r}_1);S^*_1, \ldots, s[q_1(\bar{s}_1);T_1, \ldots, s[q_m(\bar{s}_m);T_m, s[t_1]: \dagger, \ldots, s[t_n]: \dagger
\]

where each \( T \) is not an output-directed choice, we say that \( \Theta' \) is a flattening of \( \Theta \) if:

\[
\Theta' = s[p_1(\bar{r}_1);S'^*_1, \ldots, s[p_1(\bar{r}_1);S'^*_1, \ldots, s[q_1(\bar{s}_1);T_1, \ldots, s[q_m(\bar{s}_m);T_m, s[t_1]: \dagger, \ldots, s[t_n]: \dagger
\]

where each \( S'^*_i \) is a flattening of \( S^*_i \).

Progress states that however an environment reduces, an active role will be eventually able to reduce. In the case that the environment does not reduce, it is final.

Definition 41 (Ready). A configuration \( C \) is ready, written \( \text{ready}(C) \), if all subconfigurations are either zapper threads or actors evaluating terms of the form \( E[M] \) where \( M \) is a communication action.
Lemma 42 (Flattening typability). If \( \vdash \cdot : \Gamma[S] \) where \( S = (\nu s : \Theta)C \) and \( \text{ready}(C) \), then there exists some \( \Theta' \) such that \( \text{flat}(\Theta') \), where \( \Theta' \) is a flattening of \( \Theta \), and \( \vdash \cdot : \Gamma[(\nu s : \Theta')C] \).

**Proof.** By Lemma 34, we have that there exist \( \Psi, \Delta \) such that \( \Delta \) does not contain session entries, and \( \Psi; \Delta \vdash \cdot : S \).

By T-SESSION, \( \Psi; \Delta, \Theta \vdash \cdot : C \).

The result follows by induction on the derivation of \( \Psi; \Delta, \Theta \vdash \cdot : C \). In the case of T-ZAP, the result follows immediately.

In the case of E-ACTOR, we have that:

1. \( C = (a, M, s[p] (\bar{r}), \kappa) \)
2. \( \Psi; a : T, s[p] (\bar{r}) : S \vdash \cdot : C \)
3. \( \{T\} \Psi \mid S \triangleright M : A \iff \end \)

Since \( \text{ready}(C) \), it must be the case that the actor is evaluating a term of the form \( E[N] \) where \( N \) is a communication action.

By Lemma 23, \( \{T\} \Psi \mid S \triangleright N : B \iff S' \) for some \( B, S' \).

In the case that \( N \) is a disconnect, wait, receive, or accept action, then the session type cannot be output-directed and the result follows immediately.

In the case that \( M = \text{connect} \ell_j(V_j) \to W \) as \( q \), by T-COMM it must be the case that \( S = \Sigma^* \chi_j(A_j) \in \{\alpha_i \in E \} \) where \( q || \ell_j(A_j) \in \{\alpha_i \in E \} \) and \( \Psi \vdash V_j ; A_j \).

Again by T-COMM and Lemma 24, we can show that \( \{T\} \Psi \mid q || \ell_j(A_j).S' \triangleright E[\text{connect} \ell_j(V_j) \to W \) as \( q || B \iff S' \), where \( q || \ell_j(A_j).S' \) is a flattening of \( S \), as required.

The same argument holds for send actions.

Lemma 43 (Flattening preserves reducibility). Suppose safe(\( \Theta \)), prog(\( \Theta \)), and \( \Theta \longrightarrow \).

If \( \hat{\Theta} \) is a flattening of \( \Theta \), then \( \hat{\Theta} \longrightarrow \).

**Proof.** Assume for the sake of contradiction that \( \hat{\Theta} \not\longrightarrow \).

For this to be the case, each \( \rho \) such that \( \Theta ~\sim\rightarrow \) must either be \( s[p], q : \ell \) or \( s[p] \longrightarrow q : \ell \), but instead, \( \hat{\Theta} \xrightarrow{s[p]q':\ell'(A)} \) for some \( q' \neq q \). Note that \( \rho \) cannot be a disconnect action, since these are unaffected by flattening, and it cannot be the case that \( \hat{\Theta} \xrightarrow{s[p]q':\ell'(A)} \), since this is a synchronisation action and could reduce.

By (eventual communication), either \( \Theta \xrightarrow{\rho \hat{\Theta}} \Theta' \xrightarrow{s[p]q':\ell'(A)} \) or \( \Theta \xrightarrow{\rho \hat{\Theta}} \Theta' \xrightarrow{q' \hat{\Theta}} \).

Pick some \( \rho \) such that \( \rho \hat{\Theta} \) contains no roles affected by flattening; one such \( \rho \) must exist since otherwise there would be a cycle, contradicting progress.

It follows that either:

\[
\begin{align*}
\hat{\Theta} &\xrightarrow{\rho \hat{\Theta}} \Theta' \xrightarrow{s[p]q':\ell'(A)} \xrightarrow{s[q']p':\ell'(A)} \text{ meaning } \hat{\Theta} \xrightarrow{\rho \hat{\Theta}} \Theta' \xrightarrow{s[p]q':\ell'}, \text{ a contradiction; or} \\
\hat{\Theta} &\xrightarrow{\rho \hat{\Theta}} \Theta' \xrightarrow{q' \hat{\Theta}}, \text{ also a contradiction.}
\end{align*}
\]

C.6 Session progress

Lemma 44 (Readiness). Suppose \( \vdash \cdot : \Gamma[S] \) where \( C \subseteq \Gamma[S] \), \( C \) does not contain unmatched discovers, \( S \) is not a failed session, \( S = (\nu s : \Theta)D \), and \( \text{prog}(\Theta) \).

Either \( C \longrightarrow \) or \( \text{ready}(D) \).

**Proof.** For each \( (a, M, s[p] (\bar{q}), \kappa) \), by Lemma 26, \( M \) can either reduce or either a value, adaptation construct, or communication construct.

- If \( M \) can reduce, then the configuration can reduce by E-LIFTM.
- If \( M = E[\text{raise}] \), then the configuration either can reduce by E-TRYRAISE and E-LIFTM, or E-FAILS.
- If \( M \) is an adaptation action, due to the absence of unmatched discovers, the configuration can reduce by E-NEW, E-REPLACE, E-REPLACESelf, E-DISCOVER, or E-Self.
REFERENCES

- If \( M \) is a value, then by T-\text{ConnectedActor}, \( \{ T \} \Psi \rightarrow \mathsf{return} V : A \diamond \mathsf{end} \). As a consequence of \( \mathsf{prog}(\Theta) \), \( a \)
must be the only actor in the session and thus the configuration could reduce by \text{E-COMPLETE}.

- It follows that all terms must be evaluating communication actions, satisfying \( \mathsf{ready}(\mathcal{D}) \).

\begin{lemma}[Exception-aware session progress] If \( \odot \vdash \bot \mathcal{C} \) where:
\begin{enumerate}
\item \( \mathcal{C} \equiv \mathcal{G}[\mathcal{S}] \),
\item \( \mathcal{S} = (\nu s : \Theta)\mathcal{D} \),
\item \( \mathsf{flat}(\Theta) \),
\item \( \mathsf{ready}(\mathcal{C}) \),
\item \( \Theta \Rightarrow \mathcal{C} \)
\end{enumerate}
then \( \mathcal{C} \rightarrow \).\end{lemma}

\begin{proof}
Since \( \Theta \Rightarrow \), there exists some environment \( \Theta' \) and label \( \rho \) such that \( \Theta \vdash^{\rho} \Theta' \).

The proof is by induction on the derivation of \( \Theta \vdash^{\rho} \Theta' \).

**Case ET-Conn**

\[
\exists j \in I, s_j = q \llcorner \ell_j(A_j) \cdot S_j'
\]

Since \( \mathsf{flat}(\Theta) \), we have that \( \Theta \vdash^{\rho} \Theta' \). Hence, \( \mathcal{C} \) must contain an actor evaluating \( E[\mathsf{connect} \ \ell_j(A_j) \rightarrow V] \) for some actor name \( b \). Due to the definition of canonical forms, actor \( b \) must be a subconfiguration of \( \mathcal{G}[\mathcal{S}] \) and thus the configuration can reduce by either \text{E-CONN} or \text{E-CONNFAIL}.

**Case ET-Comm**

\[
\Theta_1 \vdash^{\rho} \Theta_2
\]

By ET-ACT, \( \Theta_1 \) must contain \( s[p]\langle \tilde{r} \rangle : \Sigma \in I(\beta, I) \) since \( \mathsf{flat}(\Theta) \), \( \mathcal{S} = q \llcorner \ell(A) \).

Also by ET-ACT, \( \Theta_2 \) must contain \( s[p]\langle \tilde{r} \rangle : \Sigma \in I(\beta, I) \) with \( \mathsf{ll}(\ell(A)) \in \{ \beta \} \).

Thus by typing, \( \mathcal{C} \) must contain an actor \( (a, E[\mathsf{send} \ \ell(V) \rightarrow q], s[p]\langle \tilde{r} \rangle, \kappa_a) \), and another actor \( (b, E' \mathsf{receive} \ {p \{ \ell_i(x_i) \rightarrow M_i \}}, s[q]\langle \tilde{z} \rangle, \kappa_b) \), which would reduce by E-Comm.

**Case ET-CommFail**

\[
\exists j \in I, s_j = p \llcorner \ell_j(A_j) \cdot S_j'
\]

We prove the case where \( \uparrow = ! \). Since \( \mathsf{flat}(\Theta) \), by typing we have that \( \Sigma \in I(\alpha, I) \) and \( \mathcal{C} \) must contain an actor \( (a, E[\mathsf{send} \ \ell(V) \rightarrow q], s[p]\langle \tilde{r} \rangle, \kappa_a) \) and a zapper thread \( \{ s[q]\langle \tilde{r} \rangle, \kappa_b \} \), which could then reduce by E-RAISEP.

Case E-DISCONN is similar to E-CONN, and cases E-DISCONNFAIL and E-WAITFAIL are similar to E-CommFail. Cases ET-REC, ET-CONG1, and ET-CONG2 follow by the induction hypothesis.

\begin{lemma}
If \( \mathsf{prog}(\Theta) \) and neither \( \mathsf{end}(\Theta) \) nor \( \mathsf{failed}(\Theta) \), then \( \Theta \Rightarrow \).\end{lemma}
Proof. By contradiction. Assume that \( \text{prog}(\Theta) \) and neither \( \text{end}(\Theta) \) nor \( \text{failed}(\Theta) \).

By \( \text{prog}(\Theta) \), each active role must eventually reduce.

If no roles are active, then \( \text{end}(\Theta) \) or \( \text{failed}(\Theta) \): a contradiction.

Otherwise, by the definition of \( \text{prog}(\Theta) \) for each \( s[p](q):S \) there must exist some \( p \) such that \( \Theta \xrightarrow{p} \Theta' \Rightarrow \) where \( p \in \text{subj}(\rho) \): a contradiction.

\[\textbf{Lemma 47.} \text{ If } \vdash \exists D \text{ such that } D \equiv (\nu s : \Theta)C \text{ and } \text{ready}(C), \text{ then } \Theta \text{ is not final.} \]

Proof. By contradiction. If \( \Theta \) were final, it would consist only of session types of type \( \text{end} \). No communication actions are typable under \( \text{end} \), contradicting \( \text{ready}(C) \).

\[\textbf{Lemma 18 (Session Progress).} \text{ If } \vdash \exists C \text{ where } C \text{ does not contain an unmatched discover, } C \equiv \text{G}[S] \text{ and } S = (\nu s : \Delta)D \text{ with } \text{prog}(\Delta), \text{ and } S \text{ is not a failed session, then } C \rightarrow.\]

Proof. By Lemma 27, \( \exists \Psi, \Delta' \) such that \( \Psi; \Delta' \vdash S \).

By definition, \( S = (\nu s : \Delta)D \), where \( D = \langle a_1, M_1, s[p_1](q_1), \kappa_1 \rangle \parallel \cdots \parallel \langle a_m, M_m, s[p_m](q_m), \kappa_m \rangle \parallel \langle s[p_{m+1}] \parallel \cdots \parallel \overline{i} s[p_n] \rangle \).

By \( \text{T-SESSION} \), \( \text{safe}(\Delta) \).

By definition, the zapped roles of \( C \) are \( p_{m+1}, ..., p_n \). Let us denote this set as \( \overline{p} \).

Let \( \Theta = \text{zap}(\Delta, \overline{p}) \).

By Lemma 32, \( \Psi; \Delta', \Theta \vdash D \) and so by Corollary 33, \( \vdash \exists \Gamma \text{G}[S] \).

By Lemma 36, \( \text{safe}(\Theta) \) and \( \text{prog}(\Theta) \).

By Lemma 44, either \( C \rightarrow \) or \( \text{ready}(D) \). As \( C \rightarrow \) satisfies the theorem statement, we proceed assuming \( \text{ready}(D) \).

By Lemma 42, there exists some \( \Theta' \) such that \( \Theta' \) is a flattening of \( \Theta \) and \( \vdash \exists \Gamma \text{G}[(\nu s : \Theta')D] \).

By Lemma 47, \( \Theta' \) is not final.

By Lemma 43, \( \Theta' \rightarrow . \)

Thus, by Lemma 45, \( C \rightarrow \) as required.
by global assumption, all protocols satisfy progress
it must be the case that either $a$ is an initiator of a session and blocked on connect, which could either reduce by E-CONNINIT or E-CONNFAIL, or be accepting, as required.