Multiparty Session Types for Safe Runtime Adaptation in an Actor Language

Paul Harvey[1]
Rakuten Institute of Technology
paul@paul-harvey.org

Simon Fowler[1]
School of Computing Science, University of Glasgow
Simon.Fowler@glasgow.ac.uk

Ornela Dardha
School of Computing Science, University of Glasgow
Ornela.Dardha@glasgow.ac.uk

Simon J. Gay
School of Computing Science, University of Glasgow
Simon.Gay@glasgow.ac.uk

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The era of single monolithic stand-alone computers has long been replaced by a landscape of heterogeneous and distributed computers and software applications.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],

[1] Equal contribution.

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^{X} [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^{X} 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 π-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`

Global protocol

```
global protocol Bookstore
    (role Sell, role Buy1, role Buy2) {
  book(string) from Buy1 to Sell;
  book(int) from Sell to Buy1;
  quote(int) from Buy1 to Buy2;
  choice at Buy2 {
    agree(string) from Buy2 to Buy1, Sell;
    transfer(int) from Buy1 to Sell;
    transfer(int) from Buy2 to Sell;
  } or {
    quit(string) from Buy2 to Buy1, Sell;
  } }
```

Local protocol for `Sell`

```
local protocol Bookstore_Sell
    (self Sell,role Buy1,role Buy2) {
  book(string) from Buy1;
  book(int) to Buy1;
  choice at Buy2{
    agree(string) from Buy2;
    transfer(int) from Buy1;
    transfer(int) from Buy2;
  } or {
    quit(string) from Buy2;
} }
```

Figure 1 Global and local protocols for Bookstore

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.

```
1  explicit global protocol OnlineStore
2      (role Customer, role Store, role Courier) {
3    login(string) connect Customer to Store;
4    do Browse(Customer, Store, Courier);
5  }
6
7  aux global protocol Deliver
8      (role Customer, role Store, role Courier) {
9    address(string) from Customer to Store;
10   deliver(string) connect Store to Courier;
11   ref(int) from Courier to Store;
12   disconnect Courier and Store;
13   ref(int) from Store to Customer;
14   disconnect Store and Customer;
15 }
```

```
1  aux global protocol Browse
2      (role Customer, role Store, role Courier) {
3    item(string) from Customer to Store;
4    price(int) from Store to Customer;
5    choice at Customer {
6      do Browse(Customer, Store, Courier);
7    } or {
8      do Deliver(Customer, Store, Courier);
9    } or {
10     quit() from Customer to Store;
11     disconnect Store and Customer;
12   }
13 }
```

Figure 2 Global protocol for `OnlineStore`

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 *role*s 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

```
1  type Isnd is interface(out integer output)
2  type Ircv is interface(in integer input)
3
4  stage home {
5   actor sender presents Isnd {
6    value = 1;
7    constructor() {}
8    behaviour {
9     send value on output;
10    value := value + 1;
11  } }
12  actor receiver presents Ircv {
13   constructor() {}
14   behaviour{
15    receive data from input;
16    printString("\nreceived:␣");
17    printInt(data);
18  } }
19   boot {
20     s = new sender();
21     r = new receiver();
22     establish topology(s, r);
23   }
24 }
```

Figure 3 A simple EnsembleS program

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

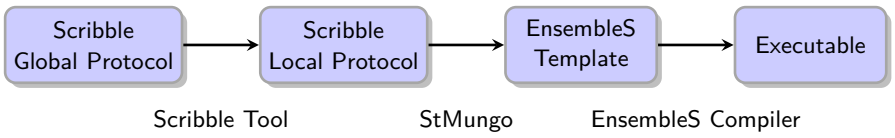

Figure 4 Automatic Actor Skeleton Generation Process

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`.

```
1  // FILE AUTOMATICALLY GENERATED
2  //************SESSIONS**************
3  type Buy1 is session(
4   book(string) to Sell;
5   book(int) from Sell;
6   quote(int) to Buy2;
7   choice at Buy2{
8    Choice0_agree(string) from Buy2;
9    transfer(int) to Sell;
10  } or {
11   Choice0_quit(string) from Buy2;
12  } )
13 //***********INTERFACES*************
14 type Buy1I is interface(
15  out {Seller, string} toSell_string,
16  in {Seller, integer} fromSell_integer,
17  out {Buy2, integer} toBuy2_integer,
18  in {Buy2, Choice0} fromBuy2_agreequit,
19  in {Buy2, string} fromBuy2_string,
20  out {Sell, integer} toSell_integer,
21 )
22 //*************ACTORS***************
23 stage home{
24 actor Buy1A presents Buy1I follows Buy1 {
25  constructor() {}
26  behaviour {
27   payload1 = "";
28   send payload1 on toSell_string;
29   receive payload2 from fromSell_integer;
30   payload3 = 42;
31   send payload3 on toBuy2_integer;
32   // Receive choice from other actor
33   receive payload4 from fromBuy2_agreequit;
34   switch(payload4) {
35    case Choice0_agree:
36     receive payload5 from fromBuy2_string;
37     payload6 = 42;
38     send payload6 on toSell_integer;
39     break;
40   case Choice0_quit:
41     receive payload7 from fromBuy2_string;
42     break;
43   }
44 } }
45 // Omitted: Buy2A and SellA actors
46 boot {
47  buyer1 = new Buy1A();
48  buyer2 = new Buy2A();
49  seller = new SellA();
50  // other actors...
51  establish topology(buyer1,buyer2,seller);
52 } }
```

Figure 5 EnsembleS static session template

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 `session`s 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 `session`s (line 6, Fig. 6). It connects all of the channels of the two specified actors such that the channel and actor's `session`s match. If the `session`s 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).

```
1  // session and interface definitions
2  actor fastA presents accountingI
3        follows accountingSession{
4   constructor() {}
5   behaviour{
6    receive data on input;
7    quicksort(data);
8    send data on output;
9    }
10 }
11
12 actor slowA presents accountingI
13       follows accountingSession{
14  pS= new property[2] of property("",0);
15  constructor() {
16   pS[0]:= new property("serial",823);
17   pS[1]:= new property("version",2);
18   publish pS;
19  }
20  behaviour{
21   receive data on input;
22   bubblesort(data);
23   send data on output;
24 } }
25
26 query alpha() { $serial==823 && $version<4; }
27
28 actor main presents mainI {
29  constructor() { }
30  behaviour {
31  // Find the slow actors matching query
32  actor_s = discover(accountingI,
33     accountingSession, alpha());
34  // Replace them with efficient versions
35  if(actor_s[0].length > 1) {
36     replace actor_s[0] with fastA();
37  }
38 } }
```

Figure 7 Session type-based replacement

3.4 Adaptation via discovery and replacement

```
1  // define query alpha
2  query_a = alpha();
3  actor_s = discover(
4   Buyer1_interface, Buyer1_session, query_a);
5  if (actor_s[0].length > 1){
6    link me with actor_s[0];
7   msg = "book";
8   send msg on toB_string;
9    unlink Buyer1_session;
10 }
```

Figure 6 Session type-based discovery

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.

```
type Iclient is interface(
 out{RootServer,string} RootServer_stringOut,
 in {RootServer,string} RootServer_stringIn,
 out{ZoneServer,string} ZoneServer_stringOut,
 in {ZoneServer,string} ZoneServer_stringIn,
 in {ZoneServer,choice_enum} ZoneServer_choiceIn,
 in {RootServer,choice_enum} RootServer_choiceIn)

type choice_enum is
 enum(TLDResponse,PartialResolution,
 InvalidDomain,ResolutionComplete,
 InvalidTLD)

query find_name(string n){ $name == n; }

actor c presents Iclient
 follows Client {
 dom_name = "nii.ac.jp";
 constructor() { }
 behaviour{
  rootQuery = find_name("jp");
  // Find Root Server
  root_s =
   discover(IServer, RootServer, rootQuery);
  // search until root_s non-empty
  link me with root_s[0];
  send domain_name on RootServer_stringOut;
  receive c_msg from RootServer_choiceIn;
  switch(c_msg){
   case TLDResponse:
    receive ZoneServerAddr_msg
     from RootServer_stringIn;
   unlink RootServer;
while(true) Lookup : {
 // Find ZoneServer
 zone_s =
    discover(IServer, ZoneServer,
      find_name(ZoneServerAddr_msg));
 link me with zone_s[0];
 // Ask ZoneServer
 send dom_name on ZoneServer_stringOut;
 receive c_msg2 from ZoneServer_choiceIn;
 switch(c_msg2){
  case PartialResolution:
   receive str_msg from ZoneServer_stringIn;
   ZoneServerAddr_msg := str_msg;
   unlink ZoneServer;
   continue Lookup;
  case InvalidDomain:
   receive str_msg from ZoneServer_stringIn;
   unlink ZoneServer;
   break;
  case ResolutionComplete:
   receive str_msg from ZoneServer_stringIn;
   unlink ZoneServer;
   break Lookup;
  }
  // keep looking
 }
case InvalidTLD:
  receive str_msg from RootServer_stringIn;
  unlink RootServer;
} } }
```

Figure 9 EnsembleS DNS client

4 Case study: DNS

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].

```
type Client is session(
  connect RootServer;
  RootRequest(DomainName) to RootServer;
  choice at RootServer{
    TLDResponse(ZoneServerAddress) from RootServer;
    disconnect RootServer;
    rec Lookup {
      connect ZoneServer;
      ResolutionRequest(DomainName) to ZoneServer;
      choice at ZoneServer {
        PartialResolution(ZoneServerAddress)
          from ZoneServer;
        disconnect ZoneServer;
        continue Lookup;
      } or {
        InvalidDomain(String) from ZoneServer;
        disconnect ZoneServer;
      } or {
        ResolutionComplete(IPAddress) from ZoneServer;
        disconnect ZoneServer;
      }
    }
  } or {
    InvalidTLD(String) from RootServer;
    disconnect RootServer;
  })
```

Figure 8 EnsembleS DNS client `session` type

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 `link`s with it (lines 27 and 39), enabling communication. When communication with the server is no longer required, the client `unlink`s 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 **actor** u **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	$::=$	**actor** u **follows** S $\{M\}$
Recursion Labels	l		
Behaviours	κ	$::=$	M | **stop**
Types	A, B	$::=$	$\mathsf{Pid}(S)$ | $\mathbf{1}$
Values	V, W	$::=$	x | $()$
Actions	L	$::=$	**return** V | **continue** l | **raise**
		|	**new** u | **self** | **replace** V **with** κ | **discover** S
		|	**connect** $\ell(V)$ **to** W **as** p | **accept from** p $\{\ell_i(x_i) \mapsto M_i\}_i$
		|	**send** $\ell(V)$ **to** p | **receive from** p $\{\ell_i(x_i) \mapsto M_i\}_i$
		|	**wait** p | **disconnect from** p
Computations	M, N	$::=$	**let** $x \Leftarrow M$ **in** N | **try** L **catch** M | $l :: M$ | L

Syntax of Session Types

Session Actions	α, β	$::=$	$\mathsf{p}!\ell(A)$ | $\mathsf{p}!!\ell(A)$ | $\mathsf{p}?\ell(A)$ | $\mathsf{p}??\ell(A)$ | $\#{\uparrow}\mathsf{p}$
Session Types	S, T, U	$::=$	$\Sigma_{i\in I}(\alpha_i \,.\, S_i)$ | $\mu X.S$ | X | $\#{\downarrow}\mathsf{p}$ | 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 **let** $x \Leftarrow M$ **in** N construct evaluates M, binding its result to x in N. The calculus supports exception handling over a single *action* L using **try** L **catch** M, where M is evaluated if L raises an exception, and labelled recursion using $l :: M$, stating that inside term M, a process can recurse to label l using **continue** l. *Actions* L denote the basic steps of a computation. The **return** V construct denotes a value.

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

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

Types. Types, ranged over by A, B, include the unit type $\mathbf{1}$ and process IDs $\mathsf{Pid}(S)$; the parameter S refers to the statically-known initial session type of the actor (i.e., the session type declared in the **follows** 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 $\Sigma_{i\in I}(\alpha \,.\, S)$, a recursive session type $\mu X.S$ binding recursion variable X in continuation S, a recursion variable X, a disconnection action $\#{\downarrow}\mathsf{p}$, or the finished session 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 α involve sending (!), receiving (?), connecting (!!), or accepting (??) a message $\ell(A)$ with label ℓ 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:

▶ **Definition 1** (Syntactic validity). *A choice type $S = \Sigma_{i\in I}(\alpha_i \,.\, S_i)$ is* syntactically valid *if:*

1. *it is an output choice, i.e., each α_i is a send or connection action; or*
2. *it is a* directed *input choice, i.e., $S = \Sigma_{i\in I}(\mathsf{p}?\ell_i(A_i).S_i)$ or $S = \Sigma_{i\in I}(\mathsf{p}??\ell_i(A_i).S_i)$; or*
3. *the choice consists of single wait action $\#{\uparrow}\mathsf{p} \,.\, S$.*

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 $\#{\downarrow}\mathsf{p}$ 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:

$$\begin{array}{lrcl} \text{Global Actions} & \pi & ::= & \mathsf{p} \to \mathsf{q} : \ell(A) \mid \mathsf{p} \twoheadrightarrow \mathsf{q} : \ell(A) \mid \mathsf{p}\#\mathsf{q} \\ \text{Global Types} & G & ::= & \Sigma_{i\in I}(\pi_i \,.\, G_i) \mid \mu X.G \mid X \mid \mathsf{end} \end{array}$$

Global actions π describe interactions between participants: $\mathsf{p} \to \mathsf{q} : \ell(A)$ states that role p sends a message with label ℓ and payload type A to q. Similarly, $\mathsf{p} \twoheadrightarrow \mathsf{q} : \ell(A)$ states that p connects to q by sending a message with label ℓ and payload type A. The disconnection action $\mathsf{p}\#\mathsf{q}$ 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) .
        Courier → Store : ref(Int) . Courier#Store . Store → Customer : ref(Int) .
        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* $\{\mathbf{p}_i : S_i\}_i$ *mapping role names to session types.*

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

$$\left\{\begin{array}{l}
\texttt{Customer} : \texttt{Store}!!login(\text{String}) . \mu\text{Browse} . \\
\quad \texttt{Store}!item(\text{String}) . \texttt{Store}?price(\text{Int}) . \text{Browse} \\
\quad + \ \texttt{Store}!address(\text{String}) . \texttt{Store}?ref(\text{Int}) . \#{\uparrow}\texttt{Store} . \text{end} \\
\quad + \ \texttt{Store}!quit(\mathbf{1}) . \#{\uparrow}\texttt{Store} . \text{end}, \\
\texttt{Store} : \texttt{Customer}??login(\text{String}) . \mu\text{Browse} . \\
\quad \texttt{Customer}?item(\text{String}) . \texttt{Customer}!price(\text{Int}) . \text{Browse} \\
\quad + \ \texttt{Customer}?address(\text{String}) . \texttt{Courier}!!deliver(\text{String}) . \texttt{Courier}?ref(\text{Int}) . \\
\quad \#{\uparrow}\texttt{Courier} . \texttt{Customer}!ref(\text{Int}) . \#{\downarrow}\texttt{Customer} \\
\quad + \ \texttt{Customer}?quit(\mathbf{1}) . \#{\downarrow}\texttt{Customer}, \\
\texttt{Courier} : \texttt{Store}??deliver(\text{String}) . \texttt{Store}!ref(\text{Int}) . \#{\downarrow}\texttt{Store}
\end{array}\right\}$$

We can now consider an implementation of a **Store** actor, which uses discovery to find a courier. We write **receive** $\ell(x)$ **from** $\mathbf{p}; M$ and **accept** $\ell(x)$ **from** $\mathbf{p}; M$ as syntactic sugar for **receive from** $\mathbf{p}$ $\{\ell(x) \mapsto M\}$ and **accept from** $\mathbf{p}$ $\{\ell(x) \mapsto M\}$ 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 $\texttt{Store}??deliver(\text{String}) . \texttt{Store}!ref(\text{Int}) . \#{\downarrow}\texttt{Store}$.

```
actor Store follows ty(Store) {
  accept login(credentials) 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* $(\overrightarrow{D}, \overrightarrow{P}, M)$ *of a set of definitions, protocols, and an initial term to be evaluated.*

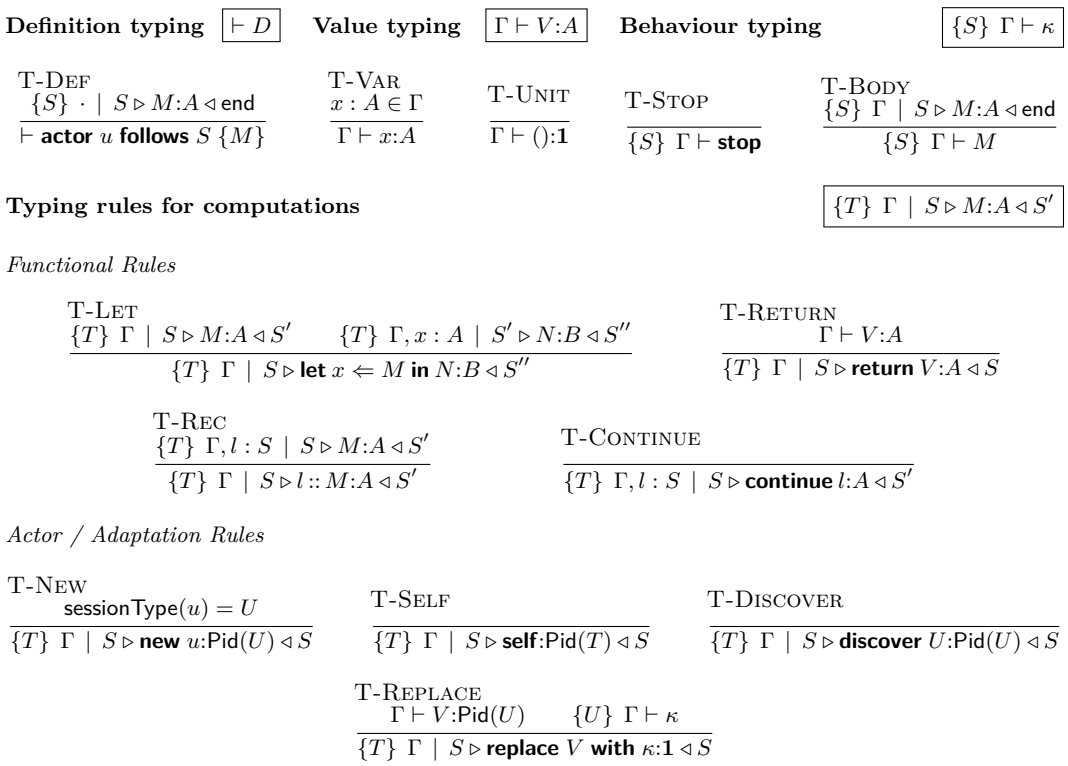

Figure 11 Typing rules (1)

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

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 Γ, value V has type A. Judgement $\vdash D$ states that an actor definition **actor** u **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 κ is well-typed under Γ. Specifically, **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 $\{T\}\ \Gamma\ \mid\ S \triangleright M{:}A \triangleleft S'$ reads "in an actor following T, under typing environment Γ 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-Let is a sequencing operation: given a construct **let** $x \Leftarrow M$ **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''.

Exception handling rules

$$\text{T-Raise}\ \frac{}{\{T\}\ \Gamma\ \mid\ S \triangleright \mathbf{raise}{:}A \triangleleft S'} \qquad \text{T-Try}\ \frac{\{T\}\ \Gamma\ \mid\ S \triangleright L{:}A \triangleleft S' \qquad \{T\}\ \Gamma\ \mid\ S \triangleright M{:}A \triangleleft S'}{\{T\}\ \Gamma\ \mid\ S \triangleright \mathbf{try}\ L\ \mathbf{catch}\ M{:}A \triangleleft S'}$$

Session communication rules

$$\text{T-Conn}\ \frac{\mathsf{p}_j!!\ell_j(A_j) \in \{\alpha_i\}_{i\in I} \qquad \Gamma \vdash V{:}A_j \qquad \Gamma \vdash W{:}\mathsf{Pid}(T) \qquad T = \mathsf{ty}(\mathsf{p}_j)}{\{T\}\ \Gamma\ \mid\ \Sigma_{i\in I}(\alpha_i \,.\, S_i) \triangleright \mathbf{connect}\ \ell_j(V)\ \mathbf{to}\ W\ \mathbf{as}\ \mathsf{p}_j{:}\mathbf{1} \triangleleft S'_j}$$

$$\text{T-Send}\ \frac{\mathsf{p}_j!\ell_j(A_j) \in \{\alpha_i\}_{i\in I} \qquad \Gamma \vdash V{:}A_j}{\{T\}\ \Gamma\ \mid\ \Sigma_{i\in I}(\alpha_i \,.\, S_i) \triangleright \mathbf{send}\ \ell_j(V)\ \mathbf{to}\ \mathsf{p}_j{:}\mathbf{1} \triangleleft S'_j}$$

$$\text{T-Accept}\ \frac{(\{T\}\ \Gamma, x : B_i\ \mid\ S_i \triangleright M_i{:}A \triangleleft S)_{i\in I}}{\{T\}\ \Gamma\ \mid\ \Sigma_{i\in I}(\mathsf{q}??\ell_i(B_i) \,.\, S_i) \triangleright \mathbf{accept\ from}\ \mathsf{q}\ \{\ell_i(x_i) \mapsto M_i\}_{i\in I}{:}A \triangleleft S}$$

$$\text{T-Recv}\ \frac{(\{T\}\ \Gamma, x : B_i\ \mid\ S_i \triangleright M_i{:}A \triangleleft S)_{i\in I}}{\{T\}\ \Gamma\ \mid\ \Sigma_{i\in I}(\mathsf{q}?\ell_i(B_i) \,.\, S_i) \triangleright \mathbf{receive\ from}\ \mathsf{q}\ \{\ell_i(x_i) \mapsto M_i\}_{i\in I}{:}A \triangleleft S}$$

$$\text{T-Wait}\ \frac{}{\{T\}\ \Gamma\ \mid\ \#{\uparrow}\mathsf{q} \,.\, S \triangleright \mathbf{wait}\ \mathsf{q}{:}\mathbf{1} \triangleleft S} \qquad \text{T-Disconn}\ \frac{}{\{T\}\ \Gamma\ \mid\ \#{\downarrow}\mathsf{q} \triangleright \mathbf{disconnect\ from}\ \mathsf{q}{:}\mathbf{1} \triangleleft \mathsf{end}}$$

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 **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-Rec extends the typing environment with a recursion label defined at the current session type. Rule T-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-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-Self retrieves a PID for the current actor, parameterised by the statically-defined session type of the local actor. Rule T-Discover states **discover** U returns a PID of type $\mathsf{Pid}(U)$. Finally, given a behaviour κ typable under a static session type U, and a process ID with the matching static type $\mathsf{Pid}(U)$, T-Replace allows replacement, and returns the unit type.

Exception handling rules. Figure 12 shows the rules for exception handling and session communication. T-Raise denotes raising an exception; since it does not return, it can have an arbitrary return type and postcondition. Rule T-Try types an exception handler **try** L **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 **try** and **catch** clauses have the same pre- and post-conditions to allow the action to be retried if necessary.

Session communication rules. Rule T-Conn types a term **connect** $\ell_j(V)$ **to** W **as** p_j. Given the precondition is a choice type containing a branch $\mathsf{p}!!\ell_j(A_j) \,.\, S'_j$, and the remote actor reference is W of type $\mathsf{Pid}(S)$, 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

Runtime syntax

Names	n	::=	$a \mid s$
Configurations	$\mathcal{C}, \mathcal{D}, \mathcal{E}$	::=	$(\nu n)\mathcal{C} \mid \mathcal{C} \parallel \mathcal{D} \mid \langle a, M, \sigma, \kappa\rangle \mid \lightning\, s[\mathsf{p}] \mid \mathbf{0}$
Connection state	σ	::=	$\bot \mid s[\mathsf{p}]\langle\tilde{\mathsf{q}}\rangle$
Runtime environments	Δ	::=	$\cdot \mid \Delta, a : S \mid \Delta, s[\mathsf{p}]\langle\tilde{\mathsf{q}}\rangle{:}S$
Evaluation contexts	E	::=	$F \mid \mathbf{let}\ x \Leftarrow E\ \mathbf{in}\ M$
Top-level contexts	F	::=	$[\,] \mid \mathbf{try}\ [\,]\ \mathbf{catch}\ M$
Pure contexts	E_{P}	::=	$[\,] \mid \mathbf{let}\ x \Leftarrow E_{\mathsf{P}}\ \mathbf{in}\ M$

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

E-Let	$\mathbf{let}\ x \Leftarrow \mathbf{return}\ V\ \mathbf{in}\ M$	$\longrightarrow_{\mathsf{M}}$	$M\{V/x\}$
E-TryReturn	$\mathbf{try}\ \mathbf{return}\ V\ \mathbf{catch}\ M$	$\longrightarrow_{\mathsf{M}}$	$\mathbf{return}\ V$
E-TryRaise	$\mathbf{try}\ \mathbf{raise}\ \mathbf{catch}\ M$	$\longrightarrow_{\mathsf{M}}$	M
E-Rec	$l :: M$	$\longrightarrow_{\mathsf{M}}$	$M\{l :: M/\mathbf{continue}\ l\}$
E-LiftM	$E[M]$	$\longrightarrow_{\mathsf{M}}$	$E[N]$ if $M \longrightarrow_{\mathsf{M}} N$

Configuration reduction (1) $\boxed{\mathcal{C} \longrightarrow \mathcal{D}}$

Actor / adaptation rules

E-Loop
$$\frac{}{\langle a, \mathbf{return}\ V, \bot, M\rangle \longrightarrow \langle a, M, \bot, M\rangle}$$

E-New
$$\frac{b \text{ is fresh} \qquad \mathsf{behaviour}(u) = M}{\langle a, E[\mathbf{new}\ u], \sigma, \kappa\rangle \longrightarrow (\nu b)(\langle a, E[\mathbf{return}\ b], \sigma, \kappa\rangle \parallel \langle b, M, \bot, M\rangle)}$$

E-Replace
$$\frac{}{\langle a, E[\mathbf{replace}\ b\ \mathbf{with}\ \kappa'], \sigma_1, \kappa_1\rangle \parallel \langle b, N, \sigma_2, \kappa_2\rangle \longrightarrow \langle a, E[\mathbf{return}\ ()], \sigma_1, \kappa_1\rangle \parallel \langle b, N, \sigma_2, \kappa'\rangle}$$

E-ReplaceSelf
$$\frac{}{\langle a, E[\mathbf{replace}\ a\ \mathbf{with}\ \kappa'], \sigma, \kappa\rangle \longrightarrow \langle a, E[\mathbf{return}\ ()], \sigma, \kappa'\rangle}$$

E-Discover
$$\frac{\mathsf{sessionType}(b) = S \qquad \neg((N = \mathbf{return}\ V \vee N = \mathbf{raise}) \wedge \kappa_2 = \mathbf{stop})}{\langle a, E[\mathbf{discover}\ S], \sigma_1, \kappa_1\rangle \parallel \langle b, E'[N], \sigma_2, \kappa_2\rangle \longrightarrow \langle a, E[\mathbf{return}\ b], \sigma_1, \kappa_1\rangle \parallel \langle b, E'[N], \sigma_2, \kappa_2\rangle}$$

E-Self
$$\frac{}{\langle a, E[\mathbf{self}], \sigma, \kappa\rangle \longrightarrow \langle a, E[\mathbf{return}\ a], \sigma, \kappa\rangle}$$

Figure 13 Operational semantics (1)

type is then advanced to S'_j. Rule T-Send follows the same pattern.

Given a session type $\Sigma_{i\in I}(\mathsf{p}??x_i(A_i))\,.\,S_i$, rule T-Accept types term $\mathbf{accept\ from}\ \mathsf{p}\ \{\ell_i(x_i) \mapsto M_i\}_{i\in I}$, enabling an actor to accept connections with messages ℓ_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 same result type and postcondition. Rule T-Recv is similar.

Rule T-Wait handles waiting for a participant p to disconnect from a session, requiring a pre-condition of $\#{\uparrow}\mathsf{p}\,.\,S$, returning the unit type and advancing the session type to S. Rule T-Disconnect is similar and advances the session type to end.

5.3 Operational semantics

We describe the semantics of EnsembleS 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 EnsembleS.

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 $\mathcal{C}, \mathcal{D}, \mathcal{E}$, represent the concurrent fragment of the language. Like in the π-calculus [43], name restrictions $(\nu n)\mathcal{C}$ bind name n in $\mathcal{C}$, $\mathcal{C} \parallel \mathcal{D}$ denotes $\mathcal{C}$ and $\mathcal{D}$ running in parallel, and the $\mathbf{0}$ 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; σ is the connection state; and κ 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 $\tilde{\mathsf{q}}$, written $s[\mathsf{p}]\langle\tilde{\mathsf{q}}\rangle$.

Following the work on Exceptional GV by Fowler et al. [22] and the work on affine sessions by Mostrous and Vasconcelos [46], a *zapper thread* $\lightning\, s[\mathsf{p}]$ 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 Γ is an unrestricted typing environment used for typing values and configurations, we introduce Δ 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 $s[\mathsf{p}]\langle\tilde{\mathsf{q}}\rangle{:}S$, stating that in session s, role p is connected to roles $\tilde{\mathsf{q}}$ 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 E_{P} 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 $(\nu a)(\langle a, M, \bot, \textbf{\textit{stop}}\rangle)$.*

5.3.2 Reduction rules

Term reduction $\longrightarrow_{\mathsf{M}}$ is standard β-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 $\langle b, M, \bot, M\rangle$ where M is the behaviour specified by u, returning the process ID b. Rules E-Replace 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** $\ell_j(V)$ **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) $\boxed{\mathcal{C} \longrightarrow \mathcal{D}}$

Session reduction rules

E-ConnInit

$$\frac{j \in I}{\langle a, E[F[\textbf{connect}\ \ell_j(V)\ \textbf{to}\ b\ \textbf{as}\ \mathsf{q}]], \bot, \kappa_1\rangle \parallel \langle b, E'[F'[\textbf{accept from}\ \mathsf{p}\ \{\ell_i(x_i) \mapsto M_i\}_{i\in I}]], \bot, \kappa_2\rangle \longrightarrow (\nu s)(\langle a, E[\textbf{return}\ ()], s[\mathsf{p}]\langle \mathsf{q}\rangle, \kappa_1\rangle \parallel \langle b, E'[M_j\{V/x_j\}], s[\mathsf{q}]\langle \mathsf{p}\rangle, \kappa_2\rangle)}$$

E-Conn

$$\frac{\mathsf{q} \notin \tilde{\mathsf{r}}}{\langle a, E[F[\textbf{connect}\ \ell_j(V)\ \textbf{to}\ b\ \textbf{as}\ \mathsf{q}]], s[\mathsf{p}]\langle \tilde{\mathsf{r}}\rangle, \kappa_1\rangle \parallel \langle b, E'[F'[\textbf{accept from}\ \mathsf{p}\ \{\ell_i(x_i) \mapsto N_i\}_{i\in I}]], \bot, \kappa_2\rangle \longrightarrow \langle a, E[\textbf{return}\ ()], s[\mathsf{p}]\langle \tilde{\mathsf{r}}, \mathsf{q}\rangle, \kappa_1\rangle \parallel \langle b, E'[N_j\{V/x_j\}], s[\mathsf{q}]\langle \mathsf{p}\rangle, \kappa_2\rangle}$$

E-ConnFail

$$\frac{((N = \textbf{return}\ V \vee N = E_{\mathsf{P}}[\textbf{raise}]) \wedge \kappa_2 = \textbf{stop}) \vee \sigma \neq \bot}{\langle a, E[\textbf{connect}\ \ell_j(V)\ \textbf{to}\ b\ \textbf{as}\ \mathsf{q}], s[\mathsf{p}]\langle \tilde{\mathsf{r}}\rangle, \kappa_1\rangle \parallel \langle b, N, \sigma, \kappa_2\rangle \longrightarrow \langle a, E[\textbf{raise}], s[\mathsf{p}]\langle \tilde{\mathsf{r}}\rangle, \kappa_1\rangle \parallel \langle b, N, \sigma, \kappa_2\rangle}$$

E-Disconn

$$\frac{}{\langle a, E[F[\textbf{wait}\ \mathsf{q}]], s[\mathsf{p}]\langle \tilde{\mathsf{r}}, \mathsf{q}\rangle, \kappa_1\rangle \parallel \langle b, E'[F'[\textbf{disconnect from}\ \mathsf{p}]], s[\mathsf{q}]\langle \mathsf{p}\rangle, \kappa_2\rangle \longrightarrow \langle a, E[\textbf{return}\ ()], s[\mathsf{p}]\langle \tilde{\mathsf{r}}\rangle, \kappa_1\rangle \parallel \langle b, E'[\textbf{return}\ ()], \bot, \kappa_2\rangle}$$

E-Comm

$$\frac{j \in I \qquad \mathsf{q} \in \tilde{\mathsf{r}} \qquad \mathsf{p} \in \tilde{\mathsf{s}}}{\langle a, E[F[\textbf{send}\ \ell_j(V)\ \textbf{to}\ \mathsf{q}]], s[\mathsf{p}]\langle \tilde{\mathsf{r}}\rangle, \kappa_1\rangle \parallel \langle b, E'[F'[\textbf{receive from}\ \mathsf{p}\ \{\ell_i(x_i) \mapsto M_i\}_{i\in I}]], s[\mathsf{q}]\langle \tilde{\mathsf{s}}\rangle, \kappa_2\rangle \longrightarrow \langle a, E[\textbf{return}\ ()], s[\mathsf{p}]\langle \tilde{\mathsf{r}}\rangle, \kappa_1\rangle \parallel \langle b, E'[M_j\{V/x_j\}], s[\mathsf{q}]\langle \tilde{\mathsf{s}}\rangle, \kappa_2\rangle}$$

E-Complete

$$\frac{}{(\nu s)(\langle a, \textbf{return}\ V, s[\mathsf{p}]\langle \emptyset\rangle, \kappa\rangle) \longrightarrow \langle a, \textbf{return}\ V, \bot, \kappa\rangle}$$

Figure 14 Operational semantics (2)

$M_i\}_{i\in I}$. Rule E-ConnInit returns the unit value to actor a; creates a fresh session name restriction s, sets the connection state of a to $s[\mathsf{p}]\langle \mathsf{q}\rangle$ and of b to $s[\mathsf{q}]\langle \mathsf{p}\rangle$; accepting actor b then evaluates continuation M_j with V substituted for x_j. Since exception handlers only scope over a single communication action, the top-level frames F, F' in each actor are discarded if the communication succeeds. Rule E-Conn handles the case where the connecting actor is already part of a session and behaves similarly to E-ConnInit, without creating a new session name restriction. A connection can fail if an actor attempts to connect to another actor which is terminated or is already involved in a session; in these cases, E-ConnFail raises an exception in the connecting actor.

Rule E-Disconn handles the case where an actor b leaves a session, synchronising with an actor a. In this case, the unit value is returned to both callers, and the connection state of b is set to $\bot$. Rule E-Comm handles session communication when two participants are already connected to the same session, and is similar to E-Conn. Rule E-Complete garbage collects a session after it has completed and sets the initiator's connection state to $\bot$.

Exception handling rules. Exception handling rules allow safe session communication in the presence of exceptions. Rule E-CommRaise 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 $\mathsf{subj}(M) = \mathsf{p}$ if $M \in \{\textbf{send}\ \ell(V)\ \textbf{to}\ \mathsf{p}, \textbf{receive from}\ \mathsf{p}\ \{\ell_i(x_i) \mapsto N_i\}_i, \textbf{wait}\ \mathsf{p}, \textbf{disconnect from}\ \mathsf{p}\}$. 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.

Configuration reduction (3) $\boxed{\mathcal{C} \longrightarrow \mathcal{D}}$

Exception handling rules

$$\text{E-CommRaise}\ \frac{\mathsf{subj}(M) = \mathsf{q}}{\langle a, E[M], s[\mathsf{p}]\langle \tilde{\mathsf{r}} \rangle, \kappa \rangle \parallel \lightning\, s[\mathsf{q}] \longrightarrow \langle a, E[\mathbf{raise}], s[\mathsf{p}]\langle \tilde{\mathsf{r}} \rangle, \kappa \rangle \parallel \lightning\, s[\mathsf{q}]} \qquad \text{E-FailS}\ \frac{}{\langle a, E_{\mathsf{P}}[\mathbf{raise}], s[\mathsf{p}]\langle \tilde{\mathsf{r}} \rangle, \kappa \rangle \longrightarrow \langle a, \mathbf{raise}, \bot, \kappa \rangle \parallel \lightning\, s[\mathsf{p}]} \qquad \text{E-FailLoop}\ \frac{}{\langle a, E_{\mathsf{P}}[\mathbf{raise}], \bot, M \rangle \longrightarrow \langle a, M, \bot, M \rangle}$$

Administrative rules

$$\text{E-LiftM}\ \frac{M \longrightarrow_{\mathsf{M}} M'}{\langle a, E[M], \sigma, \kappa \rangle \longrightarrow \langle a, E[M'], \sigma, \kappa \rangle} \qquad \text{E-Equiv}\ \frac{\mathcal{C} \equiv \mathcal{C}' \quad \mathcal{C}' \longrightarrow \mathcal{D}' \quad \mathcal{D}' \equiv \mathcal{D}}{\mathcal{C} \longrightarrow \mathcal{D}} \qquad \text{E-Par}\ \frac{\mathcal{C} \longrightarrow \mathcal{C}'}{\mathcal{C} \parallel \mathcal{D} \longrightarrow \mathcal{C}' \parallel \mathcal{D}} \qquad \text{E-Nu}\ \frac{\mathcal{C} \longrightarrow \mathcal{D}}{(\nu n)\mathcal{C} \longrightarrow (\nu n)\mathcal{D}}$$

Configuration equivalence $\boxed{\mathcal{C} \equiv \mathcal{D}}$

$$\mathcal{C} \parallel \mathcal{D} \equiv \mathcal{D} \parallel \mathcal{C} \qquad \mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E}) \equiv (\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E} \qquad (\nu n_1)(\nu n_2)\mathcal{C} \equiv (\nu n_2)(\nu n_1)\mathcal{C}$$

$$\mathcal{C} \parallel (\nu n)\mathcal{D} \equiv (\nu n)(\mathcal{C} \parallel \mathcal{D}) \quad \text{if } n \notin \mathsf{fn}(\mathcal{C}) \qquad (\nu s)(\lightning\, s[\mathsf{p}_1] \parallel \cdots \parallel \lightning\, s[\mathsf{p}_n]) \parallel \mathcal{C} \equiv \mathcal{C} \qquad \mathcal{C} \parallel \mathbf{0} \equiv \mathcal{C}$$

Figure 15 Operational semantics (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 π-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 $\Gamma; \Delta \vdash \mathcal{C}$ states that configuration $\mathcal{C}$ is well-typed under term typing environment Γ and runtime typing environment Δ.

Rule T-Pid types actor name restriction $(\nu a)\mathcal{C}$ 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 $(\nu s)\mathcal{C}$ are typed by T-Session. We follow the formulation of Scalas and Yoshida [54] which types multiparty sessions using a parametric safety property φ; we discuss safety properties in more depth in Section 5.4.2. Let Δ' be a runtime typing environment containing only mappings of the form $s[\mathsf{p}_i]\langle \tilde{\mathsf{q}}_i \rangle : S_i$. Assuming Δ does not contain any mappings involving session s and Δ' satisfies φ, the rule states that $\mathcal{C}$ is typable under typing environment Γ and runtime typing environment Δ, Δ'. It is sometimes convenient to annotate session ν-binders with their environment, e.g., $(\nu s : \Delta')\mathcal{C}$.

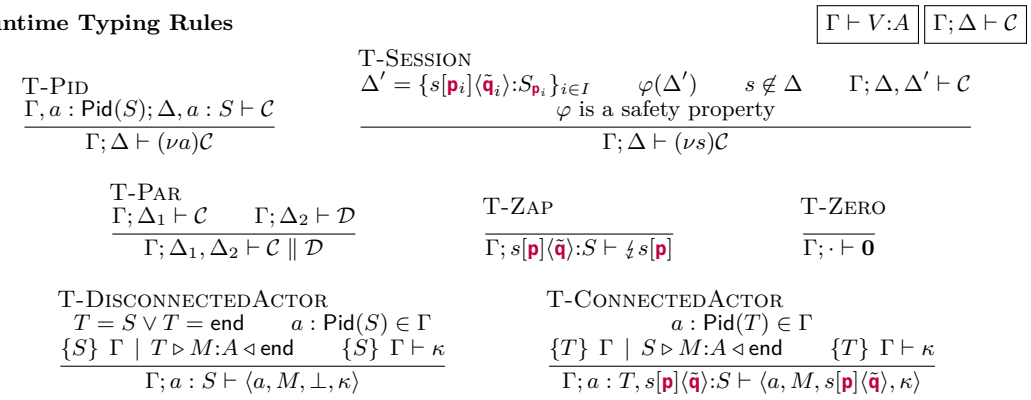

Figure 16 Runtime typing rules

Rule T-Par types each subconfiguration of a parallel composition by splitting the linear runtime environment. Rule T-Zap types a zapper thread $\lightning\, s[\mathbf{p}]$, assuming the runtime environment contains an entry $s[\mathbf{p}]\langle\tilde{\mathbf{q}}\rangle{:}S$ for any session type S.

Finally, rules T-DisconnectedActor and T-ConnectedActor type disconnected and connected actor configurations respectively. Given an actor with name a and static session type T, both rules require that the typing environment contains $a : \mathsf{Pid}(T)$ and runtime environment contains $a : T$. Both rules require that the current session type is fully consumed by the currently-evaluating term and that the actor's behaviour should be typable under T. Rule T-DisconnectedActor requires that the currently-evaluating term must be typable under either T or end, whereas to type a connection state of $s[\mathbf{p}]\langle\tilde{\mathbf{q}}\rangle$ and current session type S, T-ConnectedActor requires an entry $s[\mathbf{p}]\langle\tilde{\mathbf{q}}\rangle{:}S$ in the runtime environment.

5.4.2 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 Scalas 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: $\Delta \xrightarrow{\gamma} \Delta'$, which handles reduction of individual local types, and a *synchronisation* judgement $\Delta \overset{\rho}{\Rightarrow} \Delta'$.

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

Given a role $\mathbf{p}$ connected to $\mathbf{q}$ with a session choice containing a send or receive action $\mathbf{q}\dagger\ell_j(A)\,.\,S'_j$, rule ET-Act will emit a label $s{:}\mathbf{p}\dagger\mathbf{q}{::}\ell_j(A_j)$ and advance the session type of $\mathbf{p}$.

Rule ET-Wait handles the reduction of $\#{\uparrow}\mathbf{q}\,.\,S$ actions, $s[\mathbf{p}]\langle\tilde{\mathbf{r}},\mathbf{q}\rangle{:}\#{\uparrow}\mathbf{q}\,.\,S$, where $\mathbf{p}$ waits for $\mathbf{q}$ to disconnect: the reduction emits label $\mathbf{p}{:}\mathbf{q}\#{\uparrow}$ and removes $\mathbf{q}$ from $\mathbf{p}$'s connected roles. Similarly, rule ET-Disconn handles disconnection, by emitting label $\mathbf{p}{:}\mathbf{q}\#{\downarrow}$ and removing

Labels

Labels	γ	::=	$s{:}\mathbf{p}\dagger\mathbf{q}{::}\ell(A) \mid s{:}\mathbf{p} \twoheadrightarrow \mathbf{q}{::}\ell \mid s{:}\mathbf{p}\ddagger\mathbf{q}$
Synchronisation labels	ρ	::=	$s{:}\mathbf{p},\mathbf{q}{::}\ell \mid s{:}\mathbf{p} \twoheadrightarrow \mathbf{q}{::}\ell \mid s{:}\mathbf{p}\#\mathbf{q}$

Reduction on runtime typing environments

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

ET-Conn
$$\frac{\exists j \in I.\alpha_j = \mathbf{q}!!\ell_j(A_j) \quad \mathsf{ty}(\mathbf{q}) = \Sigma_{k\in K}(\mathbf{p}??\ell_k(B_k)\,.\,T_k) \quad j \in K \quad A_j = B_j}{s[\mathbf{p}]\langle\tilde{\mathbf{r}}\rangle{:}\Sigma_{i\in I}(\alpha_i\,.\,S_i) \xrightarrow{s:\mathbf{p}\twoheadrightarrow\mathbf{q}::\ell_j} s[\mathbf{p}]\langle\tilde{\mathbf{r}},\mathbf{q}\rangle{:}S_j, s[\mathbf{q}]\langle\mathbf{p}\rangle{:}T_j}$$

ET-Act
$$\frac{\exists j \in I.\alpha_j = \mathbf{q}\dagger\ell_j(A_j) \quad \mathbf{q} \in \tilde{\mathbf{r}}}{s[\mathbf{p}]\langle\tilde{\mathbf{r}}\rangle{:}\Sigma_{i\in I}(\alpha_i\,.\,S_i) \xrightarrow{s:\mathbf{p}\dagger\mathbf{q}::\ell_j(A_j)} s[\mathbf{p}]\langle\tilde{\mathbf{r}}\rangle{:}S_j}$$

ET-Wait
$$\frac{}{s[\mathbf{p}]\langle\tilde{\mathbf{r}},\mathbf{q}\rangle{:}\#\uparrow\mathbf{q}\,.\,S \xrightarrow{s:\mathbf{p}\#\uparrow\mathbf{q}} s[\mathbf{p}]\langle\tilde{\mathbf{r}}\rangle{:}S}$$

ET-Disconn
$$\frac{}{s[\mathbf{p}]\langle\mathbf{q}\rangle{:}\#\downarrow\mathbf{q} \xrightarrow{s:\mathbf{p}\#\downarrow\mathbf{q}} .}$$

ET-Rec
$$\frac{\Delta, s[\mathbf{p}]\langle\tilde{\mathbf{q}}\rangle{:}S\{\mu X.S/X\} \xrightarrow{\gamma} \Delta'}{\Delta, s[\mathbf{p}]\langle\tilde{\mathbf{q}}\rangle{:}\mu X.S \xrightarrow{\gamma} \Delta'}$$

ET-Cong1
$$\frac{\Delta \xrightarrow{\gamma} \Delta'}{\Delta, s[\mathbf{p}]\langle\tilde{\mathbf{q}}\rangle{:}S \xrightarrow{\gamma} \Delta', s[\mathbf{p}]\langle\tilde{\mathbf{q}}\rangle{:}S}$$

ET-Cong2
$$\frac{\Delta \xrightarrow{\gamma} \Delta'}{\Delta, a{:}S \xrightarrow{\gamma} \Delta', a{:}S}$$

Synchronisation $\boxed{\Delta \xRightarrow{\rho} \Delta'}$

ET-ConnSync
$$\frac{\Delta \xrightarrow{s:\mathbf{p}\twoheadrightarrow\mathbf{q}::\ell} \Delta'}{\Delta \xRightarrow{s:\mathbf{p}\twoheadrightarrow\mathbf{q}::\ell} \Delta'}$$

ET-Comm
$$\frac{\Delta_1 \xrightarrow{s:\mathbf{p}!\mathbf{q}::\ell(A)} \Delta_1' \qquad \Delta_2 \xrightarrow{s:\mathbf{q}?\mathbf{p}::\ell(A)} \Delta_2'}{\Delta_1, \Delta_2 \xRightarrow{s:\mathbf{p},\mathbf{q}::\ell} \Delta_1', \Delta_2'}$$

ET-Disconn
$$\frac{\Delta_1 \xrightarrow{s:\mathbf{p}\#\uparrow\mathbf{q}} \Delta_1' \qquad \Delta_2 \xrightarrow{s:\mathbf{q}\#\downarrow\mathbf{p}} \Delta_2'}{\Delta_1, \Delta_2 \xRightarrow{s:\mathbf{p}\#\mathbf{q}} \Delta_1', \Delta_2'}$$

Figure 17 Labelled transition system for runtime typing environments

the entry from the environment. ET-Rec handles recursive types, and the ET-Cong rules handle reduction of sub-environments.

Rule ET-ConnSync states that connection is a synchronisation action, and rules ET-Comm and ET-Disconn handle synchronisation between dual actions in sub-environments, emitting synchronisation labels $s{:}\mathbf{p},\mathbf{q}{::}\ell$ and $s{:}\mathbf{p}\#\mathbf{q}$ respectively. We omit the congruence rules for synchronisation actions. We say that a runtime environment *reduces*, written $\Delta \Longrightarrow$, if there exists some Δ' such that $\Delta \Longrightarrow \Delta'$.

Safety. A *safety property* describes a set of invariants on typing environments which allow us to prove preservation. Since the type system is parametric in the given safety property, we can tweak the property to permit or rule out different typing environments satisfying particular behavioural properties; however, we need only prove type preservation once, using the weakest safety property. Our safety property is different to the safety property described by Scalas and Yoshida [54] in order to account for explicit connection actions.

▶ **Definition 5** (Safety Property). *φ is a* safety property *of runtime typing contexts Δ if:*

1. *$\varphi(\Delta, s[\mathbf{p}]\langle\tilde{\mathbf{r}}\rangle{:}\Sigma_{i\in I}(\alpha_i\,.\,S_i), s[\mathbf{q}]\langle\tilde{\mathbf{s}}\rangle{:}\Sigma_{j\in J}(\mathbf{p}?\ell_j(B_j)\,.\,T_j))$ implies that if $\mathbf{q}!\ell_k(A_k) \in \{\alpha_i\}_{i\in I}$, then $k \in J$, $\mathbf{q} \in \tilde{\mathbf{r}}$, $\mathbf{p} \in \tilde{\mathbf{s}}$, and $A_k = B_k$.*
2. *$\varphi(\Delta, s[\mathbf{p}]\langle\tilde{\mathbf{r}}\rangle{:}\Sigma_{i\in I}(\alpha_i\,.\,S_i))$ implies that if $\alpha_i = \mathbf{q}!!\ell_j(A_j) \in \{\alpha_i\}_{i\in I}$, then $\mathbf{q} \notin \tilde{\mathbf{r}}$, $s[\mathbf{q}]\langle\tilde{\mathbf{r}}\rangle \notin \mathit{dom}(\Delta)$, and $\mathsf{ty}(\mathbf{q}) = \Sigma_{k\in K}(\mathbf{p}??\ell_k(B_k)\,.\,T_k)$ with $j \in K$ and $A_j = B_j$.*
3. *$\varphi(\Delta, s[\mathbf{p}]\langle\tilde{\mathbf{q}}\rangle{:}\mu X.S)$ implies $\varphi(\Delta, s[\mathbf{p}]\langle\tilde{\mathbf{q}}\rangle{:}S\{\mu X.S/X\})$*
4. *$\varphi(\Delta)$ and $\Delta \Longrightarrow \Delta'$ implies $\varphi(\Delta')$*

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

Clause (1) ensures that communication actions between participants are compatible: if $\mathbf{p}$ is sending a message with label ℓ and payload type A to $\mathbf{q}$, and $\mathbf{q}$ is receiving from $\mathbf{p}$, then the two roles must be connected, and $\mathbf{q}$ must be able to receive ℓ with a matching payload.

Clause (2) states that if $\mathbf{p}$ is connecting to a role $\mathbf{q}$ with label ℓ, then $\mathbf{q}$ should not already be involved in the session, and should be able to accept from $\mathbf{p}$ on message label ℓ with a compatible payload type. The requirement that $\mathbf{q}$ 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 Δ is safe, we define $\varphi(\Delta) = \{\Delta' \mid \Delta \Longrightarrow^* \Delta'\}$ and verify that φ 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* $\mathsf{inactive}(S)$, *if $S = \mathsf{end}$ or $S = \Sigma_{i\in I}(\mathbf{p}??\ell_i(A_i).S_i)$. Otherwise, S is* active, *written* $\mathsf{active}(S)$.

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

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

▶ **Definition 8** (Well-formed protocol). *A protocol $P = \{\mathbf{p}_i : S_i\}_{i\in I}$ is* well-formed *if it has a unique initiator $\mathbf{q}$ of type S and $\mathsf{safe}(s[\mathbf{q}]\langle\emptyset\rangle{:}S)$ 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 $\mathsf{safe}(s[\texttt{Customer}]\langle\emptyset\rangle{:}\,\mathsf{ty}(\texttt{Customer}))$.

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

1. *For each actor definition $D =$ **actor** u **follows** S $\{N\} \in \overrightarrow{D}$, there exists some role $\mathbf{p} \in \overrightarrow{P}$ such that $\mathsf{ty}(\mathbf{p}) = S$, and $\{S\} \cdot \mid S \triangleright N{:}A \triangleleft \mathsf{end}$*
2. *Each protocol $P \in \overrightarrow{P}$ 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: $\{\mathsf{end}\} \cdot \mid \mathsf{end} \triangleright M{:}A \triangleleft \mathsf{end}$*

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

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* Δ *is* final*, written* $\mathsf{end}(\Delta)$*,* $\Delta = \{s[\mathtt{p}]\langle\emptyset\rangle{:}\mathsf{end}\}$ *for some* s *and* $\mathtt{p}$*.*

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 $\mathsf{roles}(\rho)$ denote the roles referenced in a synchronisation label (i.e., $\mathsf{roles}(\rho) = \{\mathtt{p}, \mathtt{q}\}$ for $\rho \in \{s{:}\mathtt{p} \twoheadrightarrow \mathtt{q}{::}\ell, s{:}\mathtt{p}, \mathtt{q}{::}\ell, s{:}\mathtt{p}\#\mathtt{q}\}$).

▶ **Definition 12** (Progress (Runtime typing environments)). *A runtime typing environment* Δ satisfies progress*, written* $\mathsf{prog}(\Delta)$*, if:*

- *(Role progress) for each* $s[\mathtt{p}_i]\langle\tilde{\mathtt{q}}_i\rangle{:}S_i \in \Delta$ *s.t.* $\mathsf{active}(S_i)$*,* $\Delta \Longrightarrow^* \Delta' \overset{\rho}{\Longrightarrow}$ *with* $\mathtt{p} \in \mathsf{roles}(\rho)$
- *(Eventual comm.) if* $\Delta \Longrightarrow^* \Delta' \xrightarrow{s:\mathtt{p}!\mathtt{q}::\ell(A)}$*, then* $\Delta' \overset{\overrightarrow{\rho}}{\Longrightarrow} \Delta'' \xrightarrow{s:\mathtt{q}?\mathtt{p}::\ell(A)}$*, with* $\mathtt{p} \notin \mathsf{roles}(\overrightarrow{\rho})$
- *(Correct termination)* $\Delta \Longrightarrow^* \Delta' \not\Longrightarrow$ *implies* $\mathsf{end}(\Delta)$

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 $s[\mathtt{Customer}]\langle\emptyset\rangle{:}\mathsf{end}$.

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 $\mathcal{G} ::= [\,] \mid (\nu s)\mathcal{G} \mid \mathcal{G} \parallel \mathcal{C}$.

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 $\mathcal{S}$ *if it can be written:*
$(\nu s)(\langle a_1, M_1, s[\mathtt{p}_1]\langle\tilde{\mathtt{q}}_1\rangle, \kappa_1\rangle \parallel \cdots \parallel \langle a_m, M_m, s[\mathtt{p}_m]\langle\tilde{\mathtt{q}}_m\rangle, \kappa_m\rangle \parallel \lightning\, s[\mathtt{p}_{m+1}] \parallel \cdots \parallel \lightning\, s[\mathtt{p}_n])$

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* $\mathbf{0}$ *or can be written:* $(\nu a_1 \cdots a_l)(\nu b_1 \cdots b_m)(\mathcal{S}_1 \parallel \cdots \parallel \mathcal{S}_n \parallel \langle b_1, M_1, \bot, \kappa_1\rangle \parallel \cdots \parallel \langle b_m, M_m, \bot, \kappa_n\rangle)$.

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

▶ **Lemma 15** (Canonical forms). *If* $\cdot;\cdot \vdash \mathcal{C}$*, 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* $M = \mathbf{return}\, V$ *or* $M = E_P[\mathbf{raise}]$*, and* $\kappa = \mathbf{stop}$*.*

- *An actor* $\langle a, M, \sigma, \kappa\rangle$ *which is a subconfiguration of* $\mathcal{C}$ *has an* unmatched discover *if no other non-terminated actor in* $\mathcal{C}$ *has session type* S.
- *An actor* $\langle a, M, \sigma, \kappa\rangle$ *is* accepting *if* $M = E[\textbf{\textit{accept from}}\ \mathtt{p}\ \{\ell_j(x_j) \mapsto N_j\}_j]$ *for some evaluation context* E *and role* $\mathtt{p}$.

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* $\mathcal{S}$ *is a* failed session*, written* $\mathit{failed}(\mathcal{S})$*, if* $\mathcal{S} \equiv (\nu s)(\lightning\, s[\mathtt{p}_1] \parallel \cdots \parallel \lightning\, s[\mathtt{p}_n])$.

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* $\cdot;\cdot \vdash \mathcal{C}$ *where* $\mathcal{C}$ *does not contain an unmatched discover,* $\mathcal{C} \equiv \mathcal{G}[\mathcal{S}]$ *and* $\mathcal{S} = (\nu s : \Delta)\mathcal{D}$ *with* $\mathit{prog}(\Delta)$*, and* $\mathcal{S}$ *is not a failed session, then* $\mathcal{C} \longrightarrow$.

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* $\cdot;\cdot \vdash \mathcal{C}$ *where* $\mathcal{C}$ *is in canonical form. If* $\mathcal{C}$ *does not contain an unmatched discover, either* $\exists \mathcal{D}$ *such that* $\mathcal{C} \longrightarrow \mathcal{D}$*, or* $\mathcal{C} \equiv \mathbf{0}$*, or* $\mathcal{C} \equiv (\nu b_1 \cdots \nu b_n)(\langle b_1, N_1, \bot, \kappa_1\rangle \parallel \cdots \parallel \langle b_n, N_n, \bot, \kappa_n\rangle)$ *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.

6 Related Work

Behavioural typing for actors. Mostrous 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* multiparty session calculus. Instead of ensuring safety using binary duality, they define an LTS on local types and *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 AIOCJ 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 Press series in artificial intelligence. MIT Press, 1990.
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, 2012. Proceedings*, volume 7705 of *Lecture Notes in Computer Science*, pages 207–222. Springer, 2012. ISBN 978-3-642-35181-5. doi: 10.1007/978-3-642-35182-2_15. URL https://doi.org/10.1007/978-3-642-35182-2_15.
3. Robert Atkey. Parameterised notions of computation. *Journal of Functional Programming*, 19(3-4):335–376, 2009.
4. Mehdi Bagherzadeh and Hridesh Rajan. Order types: static reasoning about message races in asynchronous message passing concurrency. In *AGERE!@SPLASH*, pages 21–30. ACM, 2017.
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. *Journal of Logical and Algebraic Methods in Programming*, 84(6):763–780, 2015. doi: 10.1016/j.jlamp.2015.09.003. URL https://doi.org/10.1016/j.jlamp.2015.09.003.
6. J. Baumann, F. Hohl, K. Rothermel, and M. Straßer. *MOLE — Concepts of Mobile Agent System*, page 535–554. ACM Press/Addison-Wesley Publishing Co., USA, 1999. ISBN 0201379287.
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): 4–14, 2010. doi: 10.1109/JSAC.2010.100102.
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 7609 of *Lecture Notes in Computer Science*, pages 269–283. Springer, 2012. doi: 10.1007/978-3-642-34026-0_20. URL https://doi.org/10.1007/978-3-642-34026-0_20.

9 Mario Bravetti, Cinzia Di Giusto, Jorge A. Pérez, and Gianluigi Zavattaro. Adaptable processes. *Logical Methods in Computer Science*, 8(4), 2012. doi: 10.2168/LMCS-8(4:13)2012. URL https://doi.org/10.2168/LMCS-8(4:13)2012.

10 Callum Cameron, Paul Harvey, and Joseph Sventek. A virtual machine for the Insense language. In *Proceedings of the International Conference on Mobile Wireless Middleware, Operating Systems and Applications (Mobilware)*, pages 1–10. IEEE, 2013. doi: 10.1109/Mobilware.2013.17.

11 Ilaria Castellani, Mariangiola Dezani-Ciancaglini, and Jorge A. Pérez. Self-adaptation and secure information flow in multiparty communications. *Formal Aspects of Computing*, 28(4):669–696, 2016. doi: 10.1007/s00165-016-0381-3. URL https://doi.org/10.1007/s00165-016-0381-3.

12 Francesco Cesarini and Steve Vinoski. *Designing for Scalability with Erlang/OTP: Implement Robust, Fault-Tolerant Systems*. O'Reilly Media, Inc., 1st edition, 2016. ISBN 1449320732.

13 Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. Self-adaptive multiparty sessions. *Service Oriented Computing and Applications*, 9(3-4):249–268, 2015. doi: 10.1007/s11761-014-0171-9. URL https://doi.org/10.1007/s11761-014-0171-9.

14 Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. Global progress for dynamically interleaved multiparty sessions. *Mathematical Structures in Computer Science*, 26(2):238–302, 2016.

15 Frank S. de Boer, Dave Clarke, and Einar Broch Johnsen. A complete guide to the future. In *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 316–330. Springer, 2007.

16 Ugo de'Liguoro and Luca Padovani. Mailbox types for unordered interactions. In *ECOOP*, volume 109 of *LIPIcs*, pages 15:1–15:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

17 Romain Demangeon and Kohei Honda. Nested protocols in session types. In *CONCUR*, volume 7454 of *Lecture Notes in Computer Science*, pages 272–286. Springer, 2012.

18 Cinzia Di Giusto and Jorge A. Pérez. An event-based approach to runtime adaptation in communication-centric systems. In *Proceedings of the 11th and 12th International Workshops on Web Services, Formal Methods and Behavioural Types (WS-FM 2014, WS-FM/BEAT 2015)*, Lecture Notes in Computer Science, pages 67–85. Springer, 2015. doi: 10.1007/978-3-319-33612-1_5. URL https://doi.org/10.1007/978-3-319-33612-1_5.

19 Cinzia Di Giusto and Jorge A. Pérez. Disciplined structured communications with disciplined runtime adaptation. *Science of Computer Programming*, 97:235–265, 2015. doi: 10.1016/j.scico.2014.04.017. URL https://doi.org/10.1016/j.scico.2014.04.017.

20 Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *PLDI*, pages 237–247. ACM, 1993.

21 Simon Fowler. An Erlang implementation of multiparty session actors. In *Proceedings of the 9th Interaction and Concurrency Experience (ICE)*, volume 223 of *Electronic Proceedings in Theoretical Computer Science*, pages 36–50. Open Publishing Association, 2016. doi: 10.4204/EPTCS.223.3.

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

23 T. Gu, H. K. Pung, and D. Q. Zhang. Toward an osgi-based infrastructure for context-aware applications. *IEEE Pervasive Computing*, 3(4):66–74, 2004. doi: 10.1109/MPRV.2004.19.

24 Paul Harvey. *A linguistic approach to concurrent, distributed, and adaptive programming across heterogeneous platforms*. PhD thesis, School of Computing Science, University of Glasgow, 2015. URL http://theses.gla.ac.uk/6749/.

25 Paul Harvey and Joseph Sventek. Adaptable actors: just what the world needs. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems (PLOS)*, pages 22–28. ACM, 2017. ISBN 978-1-4503-5153-9. doi: 10.1145/3144555.3144559. URL http://doi.acm.org/10.1145/3144555.3144559.

26 Richard Hayton, Michael Bursell, Douglas I. Donaldson, W. Harwood, and Andrew Herbert. Mobile java objects. *Distributed Syst. Eng.*, 6(1):51, 1999. doi: 10.1088/0967-1846/6/1/306. URL https://doi.org/10.1088/0967-1846/6/1/306.

27 Carl Hewitt, Peter Boehler Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In Nils J. Nilsson, editor, *Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Standford, CA, USA, August 20-23, 1973*, pages 235–245. William Kaufmann, 1973. URL http://ijcai.org/Proceedings/73/Papers/027B.pdf.

28 Carl Hewitt, Peter Boehler Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc. URL http://dl.acm.org/citation.cfm?id=1624775.1624804.

29 Kohei Honda. Types for dyadic interaction. In *CONCUR '93, 4th International Conference on Concurrency Theory*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993. doi: 10.1007/3-540-57208-2_35. URL https://doi.org/10.1007/3-540-57208-2_35.

30 Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998. doi: 10.1007/BFb0053567. URL https://doi.org/10.1007/BFb0053567.

31 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, volume 43, pages 273–284. ACM, 2008. doi: 10.1145/1328897.1328472. URL http://doi.acm.org/10.1145/1328897.1328472.

32 Raymond Hu and Nobuko Yoshida. Explicit connection actions in multiparty session types. In *Proceedings of the 20th International Conference on Fundamental Approaches to Software Engineering (FASE)*, Lecture Notes in Computer Science, pages 116–133. Springer, 2017. ISBN 978-3-662-54494-5. doi: 10.1007/978-3-662-54494-5_7. URL https://doi.org/10.1007/978-3-662-54494-5_7.

33 Danny Hughes, Klaas Thoelen, Wouter Horré, Nelson Matthys, Javier Del Cid, Sam Michiels, Christophe Huygens, and Wouter Joosen. LooCI: a loosely-coupled component infrastructure for networked embedded systems. In *Proceedings of the 7th International Conference on Advances in Mobile Computing and Multimedia (MoMM)*, pages 195–203. ACM, 2009. ISBN 978-1-60558-659-5. doi: http://doi.acm.org/10.1145/1821748.1821787. URL http://doi.acm.org/10.1145/1821748.1821787.

34 Jonathan W. Hui and David Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems (SenSys)*, pages 81–94. ACM, 2004.

ISBN 1-58113-879-2. doi: 10.1145/1031495.1031506. URL http://doi.acm.org/10.1145/1031495.1031506.

35 Atsushi Igarashi and Naoki Kobayashi. A generic type system for the pi-calculus. *Theor. Comput. Sci.*, 311(1-3):121–163, 2004.

36 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.

37 Antonio J Jara, Pedro Martinez-Julia, and Antonio Skarmeta. Light-weight multicast dns and dns-sd (lmdns-sd): Ipv6-based resource and service discovery for the web of things. In *2012 Sixth international conference on innovative mobile and internet services in ubiquitous computing*, pages 731–738. IEEE, 2012.

38 Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In *FMCO*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2010.

39 Eduard Kamburjan, Crystal Chang Din, and Tzu-Chun Chen. Session-based compositional analysis for actor-based languages using futures. In *ICFEM*, volume 10009 of *Lecture Notes in Computer Science*, pages 296–312, 2016.

40 Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. Typechecking protocols with Mungo and StMungo. In *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming (PPDP)*, pages 146–159. ACM, 2016. doi: 10.1145/2967973.2968595. URL http://doi.acm.org/10.1145/2967973.

41 Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. Typechecking protocols with Mungo and StMungo: a session type toolchain for Java. *Science of Computer Programming*, 155:52–75, 2018. ISSN 0167-6423. doi: https://doi.org/10.1016/j.scico.2017.10.006. URL http://www.sciencedirect.com/science/article/pii/S0167642317302186.

42 Paul Blain Levy, John Power, and Hayo Thielecke. Modelling environments in call-by-value programming languages. *Information and Computation*, 185(2):182–210, 2003.

43 Robin Milner. *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press, 1999.

44 Paul V Mockapetris. RFC 1035: Domain names - implementation and specification, 1987.

45 Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. Composing services with JOLIE. In *ECOWS*, pages 13–22. IEEE Computer Society, 2007.

46 Dimitris Mostrous and Vasco T. Vasconcelos. Affine sessions. *Log. Methods Comput. Sci.*, 14(4), 2018.

47 Dimitris Mostrous and Vasco Thudichum Vasconcelos. Session typing for a featherweight Erlang. In *Proceedings of the 13th International Conference on Coordination Models and Languages (COORDINATION)*, volume 6721 of *Lecture Notes in Computer Science*, pages 95–109. Springer, 2011. doi: 10.1007/978-3-642-21464-6_7. URL https://doi.org/10.1007/978-3-642-21464-6_7.

48 Rumyana Neykova and Nobuko Yoshida. Multiparty session actors. In *Proceedings of the 16th IFIP WG 6.1 Conference on Coordination Models and Languages (COORDINATION)*, volume 8459 of *Lecture Notes in Computer Science*, pages 131–146. Springer, 2014. doi: 10.1007/978-3-662-43376-8_9. URL https://doi.org/10.1007/978-3-662-43376-8_9.

49 Rumyana Neykova and Nobuko Yoshida. Multiparty session actors. *Logical Methods in Computer Science*, 13(1:17):1–30, 2017. doi: 10.23638/LMCS-13(1:17)2017. URL https://doi.org/10.23638/LMCS-13(1:17)2017.

50 Rumyana Neykova and Nobuko Yoshida. Let it recover: multiparty protocol-induced recovery. In *Proceedings of the 26th International Conference on Compiler Construction (CC)*, pages 98–108. ACM, 2017. URL http://dl.acm.org/citation.cfm?id=3033031.

51 Barry Porter, Matthew Grieves, Roberto Rodrigues Filho, and David Leslie. REX: A development platform and online learning approach for runtime emergent software systems. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 333–348. USENIX Association, 2016. ISBN 978-1-931971-33-1. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/porter.

52 Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. Dynamic choreographies: Theory and implementation. *Logical Methods in Computer Science*, 13(2), 2017. doi: 10.23638/LMCS-13(2:1)2017. URL https://doi.org/10.23638/LMCS-13(2:1)2017.

53 Jan S Rellermeyer, Gustavo Alonso, and Timothy Roscoe. R-osgi: distributed applications through software modularization. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 1–20. Springer, 2007.

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

55 Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. A linear decomposition of multiparty sessions for safe distributed programming. In *Proceedings of the 31st European Conference on Object-Oriented Programming (ECOOP)*, volume 74 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:31, 2017. doi: 10.4230/LIPIcs.ECOOP.2017.24. URL https://doi.org/10.4230/LIPIcs.ECOOP.2017.24.

56 Filippo Visintainer, Leandro D'Orazio, Marco Darin, and Luciano Altomare. Cooperative systems in motorway environment: The example of trento test site in italy. In Jan Fischer-Wolfarth and Gereon Meyer, editors, *Advanced Microsystems for Automotive Applications 2013*, pages 147–158, Heidelberg, 2013. Springer International Publishing. ISBN 978-3-319-00476-1.

57 Feng Xia, Laurence T. Yang, Lizhe Wang, and Alexey V. Vinel. Internet of things. *International Journal of Communication Systems*, 25(9):1101–1102, 2012. doi: 10.1002/dac.2417. URL https://doi.org/10.1002/dac.2417.

58 Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The Scribble protocol language. In *Proceedings of the 8th International Symposium on Trustworthy Global Computing (TGC)*, volume 8358, pages 22–41. Springer, 2014. ISBN 978-3-319-05118-5. doi: 10.1007/978-3-319-05119-2_3. URL https://doi.org/10.1007/978-3-319-05119-2_3.

REFERENCES

1042

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 $\mathsf{r} \to \mathsf{s}\{\ell_i : G_i\}_i$, where a single role makes a choice, this presentation of global types provides arbitrary summations of *global actions*. The standard sending action is therefore $\mathsf{p} \to \mathsf{q} : \ell(A)\,.\,G$ which involves role p sending message $\ell(A)$ to q.

A key technical innovation is *explicit connection actions* ($\mathsf{p} \twoheadrightarrow \mathsf{q} : \ell(A)$) which establish a connection between two roles p and q by sending a label ℓ. Conversely, *disconnection* $\mathsf{p}\#\mathsf{q}$ 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 projection function $G \upharpoonright_\Phi \mathsf{r}$ projects global type G at role r, and is parameterised by a set of recursion variables Φ. The inclusion of the Φ 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 $\Sigma_{i \in I}(G_i) \upharpoonright_\Phi \mathsf{r}$. If $G_i \upharpoonright_\Phi \mathsf{r} = \mathsf{end}$ 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.

REFERENCES

Meta-level definitions

$$\begin{aligned} \mathsf{isOutput}(L) &\triangleq L \in \{\mathtt{p}!\ell(A), \mathtt{p}!!\ell(A)\} \\ \mathsf{isInput}(L) &\triangleq L \in \{\mathtt{p}?\ell(A), \mathtt{p}??\ell(A)\} \end{aligned}$$

Projection $G \upharpoonright \mathtt{r}$ $G \upharpoonright_\Phi$

$$\begin{aligned}
G \upharpoonright \mathtt{r} &= G \upharpoonright_\emptyset \mathtt{r} \\
(\mathtt{p} \to \mathtt{q} : \ell(A) \,.\, G) \upharpoonright_\Phi \mathtt{r} &= \begin{cases} \mathtt{q}!\ell(A) \,.\, (G \upharpoonright_\Phi \mathtt{r}) & \mathtt{r} = \mathtt{p} \\ \mathtt{p}?\ell(A) \,.\, (G \upharpoonright_\Phi \mathtt{r}) & \mathtt{r} = \mathtt{q} \\ G \upharpoonright_\Phi \mathtt{r} & \mathtt{r} \notin \{\mathtt{p}, \mathtt{q}\} \end{cases} \\
(\mathtt{p} \twoheadrightarrow \mathtt{q} : \ell(A) \,.\, G) \upharpoonright_\Phi \mathtt{r} &= \begin{cases} \mathtt{q}!!\ell(A) \,.\, (G \upharpoonright_\Phi \mathtt{r}) & \mathtt{r} = \mathtt{p} \\ \mathtt{p}??\ell(A) \,.\, (G \upharpoonright_\Phi \mathtt{r}) & \mathtt{r} = \mathtt{q} \\ G \upharpoonright_\Phi \mathtt{r} & \mathtt{r} \notin \{\mathtt{p}, \mathtt{q}\} \end{cases} \\
(\mathtt{p}\#\mathtt{q} \,.\, G) \upharpoonright_\Phi \mathtt{r} &= \begin{cases} \#{\downarrow}\mathtt{q} & \mathtt{r} = \mathtt{p} \text{ and } G \upharpoonright_\emptyset \mathtt{r} = \mathsf{end} \\ \#{\uparrow}\mathtt{p} \,.\, (G \upharpoonright_\emptyset \mathtt{r}) & \mathtt{r} = \mathtt{q} \\ G \upharpoonright_\Phi \mathtt{r} & \mathtt{r} \notin \{\mathtt{p}, \mathtt{q}\} \end{cases} \\
\Sigma_{i \in I}(G_i) \upharpoonright_\Phi \mathtt{r} &= \begin{cases} X & \forall i \in I.(G_i \upharpoonright_\Phi \mathtt{r}) = X \\ \mathsf{end} & \forall i \in I.(G_i \upharpoonright_\Phi \mathtt{r}) = \mathsf{end} \\ \Sigma_{j \in J \subseteq I}(L_j = G_j \upharpoonright_\Phi \mathtt{r}) & |I| > 1, |J| > 0, \text{ and} \\ & \forall k \in I \setminus J.G_k \upharpoonright_\Phi \mathtt{r} = \mathsf{end} \text{ or } X \in \Phi, \text{ and} \\ & \quad \text{either} \begin{cases} \forall j \in J.\mathsf{isOutput}(L_j) \\ \exists \mathtt{p}.\forall j \in J.\mathsf{isInput}(L_j) \wedge \mathsf{subj}(L_j) = \mathtt{p} \end{cases} \end{cases} \\
\mu X.G \upharpoonright_\Phi \mathtt{r} &= \begin{cases} \mathsf{end} & G \upharpoonright_{\Phi,X} \mathtt{r} = X' \text{ or } \mathsf{end} \\ G \upharpoonright_{\Delta,X} \mathtt{r} & \text{otherwise} \end{cases} \\
X \upharpoonright_\Phi \mathtt{r} &= X \\
\mathsf{end} \upharpoonright_\Phi \mathtt{r} &= \mathsf{end}
\end{aligned}$$

Figure 18 Global types and projection

B Proofs for Section 5

B.1 Preservation

▶ **Lemma 20** (Preservation (Terms)). *If* $\{T\}\ \Gamma \mid S \triangleright M{:}A \triangleleft S'$ *and* $M \longrightarrow_{\mathsf{M}} N$*, then* $\{T\}\ \Gamma \mid S \triangleright N{:}A \triangleleft S'$*.*

Proof. By induction on the derivation of $M \longrightarrow_{\mathsf{M}} N$. ◀

▶ **Lemma 21** (Preservation (Equivalence)). *If* $\Gamma; \Delta \vdash \mathcal{C}$ *and* $\mathcal{C} \equiv \mathcal{D}$*, then* $\Gamma; \Delta \vdash \mathcal{D}$*.*

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

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

Proof. By induction on the derivation of $\{T\}\ \Gamma, x : A \mid S \triangleright M{:}B \triangleleft S'$. ◀

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

Proof. By induction on the structure of E. ◀

▶ **Lemma 24** (Replacement). *If:*

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

Then $\{U\}\ \Gamma \mid S \triangleright E[N]{:}A \triangleleft T$*.*

Proof. By induction on the structure of E. ◀

Due to the absence of exception handling frames which constrain the pre-condition to match that of the failure continuation, pure contexts admit a more liberal replacement lemma.

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

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

Then $\{U\}\ \Gamma \mid S' \triangleright E_P[N]{:}A \triangleleft T$*.*

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

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

Case E-Loop

$\langle a, \textbf{return}\ V, \bot, M\rangle \longrightarrow \langle a, M, \bot, M\rangle$

Assumption:

$$\frac{a : \mathsf{Pid}(T) \in \Gamma \qquad S = T \vee S = \mathsf{end} \qquad \{T\}\ \Gamma\ \mid\ S \triangleright \textbf{return}\ V{:}B \triangleleft \mathsf{end} \qquad \dfrac{\{T\}\ \Gamma\ \mid\ T \triangleright M{:}A \triangleleft \mathsf{end}}{\{T\}\ \Gamma \vdash M}}{\Gamma; a : T \vdash \langle a, \textbf{return}\ V, \bot, M\rangle}$$

Recomposing:

$$\frac{a : \mathsf{Pid}(T) \in \Gamma \qquad \{T\}\ \Gamma\ \mid\ T \triangleright M{:}A \triangleleft \mathsf{end} \qquad \dfrac{\{T\}\ \Gamma\ \mid\ T \triangleright M{:}A \triangleleft \mathsf{end}}{\{T\}\ \Gamma \vdash M}}{\Gamma; a : T \vdash \langle a, M, \bot, M\rangle}$$

as required.

Case E-New

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

where b is fresh, and $u.\mathsf{behaviour} = M$.

Assumption:

$$\frac{a : \mathsf{Pid}(S) \in \Gamma \qquad \{S\}\ \Gamma\ \mid\ S \triangleright E[\textbf{new}\ u]{:}A \triangleleft \mathsf{end} \qquad \{S\}\ \Gamma \vdash \kappa_1}{\Gamma; a : S, s[\mathsf{p}]\langle\tilde{\mathsf{q}}\rangle{:}S' \vdash \langle a, E[\textbf{new}\ u], s[\mathsf{p}]\langle\tilde{\mathsf{q}}\rangle, \kappa_1\rangle}$$

By Lemma 23:

$$\frac{u.\mathsf{sessionType} = T}{\{S\}\ \Gamma\ \mid\ S' \triangleright \textbf{new}\ u{:}\mathsf{Pid}(T) \triangleleft S'}$$

By Lemma 24, $\{S\}\ \Gamma, b : \mathsf{Pid}(T)\ \mid\ T \triangleright \textbf{return}\ b{:}\mathsf{Pid}(T) \triangleleft S'$.
Let $\Gamma' = \Gamma, b : \mathsf{Pid}(T)$. Note that by weakening, everything typable under Γ is also typable under Γ'.
By T-Def:

$$\frac{\{T\}\ \cdot\ \mid\ T \triangleright M{:}B \triangleleft \mathsf{end}}{\vdash \textbf{actor}\ u\ \textbf{follows}\ T\ \{M\}}$$

Again by weakening, $\{T\}\ \Gamma'\ \mid\ T \triangleright M{:}B \triangleleft \mathsf{end}$.
Recomposing:

$$\frac{\dfrac{\dfrac{a : \mathsf{Pid}(S) \in \Gamma'}{\{S\}\ \Gamma'\ \mid\ S \triangleright E[\textbf{return}\ b]{:}A \triangleleft \mathsf{end}} \qquad \{S\}\ \Gamma' \vdash \kappa_1}{\Gamma'; a : S, s[\mathsf{p}]\langle\tilde{\mathsf{q}}\rangle{:}S' \vdash \langle a, E[\textbf{return}\ b], s[\mathsf{p}]\langle\tilde{\mathsf{q}}\rangle, \kappa_1\rangle} \qquad \frac{b : \mathsf{Pid}(T) \in \Gamma' \qquad \{T\}\ \Gamma'\ \mid\ T \triangleright M{:}B \triangleleft \mathsf{end} \qquad \dfrac{\{T\}\ \Gamma'\ \mid\ T \triangleright M{:}B \triangleleft \mathsf{end}}{\{T\}\ \Gamma' \vdash M}}{\Gamma'; b : T \vdash \langle b, M, \bot, M\rangle}}{\Gamma'; a : S, b : S \vdash \langle a, E[\textbf{return}\ b], s[\mathsf{p}]\langle\tilde{\mathsf{q}}\rangle, \kappa_1\rangle \parallel \langle b, M, \bot, M\rangle}}{\Gamma; a : S \vdash (\nu b)(\langle a, E[\textbf{return}\ b], s[\mathsf{p}]\langle\tilde{\mathsf{q}}\rangle, \kappa_1\rangle \parallel \langle b, M, \bot, M\rangle)}$$

as required.

Case E-Replace

$\langle a, E[\textbf{replace } b \textbf{ with } \kappa_2'], \sigma, \kappa_1\rangle \parallel \langle b, M, \sigma, \kappa_2\rangle$

Assumption:

$$\frac{\dfrac{a : \mathsf{Pid}(U_a) \in \Gamma \quad \{U_a\}\ \Gamma\ \mid\ S \triangleright E[\textbf{replace } b \textbf{ with } \kappa_2']{:}A \triangleleft \mathsf{end} \qquad \{U_a\}\ \Gamma \vdash \kappa_1}{\Gamma; a : U_a, s[\mathsf{p}]\langle S\rangle{:}\tilde{\mathsf{r}} \vdash \langle a, E[\textbf{replace } b \textbf{ with } \kappa_2'], \sigma, s[\mathsf{p}]\langle\tilde{\mathsf{r}}\rangle\rangle} \qquad \frac{b : \mathsf{Pid}(U_b) \in \Gamma \qquad \{U_b\}\ \Gamma\ \mid\ S \triangleright M{:}B \triangleleft \mathsf{end} \qquad \{U_b\}\ \Gamma \vdash \kappa_2}{\Gamma; b : U_b, t[\mathsf{q}]\langle\tilde{\mathsf{s}}\rangle{:}T\tilde{\mathsf{s}} \vdash \langle b, M, t[\mathsf{q}]\langle\tilde{\mathsf{s}}\rangle, \kappa_2\rangle}}{\Gamma; a : U_a, b : U_b, s[\mathsf{p}]\langle\tilde{\mathsf{r}}\rangle{:}S, t[\mathsf{q}]\langle\tilde{\mathsf{s}}\rangle{:}T \vdash \langle a, E[\textbf{replace } b \textbf{ with } \kappa_2'], \sigma, s[\mathsf{p}]\langle\tilde{\mathsf{r}}\rangle\rangle \parallel \langle b, M, t[\mathsf{q}]\langle\tilde{\mathsf{s}}\rangle, \kappa_2\rangle}$$

By Lemma 23:

$$\frac{\{U_b\}\ \Gamma \vdash \kappa_2' \qquad \Gamma \vdash b{:}\mathsf{Pid}(U_b)}{\{U_a\}\ \Gamma\ \mid\ S \triangleright \textbf{replace } b \textbf{ with } \kappa_2'{:}\mathbf{1} \triangleleft S}$$

By Lemma 24, $\{U_a\}\ \Gamma\ \mid\ S \triangleright E[\textbf{return } ()]{:}A \triangleleft \mathsf{end}$.
Recomposing:

$$\frac{\dfrac{a : \mathsf{Pid}(U_a) \in \Gamma \quad \{U_a\}\ \Gamma\ \mid\ S \triangleright E[\textbf{return } ()]{:}A \triangleleft \mathsf{end} \qquad \{U_a\}\ \Gamma \vdash \kappa_1}{\Gamma; a : U_a, s[\mathsf{p}]\langle S\rangle{:}\tilde{\mathsf{r}} \vdash \langle a, E[\textbf{return } ()], \sigma, s[\mathsf{p}]\langle\tilde{\mathsf{r}}\rangle\rangle} \qquad \frac{b : \mathsf{Pid}(U_b) \in \Gamma \qquad \{U_b\}\ \Gamma\ \mid\ S \triangleright M{:}B \triangleleft \mathsf{end} \qquad \{U_b\}\ \Gamma \vdash \kappa_2'}{\Gamma; b : U_b, t[\mathsf{q}]\langle T\rangle{:}\tilde{\mathsf{r}} \vdash \langle b, M, t[\mathsf{q}]\langle\tilde{\mathsf{s}}\rangle, \kappa_2'\rangle}}{\Gamma; a : U_a, b : U_b, s[\mathsf{p}]\langle\tilde{\mathsf{r}}\rangle{:}S, t[\mathsf{q}]\langle\tilde{\mathsf{s}}\rangle{:}T \vdash \langle a, E[\textbf{return } ()], \sigma, s[\mathsf{p}]\langle\tilde{\mathsf{r}}\rangle\rangle \parallel \langle b, M, t[\mathsf{q}]\langle\tilde{\mathsf{s}}\rangle, \kappa_2'\rangle}$$

as required.

Case E-ReplaceSelf

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

Assumption:

$$\frac{a : \mathsf{Pid}(T) \in \Gamma \qquad \{T\}\ \Gamma\ \mid\ S \triangleright E[\textbf{replace } a \textbf{ with } \kappa']{:}A \triangleleft \mathsf{end} \qquad \{T\}\ \Gamma \vdash \kappa}{\Gamma; a : T, s[\mathsf{p}]\langle\tilde{\mathsf{q}}\rangle{:}S \vdash \langle a, E[\textbf{replace } a \textbf{ with } \kappa'], s[\mathsf{p}]\langle\tilde{\mathsf{q}}\rangle, \kappa\rangle}$$

By Lemma 23:

$$\frac{\{T\}\ \Gamma \vdash \kappa' \qquad \Gamma \vdash a{:}\mathsf{Pid}(T)}{\{T\}\ \Gamma\ \mid\ S \triangleright \textbf{replace } a \textbf{ with } \kappa'{:}\mathbf{1} \triangleleft S}$$

(noting that $\Gamma \vdash b{:}\mathsf{Pid}(T)$ because $a : \mathsf{Pid}(T) \in \Gamma$, as per the T-ConnectedActor and T-DisconnectedActor rules).
By Lemma 24, $\{T\}\ \Gamma\ \mid\ S \triangleright E[\textbf{return } ()]{:}A \triangleleft \mathsf{end}$.

Recomposing:

$$\frac{a : \mathsf{Pid}(T) \in \Gamma \qquad \{T\}\ \Gamma\ \mid\ S \triangleright E[\mathbf{return}\ ()]{:}A \triangleleft \mathsf{end} \qquad \{T\}\ \Gamma \vdash \kappa'}{\Gamma; a : T, s[\mathbf{p}]\langle S\rangle{:}\tilde{\mathbf{q}} \vdash \langle a, E[\mathbf{return}\ ()], s[\mathbf{p}]\langle\tilde{\mathbf{q}}\rangle, \kappa'\rangle}$$

as required.

Case E-Discover

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

where $b.\mathsf{sessionType} = T$ and $\neg(M = \mathbf{return}\ V \wedge \kappa_2 = \mathbf{stop})$.
Assumption:

$$\frac{\dfrac{a : \mathsf{Pid}(S) \in \Gamma \qquad \{S\}\ \Gamma\ \mid\ S' \triangleright E[\mathbf{discover}\ T]{:}A \triangleleft \mathsf{end} \qquad \{S\}\ \Gamma \vdash \kappa_1}{\Gamma; a : S, s[\mathbf{p}]\langle\tilde{\mathbf{r}}\rangle{:}S' \vdash \langle a, E[\mathbf{discover}\ T], s[\mathbf{p}]\langle\tilde{\mathbf{r}}\rangle, \kappa_1\rangle} \qquad \dfrac{b : \mathsf{Pid}(T) \in \Gamma \qquad \{T\}\ \Gamma\ \mid\ T' \triangleright M{:}B \triangleleft \mathsf{end} \qquad \{T\}\ \Gamma \vdash \kappa_2}{\Gamma; b : T, t[\mathbf{q}]\langle T'\rangle{:} \vdash \langle b, M, t[\mathbf{q}]\langle\tilde{\mathbf{s}}\rangle, \kappa_2\rangle}}{\Gamma; a : S, b : T, s[\mathbf{p}]\langle\tilde{\mathbf{r}}\rangle{:}S', t[\mathbf{q}]\langle\tilde{\mathbf{s}}\rangle{:}T' \vdash \langle a, E[\mathbf{discover}\ S], \sigma_1, \kappa_1\rangle \parallel \langle b, M, \sigma_2, \kappa_2\rangle}$$

By Lemma 23:

$$\frac{}{\{T\}\ \Gamma\ \mid\ S' \triangleright \mathbf{discover}\ T{:}\mathsf{Pid}(T) \triangleleft S'}$$

Since $b : \mathsf{Pid}(T) \in \Gamma$, we can show $\{T\}\ \Gamma\ \mid\ S' \triangleright \mathbf{discover}\ T{:}\mathsf{Pid}(T) \triangleleft S'$.
By Lemma 24, $\{T\}\ \Gamma\ \mid\ S' \triangleright E[\mathbf{return}\ b]{:}A \triangleleft S'$.
Thus, recomposing:

$$\frac{\dfrac{a : \mathsf{Pid}(S) \in \Gamma \qquad \{S\}\ \Gamma\ \mid\ S' \triangleright E[\mathbf{return}\ b]{:}A \triangleleft \mathsf{end} \qquad \{S\}\ \Gamma \vdash \kappa_1}{\Gamma; a : S, s[\mathbf{p}]\langle\tilde{\mathbf{r}}\rangle{:}S' \vdash \langle a, E[\mathbf{discover}\ T], s[\mathbf{p}]\langle\tilde{\mathbf{r}}\rangle, \kappa_1\rangle} \qquad \dfrac{b : \mathsf{Pid}(T) \in \Gamma \qquad \{T\}\ \Gamma\ \mid\ T' \triangleright M{:}B \triangleleft \mathsf{end} \qquad \{T\}\ \Gamma \vdash \kappa_2}{\Gamma; b : T, t[\mathbf{q}]\langle\tilde{\mathbf{s}}\rangle{:}T' \vdash \langle b, M, t[\mathbf{q}]\langle\tilde{\mathbf{s}}\rangle, \kappa_2\rangle}}{\Gamma; a : S, b : T, s[\mathbf{p}]\langle\tilde{\mathbf{r}}\rangle{:}S', s[\mathbf{q}]\langle\tilde{\mathbf{s}}\rangle{:}T' \vdash \langle a, E[\mathbf{return}\ b], s[\mathbf{p}]\langle\tilde{\mathbf{r}}\rangle, \kappa_1\rangle \parallel \langle b, M, t[\mathbf{q}]\langle\tilde{\mathbf{s}}\rangle, \kappa_2\rangle}$$

as required.

Case E-Self

$$\langle a, E[\mathbf{self}], \sigma, \kappa\rangle \longrightarrow \langle a, E[\mathbf{self}], \sigma, \kappa\rangle$$

Assumption:

$$\frac{a : \mathsf{Pid}(T) \in \Gamma \qquad \{T\}\ \Gamma\ \mid\ S \triangleright E[\mathbf{self}]{:}A \triangleleft S}{\Gamma; a : T, s[\mathbf{p}]\langle\tilde{\mathbf{q}}\rangle{:}S \vdash \langle a, E[\mathbf{self}], s[\mathbf{p}]\langle\tilde{\mathbf{q}}\rangle, \kappa\rangle}$$

By Lemma 23:

$$\frac{}{\{T\}\ \Gamma\ \mid\ S \triangleright \mathbf{self}{:}\mathsf{Pid}(T) \triangleleft S}$$

We can show:

$$\frac{\Gamma \vdash a{:}\mathsf{Pid}(T)}{\{T\}\ \Gamma\ \mid\ S \triangleright \mathbf{return}\ a{:}\mathsf{Pid}(T) \triangleleft S}$$

By Lemma 24, $\{T\}\ \Gamma\ \mid\ S \triangleright E[\mathbf{return}\ a]{:}A \triangleleft \mathsf{end}$.
Recomposing:

$$\frac{a : \mathsf{Pid}(T) \in \Gamma \qquad \{T\}\ \Gamma\ \mid\ S \triangleright E[\mathbf{return}\ a]{:}A \triangleleft S}{\Gamma; a : T, s[\mathsf{p}]\langle\tilde{\mathsf{q}}\rangle{:}S \vdash \langle a, E[\mathbf{return}\ a], s[\mathsf{p}]\langle\tilde{\mathsf{q}}\rangle, \kappa\rangle}$$

as required.

Case E-ConnInit

$$\langle a, E[F[\mathbf{connect}\ \ell_j(V)\ \mathbf{to}\ b\ \mathbf{as}\ \mathsf{q}]], \bot, \kappa_1\rangle \parallel \langle b, E'[F[\mathbf{accept\ from}\ \mathsf{p}\ \{\ell_i(x_i) \mapsto M_i\}_{i\in I}]], \bot, \kappa_2\rangle \longrightarrow (\nu s)(\langle a, E[\mathbf{return}\ ()], s[\mathsf{p}]\langle\mathsf{q}\rangle, \kappa_1\rangle \parallel \langle b, E'[M_j\{V/x_i\}], s[\mathsf{q}]\langle\mathsf{p}\rangle, \kappa_2\rangle)$$

Assumption:

$$\frac{\dfrac{\begin{array}{c}a : \mathsf{Pid}(U_a) \in \Gamma \qquad S = U_a \vee S = \mathsf{end} \\ \{U_a\}\ \Gamma\ \mid\ S \triangleright E[\mathbf{connect}\ \ell_j(V)\ \mathbf{to}\ b\ \mathbf{as}\ \mathsf{q}]{:}A \triangleleft \mathsf{end} \\ \{U_a\}\ \Gamma \vdash \kappa_1\end{array}}{\Gamma; a : U_a \vdash \langle a, E[\mathbf{connect}\ \ell_j(V)\ \mathbf{to}\ b\ \mathbf{as}\ \mathsf{q}], \bot, \kappa_1\rangle} \qquad \frac{\begin{array}{c}b : \mathsf{Pid}(U_b) \in \Gamma \qquad T = U_b \vee T = \mathsf{end} \\ \{U_b\}\ \Gamma\ \mid\ T \triangleright E'[\mathbf{accept\ from}\ \mathsf{p}\ \{\ell_i(x_i) \mapsto M_i\}_{i\in I}{:}B \triangleleft \mathsf{end} \\ \{U_b\}\ \Gamma \vdash \kappa_2\end{array}}{\Gamma; b : U_b \vdash \langle b, E'[\mathbf{accept\ from}\ \mathsf{p}\ \{\ell_i(x_i) \mapsto M_i\}_{i\in I}], \bot, \kappa_2\rangle}}{\Gamma; a : U_a, b : U_b \vdash \langle a, E[\mathbf{connect}\ \ell_j(V)\ \mathbf{to}\ b\ \mathbf{as}\ \mathsf{q}], \bot, \kappa_1\rangle \parallel \langle b, E'[\mathbf{accept\ from}\ \mathsf{p}\ \{\ell_i(x_i) \mapsto M_i\}_{i\in I}], \bot, \kappa_2\rangle}$$

By Lemma 23, by case analysis on F, wee either have:

$$\frac{\mathsf{q}!!\ell_j(A_j)\,.\,S'_j \in \{S_i\}_i \qquad \Gamma \vdash b{:}T_b \qquad \mathsf{ty}(\mathsf{q}) = T_b}{\{U_a\}\ \Gamma\ \mid\ \Sigma_{i\in I}(S_i) \triangleright \mathbf{connect}\ \ell_j(V)\ \mathbf{to}\ b\ \mathbf{as}\ \mathsf{q}{:}\mathbf{1} \triangleleft S'_j}$$

or

$$\frac{\dfrac{\mathsf{q}!!\ell_j(A_j)\,.\,S'_j \in \{S_i\}_i \qquad \Gamma \vdash b{:}T_b \qquad \mathsf{ty}(\mathsf{q}) = T_b}{\{U_a\}\ \Gamma\ \mid\ \Sigma_{i\in I}(S_i) \triangleright \mathbf{connect}\ \ell_j(V)\ \mathbf{to}\ b\ \mathbf{as}\ \mathsf{q}{:}\mathbf{1} \triangleleft S'_j} \qquad \{U_a\}\ \Gamma\ \mid\ \Sigma_{i\in I}(S_i) \triangleright M{:}\mathbf{1} \triangleleft S'_j}{\{U_a\}\ \Gamma\ \mid\ \Sigma_{i\in I}(S_i) \triangleright \mathbf{try\ connect}\ \ell_j(V)\ \mathbf{to}\ b\ \mathbf{as}\ \mathsf{q}\ \mathbf{catch}\ M{:}\mathbf{1} \triangleleft S'_j}$$

.

Since the evaluation context will be discarded, WLOG we proceed assuming that $F = [\ ]$.
As $b : \mathsf{Pid}(U_b) \in \Gamma$, we have that $T_b = U_b$ and therefore that $\mathsf{ty}(\mathsf{q}) = U_b$.

Again by Lemma 23, $\{U_b\}\ \Gamma \mid \Sigma_{i\in I}(\mathbf{p}??\ell_k(C_k)\,.\,T_k) \triangleright \textbf{accept from } \mathbf{p}\ \{\ell_i(x_k) \mapsto M_k\}_{k\in K}{:}B' \triangleleft T'$.

By case analysis on F', we either have:

$$\frac{(\{U_b\}\ \Gamma, x : C_k \mid T_k \triangleright M_k{:}B' \triangleleft T')_{k\in K}}{\{U_b\}\ \Gamma \mid \Sigma_{i\in I}(\mathbf{p}??\ell_k(C_k)\,.\,T_k) \triangleright \textbf{accept from } \mathbf{p}\ \{\ell_i(x_k) \mapsto M_k\}_{k\in K}{:}B' \triangleleft T'}$$

or

$$\frac{\dfrac{(\{U_b\}\ \Gamma, x : C_k \mid T_k \triangleright M_k{:}B' \triangleleft T')_{k\in K}}{\{U_b\}\ \Gamma \mid \Sigma_{i\in I}(\mathbf{p}??\ell_k(C_k)\,.\,T_k) \triangleright \textbf{accept from } \mathbf{p}\ \{\ell_i(x_k) \mapsto M_k\}_{k\in K}{:}B' \triangleleft T'} \qquad \{U_b\}\ \Gamma \mid \Sigma_{i\in I}(\mathbf{p}??\ell_k(C_k)\,.\,T_k) \triangleright M{:}A \triangleleft T'}{\{U_b\}\ \Gamma \mid \Sigma_{i\in I}(\mathbf{p}??\ell_k(C_k)\,.\,T_k) \triangleright \textbf{try } (\textbf{accept from } \mathbf{p}\ \{\ell_i(x_k) \mapsto M_k\}_{k\in K}) \textbf{ catch } M{:}B' \triangleleft T'}$$

Again, we consider the first case.

Thus, $U_b = \textbf{accept from } \mathbf{p}\ \{\ell_i(x_k) \mapsto M_k\}_{k\in K}$.

We now need to introduce the new runtime typing environment for the new session. We begin with a singleton runtime typing environment $\{s[\mathbf{p}]\langle\emptyset\rangle{:}U_a\}$, and recall that $U_a = \{\Sigma_{i\in I}(S_i)\}$ and $\mathbf{q}!!\ell_j(A_j)\,.\,S'_j \in \{S_i\}_i$.

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

As a consequence of safety, we know that $C_j = A_j$.

We can then show a reduction on typing environments:

$$\frac{\exists j \in I.S_j = \mathbf{q}!!\ell_j(A_j)\,.\,S'_j \qquad \mathsf{ty}(\mathbf{q}) = \mathbf{p}??\ell_k(C_k)\,.\,T_k \qquad j \in K \qquad C_j = A_j}{\dfrac{s[\mathbf{p}]\langle\emptyset\rangle{:}\Sigma_{i\in I}(S_i) \xrightarrow{s:\mathbf{p}\twoheadrightarrow\mathbf{q}::\ell_j} s[\mathbf{p}]\langle\mathbf{q}\rangle{:}S'_j, s[\mathbf{q}]\langle\mathbf{p}\rangle{:}T_j}{s[\mathbf{p}]\langle\emptyset\rangle{:}\Sigma_{i\in I}(S_i) \xrightarrow{s:\mathbf{p}\twoheadrightarrow\mathbf{q}::\ell_j} s[\mathbf{p}]\langle\mathbf{q}\rangle{:}S'_j, s[\mathbf{q}]\langle\mathbf{p}\rangle{:}T_j}}$$

Since $s[\mathbf{p}]\langle\emptyset\rangle{:}\Sigma_{i\in I}(S_i) \longrightarrow s[\mathbf{p}]\langle\mathbf{q}\rangle{:}S'_j, s[\mathbf{q}]\langle\mathbf{p}\rangle{:}T_j$, it follows by safety that $\mathsf{safe}(s[\mathbf{p}]\langle\mathbf{q}\rangle{:}S'_j, s[\mathbf{q}]\langle\mathbf{p}\rangle{:}T_j)$.

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

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

Recomposing:

$$\frac{\dfrac{a : \mathsf{Pid}(U_a) \in \Gamma \qquad \{U_a\}\ \Gamma \mid S'_j \triangleright E[\textbf{return } ()]{:}A \triangleleft \mathsf{end} \qquad \{U_a\}\ \Gamma \vdash \kappa_1}{\Gamma; a : U_a, s[\mathbf{p}]\langle\mathbf{q}\rangle{:}S'_j \vdash \langle a, E[\textbf{return } ()], s[\mathbf{p}]\langle\mathbf{q}\rangle, \kappa_1\rangle} \qquad \dfrac{b : \mathsf{Pid}(U_b) \in \Gamma \qquad \{U_b\}\ \Gamma \mid T_j \triangleright E'[M_j\{V/x_j\}]{:}B \triangleleft \mathsf{end} \qquad \{U_b\}\ \Gamma \vdash \kappa_2}{\Gamma; b : U_b, s[\mathbf{p}]\langle\mathbf{p}\rangle{:}T_j \vdash \langle b, E'[M_j\{V/x_j\}], s[\mathbf{q}]\langle\mathbf{p}\rangle, \kappa_2\rangle}}{\Gamma; a : U_a, b : U_b, s[\mathbf{p}]\langle\mathbf{q}\rangle{:}S'_j, s[\mathbf{q}]\langle\mathbf{p}\rangle{:}T_j \vdash \langle a, E[\textbf{return } ()], s[\mathbf{p}]\langle\mathbf{q}\rangle, \kappa_1\rangle \parallel \langle b, E'[M_j\{V/x_j\}], s[\mathbf{q}]\langle\mathbf{p}\rangle T_j, \kappa_2\rangle}}{\Gamma; a : U_a, b : U_b \vdash (\nu s)(\langle a, E[\textbf{return } ()], s[\mathbf{p}]\langle\mathbf{q}\rangle, \kappa_1\rangle \parallel \langle b, E'[M_j\{V/x_j\}], s[\mathbf{q}]\langle\mathbf{p}\rangle, \kappa_2\rangle)}$$

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

$\langle a, E[\textbf{connect}\ \ell_j(V)\ \textbf{to}\ b\ \textbf{as}\ \mathsf{q}], s[\mathsf{p}]\langle\tilde{\mathsf{r}}\rangle, \kappa_1\rangle \parallel \langle b, E'[\textbf{accept from}\ \mathsf{p}\ \{\ell_i(x_i) \mapsto M_i\}_{i\in I}], \bot, \kappa_2\rangle \longrightarrow$
$\langle a, E[\textbf{return}\ ()], s[\mathsf{p}]\langle\tilde{\mathsf{r}}, \mathsf{q}\rangle, \kappa_1\rangle \parallel \langle b, E'[M_j\{V/x_i\}], s[\mathsf{q}]\langle\mathsf{p}\rangle, \kappa_2\rangle$

with $j \in K$.
Assumption:

$$\frac{\dfrac{a : \mathsf{Pid}(U_a) \in \Gamma \quad \{U_a\}\ \Gamma\ \mid\ S \triangleright E[\textbf{connect}\ \ell_j(V)\ \textbf{to}\ b\ \textbf{as}\ \mathsf{q}]{:}A \triangleleft \mathsf{end} \quad \{U_a\}\ \Gamma \vdash \kappa_1}{\Gamma; a : U_a, s[\mathsf{p}]\langle\tilde{\mathsf{r}}\rangle{:}S \vdash \langle a, E[\textbf{connect}\ \ell_j(V)\ \textbf{to}\ b\ \textbf{as}\ \mathsf{q}], s[\mathsf{p}]\langle\tilde{\mathsf{r}}\rangle, \kappa_1\rangle} \quad \frac{b : \mathsf{Pid}(U_b) \in \Gamma \quad T = U_b \vee T = \mathsf{end} \quad \{U_b\}\ \Gamma\ \mid\ T \triangleright E'[\textbf{accept from}\ \mathsf{p}\ \{\ell_i(x_i) \mapsto M_i\}_{i\in I}]{:}B \triangleleft \mathsf{end} \quad \{U_b\}\ \Gamma \vdash \kappa_2}{\Gamma; b : U_b \vdash \langle b, E'[\textbf{accept from}\ \mathsf{p}\ \{\ell_i(x_i) \mapsto M_i\}_{i\in I}], \bot, \kappa_2\rangle}}{\Gamma; a : U_a, b : U_b s[\mathsf{p}]\langle\tilde{\mathsf{r}}\rangle{:}S \vdash \langle a, E[\textbf{connect}\ \ell_j(V)\ \textbf{to}\ b\ \textbf{as}\ \mathsf{q}], s[\mathsf{p}]\langle\tilde{\mathsf{r}}\rangle, \kappa_1\rangle \parallel \langle b, E'[\textbf{accept from}\ \mathsf{p}\ \{\ell_i(x_i) \mapsto M_i\}_{i\in I}], \bot, \kappa_2\rangle}$$

where $j \in I$ and $\mathsf{safe}(a : U_a, b : U_b, s[\mathsf{p}]\langle\tilde{\mathsf{r}}\rangle{:}S)$.
Consider the subderivation $\{U_a\}\ \Gamma\ \mid\ S \triangleright E[\textbf{connect}\ \ell_j(V)\ \textbf{to}\ b\ \textbf{as}\ \mathsf{q}]{:}A \triangleleft \mathsf{end}$.
By Lemma 23:

$$\frac{\mathsf{q}!!\ell_j(A_j)\,.\,S'_j \in \{S_i\}_i \quad \Gamma \vdash b{:}\mathsf{Pid}(T_b) \quad \mathsf{ty}(\mathsf{q}) = T_b}{\{U_a\}\ \Gamma\ \mid\ \Sigma_{i\in I}(S_i) \triangleright \textbf{connect}\ \ell_j(V)\ \textbf{to}\ b\ \textbf{as}\ \mathsf{q}{:}\mathbf{1} \triangleleft S'_j}$$

Since $b : \mathsf{Pid}(U_b) \in \Gamma$, we have that $T_b = U_b$. We also deduce that $S = \Sigma_{i\in I}(S_i)$ where $\mathsf{q}!!\ell_j(A_j)\,.\,S'_j \in \{S_i\}_i$.
Also by Lemma 23:

$$\frac{(\{U_b\}\ \Gamma, x : C_k\ \mid\ T_k \triangleright M_k{:}B' \triangleleft T')_{k\in K}}{\{U_b\}\ \Gamma\ \mid\ \Sigma_{i\in I}(\mathsf{p}??\ell_k(C_k)\,.\,T_k) \triangleright \textbf{accept from}\ \mathsf{p}\ \{\ell_i(x_k) \mapsto M_k\}_{k\in K}{:}B' \triangleleft T'}$$

Thus, $U_b = \textbf{accept from}\ \mathsf{p}\ \{\ell_i(x_k) \mapsto M_k\}_{k\in K}$.
We can then show a reduction on typing environments:

$$\frac{\exists j \in I.S_j = \mathsf{q}!!\ell_j(A_j)\,.\,S'_j \quad \mathsf{ty}(\mathsf{q}) = \mathsf{p}??\ell_k(C_k)\,.\,T_k \quad j \in K \quad C_j = A_j}{s[\mathsf{p}]\langle\tilde{\mathsf{r}}\rangle{:}\Sigma_{i\in I}(S_i) \xRightarrow{s:\mathsf{p}\to\mathsf{q}::\ell_j} s[\mathsf{p}]\langle\tilde{\mathsf{r}}, \mathsf{q}\rangle{:}S'_j, s[\mathsf{q}]\langle\mathsf{p}\rangle{:}T_j}$$

(noting that as a consequence of safety, we know that $C_j = A_j$).
Since $s[\mathsf{p}]\langle\tilde{\mathsf{r}}\rangle{:}\Sigma_{i\in I}(S_i) \longrightarrow s[\mathsf{p}]\langle\tilde{\mathsf{r}}, \mathsf{q}\rangle{:}S'_j, s[\mathsf{q}]\langle\mathsf{p}\rangle{:}T_j$, it follows by safety that $\mathsf{safe}(s[\mathsf{p}]\langle\tilde{\mathsf{r}}, \mathsf{q}\rangle{:}S'_j, s[\mathsf{q}]\langle\mathsf{p}\rangle{:}T_j)$.
Noting that only top-level frames (i.e., F, F') can contain exception-handling frames, E, E' are pure. We can show $\{U_a\}\ \Gamma\ \mid\ S'_j \triangleright \textbf{return}\ ()$:$A \triangleleft S'_j$ and so by Lemma 25, $\{U_a\}\ \Gamma\ \mid\ S'_j \triangleright E[\textbf{return}\ ()]{:}\mathbf{1} \triangleleft \mathsf{end}$.
Similarly, we can show $\{U_b\}\ \Gamma\ \mid\ T_j \triangleright M_j\{V/x_j\}{:}B' \triangleleft T'$ and so by Lemma 25, $\{U_b\}\ \Gamma\ \mid\ T_j \triangleright E'[M_j\{V/x_j\}]{:}B \triangleleft \mathsf{end}$.
Recomposing:

$$\frac{\dfrac{a : \mathsf{Pid}(U_a) \in \Gamma \quad \{U_a\}\ \Gamma\ \mid\ S'_j \triangleright E[\textbf{connect}\ \ell_j(V)\ \textbf{to}\ b\ \textbf{as}\ \mathsf{q}]{:}A \triangleleft \mathsf{end} \quad \{U_a\}\ \Gamma \vdash \kappa_1}{\Gamma; a : U_a, s[\mathsf{p}]\langle\tilde{\mathsf{r}}, \mathsf{q}\rangle{:}S'_j \vdash \langle a, E[\textbf{return}\ ()], s[\mathsf{p}]\langle\tilde{\mathsf{r}}, \mathsf{q}\rangle, \kappa_1\rangle} \quad \frac{b : \mathsf{Pid}(U_b) \in \Gamma \quad \{U_b\}\ \Gamma\ \mid\ T_j \triangleright E'[M_j\{V/x_j\}]{:}B \triangleleft \mathsf{end} \quad \{U_b\}\ \Gamma \vdash \kappa_2}{\Gamma; b : U_b, s[\mathsf{q}]\langle\mathsf{p}\rangle{:}T_j \vdash \langle b, E'[M_j\{V/x_j\}], s[\mathsf{q}]\langle\mathsf{p}\rangle, \kappa_2\rangle}}{\Gamma; a : U_a, b : U_b, s[\mathsf{p}]\langle\tilde{\mathsf{r}}, \mathsf{q}\rangle{:}S'_j, s[\mathsf{q}]\langle\mathsf{p}\rangle{:}T_j \vdash \langle a, E[\textbf{return}\ ()], s[\mathsf{p}]\langle\tilde{\mathsf{r}}, \mathsf{q}\rangle, \kappa_1\rangle \parallel \langle b, E'[M_j\{V/x_j\}], s[\mathsf{q}]\langle\mathsf{p}\rangle, \kappa_2\rangle}$$

with $\mathsf{safe}(a : U_a, b : U_b, s[\mathsf{p}]\langle\tilde{\mathsf{r}}, \mathsf{q}\rangle{:}S'_j, s[\mathsf{q}]\langle\mathsf{p}\rangle{:}T_j)$ as required.

REFERENCES

Case E-Comm

Let $\Delta_1 = a : U_a, s[\mathbf{p}]\langle\tilde{\mathbf{r}}\rangle{:}S_a$ and $\Delta_2 = b : U_b, s[\mathbf{q}]\langle\tilde{\mathbf{s}}\rangle{:}S_b$.
Assumption:

$$\dfrac{\dfrac{a : \mathsf{Pid}(U_a) \in \Gamma \quad \{U_a\}\ \Gamma \mid S_a \triangleright E[\mathbf{send}\ \ell_j(V)\ \mathbf{to}\ \mathbf{q}]{:}A \triangleleft \mathsf{end} \quad \{U_a\}\ \Gamma \vdash \kappa_1}{\Gamma; \Delta_1 \vdash \langle a, E[\mathbf{send}\ \ell_j(V)\ \mathbf{to}\ \mathbf{q}], s[\mathbf{p}]\langle\tilde{\mathbf{r}}\rangle, \kappa_1\rangle} \quad \dfrac{b : \mathsf{Pid}(U_b) \in \Gamma \quad \{U_b\}\ \Gamma \mid S_b \triangleright E'[\mathbf{receive\ from}\ \mathbf{p}\ \{\ell_i(x_i) \mapsto M_i\}_{i\in I}]{:}B \triangleleft \mathsf{end} \quad \{U_b\}\ \Gamma \vdash \kappa_2}{\Gamma; \Delta_2 \vdash \langle b, E'[\mathbf{receive\ from}\ \mathbf{p}\ \{\ell_i(x_i) \mapsto M_i\}_{i\in I}], s[\mathbf{q}]\langle\tilde{\mathbf{s}}\rangle, \kappa_2\rangle}}{\Gamma; \Delta_1, \Delta_2 \vdash \langle a, E[\mathbf{send}\ \ell_j(V)\ \mathbf{to}\ \mathbf{q}], s[\mathbf{p}]\langle\tilde{\mathbf{r}}\rangle, \kappa_1\rangle \parallel \langle b, E'[\mathbf{receive\ from}\ \mathbf{p}\ \{\ell_i(x_i) \mapsto M_i\}_{i\in I}], s[\mathbf{q}]\langle\tilde{\mathbf{s}}\rangle, \kappa_2\rangle}$$

where $j \in I$ and $\mathsf{safe}(\Delta_1, \Delta_2)$.
Consider the subderivation $\{U_a\}\ \Gamma \mid S_a \triangleright E[\mathbf{send}\ \ell_j(V)\ \mathbf{to}\ \mathbf{q}]{:}A \triangleleft \mathsf{end}$. By Lemma 23:

$$\dfrac{\mathbf{q}!\ell_j(A_j) \in \{S_i\}_{i\in I} \qquad \Gamma \vdash V{:}A_j}{\{U_a\}\ \Gamma \mid \Sigma_{i\in I}(S_i) \triangleright \mathbf{send}\ \ell_j(V)\ \mathbf{to}\ \mathbf{q}{:}\mathbf{1} \triangleleft S_j}$$

Next, consider the subderivation $\{U_b\}\ \Gamma \mid S_b \triangleright E'[\mathbf{receive\ from}\ \mathbf{p}\ \{\ell_i(x_i) \mapsto M_i\}_{i\in I}]{:}B \triangleleft \mathsf{end}$. Again by Lemma 23:

$$\dfrac{(\{U_b\}\ \Gamma, x_k : B_k \mid T_k \triangleright M_i{:}B \triangleleft T')_{k\in K}}{\{U_b\}\ \Gamma \mid \Sigma_{k\in K}(\mathbf{p}?\ell_k(B_k)\,.\,T_k) \triangleright \mathbf{receive\ from}\ \mathbf{p}\ \{\ell_i(x_i) \mapsto M_i\}_{i\in I}{:}A \triangleleft T'}$$

From the assumption we know that $j \in I$. As a result of the two subderivations above, we can refine our definitions of Δ_1 and Δ_2:

- $\Delta_1 = a : U_a, s[\mathbf{p}]\langle\tilde{\mathbf{r}}\rangle{:}\Sigma_{i\in I}(S_i)$, where $\mathbf{q}!\ell_j(A_j)\,.\,S'_j \in \{S_i\}_{i\in I}$
- $\Delta_2 = b : U_b, s[\mathbf{q}]\langle\tilde{\mathbf{s}}\rangle{:}\Sigma_{k\in K}(\mathbf{p}?\ell_k(B_k)\,.\,T_k)$

Since $\mathsf{safe}(\Delta_1, \Delta_2)$, we have that $j \in K$, $B_j = A_j$.
Thus we can construct a reduction on typing environments:

$$\dfrac{\dfrac{s!\ell_j(A_j)\,.\,S'_j \in \{S_i\}_i}{s[\mathbf{p}]\langle\tilde{\mathbf{r}}\rangle{:}\Sigma_{i\in I}(S_i) \xrightarrow{s:\mathbf{p}!\mathbf{q}::\ell_j(A_j)} s[\mathbf{p}]\langle\tilde{\mathbf{r}}\rangle{:}S'_j} \quad \dfrac{j \in K}{s[\mathbf{q}]\langle\tilde{\mathbf{s}}\rangle{:}\Sigma_{k\in K}(\mathbf{p}?\ell_k(B_k)\,.\,T_k) \xrightarrow{s:\mathbf{q}?\mathbf{p}::\ell_j(A_j)} s[\mathbf{q}]\langle\tilde{\mathbf{s}}\rangle{:}T_j}}{s[\mathbf{p}]\langle\tilde{\mathbf{r}}\rangle{:}\Sigma_{i\in I}(S_i), s[\mathbf{q}]\langle\tilde{\mathbf{s}}\rangle{:}\Sigma_{k\in K}(\mathbf{p}?\ell_k(B_k)\,.\,T_k) \xRightarrow{s:\mathbf{p},\mathbf{q}::\ell_j} s[\mathbf{p}]\langle\tilde{\mathbf{r}}\rangle{:}S'_j, s[\mathbf{q}]\langle\tilde{\mathbf{s}}\rangle{:}T_j}}{a : U_a, b : U_b, s[\mathbf{p}]\langle\tilde{\mathbf{r}}\rangle{:}\Sigma_{i\in I}(S_i), s[\mathbf{q}]\langle\tilde{\mathbf{s}}\rangle{:}\Sigma_{k\in K}(\mathbf{p}?\ell_k(B_k)\,.\,T_k) \xRightarrow{s:\mathbf{p},\mathbf{q}::\ell_j} a : U_a, b : U_b, s[\mathbf{p}]\langle S'_j\rangle{:}, s[\mathbf{q}]\langle T_j\rangle{:}}$$

Let $\Delta' = a : U_a, b : U_b, s[\mathbf{p}]\langle S'_j\rangle{:}, s[\mathbf{q}]\langle T_j\rangle{:}$.
Since $\mathsf{safe}(\Delta_1, \Delta_2)$ and $\Delta_1, \Delta_2 \longrightarrow \Delta'$, by the definition of safety, $\mathsf{safe}(\Delta')$.
Noting that only top-level frames (i.e., F, F') can contain exception-handling frames, E, E' are pure. By Lemma 25, $\{U_a\}\ \Gamma \mid S'_j \triangleright E[\mathbf{return}\ ()]{:}A \triangleleft \mathsf{end}$.
By Lemma 22, $\{U_b\}\ \Gamma \mid T_j \triangleright M_j\{V/x_j\}{:}B \triangleleft T'$.
By Lemma 25, $\{U_b\}\ \Gamma \mid T_j \triangleright E'[M_j\{V/x_j\}]{:}B \triangleleft \mathsf{end}$.
Letting $\Delta'_1 = a : U_a, s[\mathbf{p}]\langle S'_j\rangle{:}$ and $\Delta'_2 = b : U_b, s[\mathbf{q}]\langle T_j\rangle{:}$, recomposing:

$$\dfrac{\dfrac{a : \mathsf{Pid}(U_a) \in \Gamma \quad \{U_a\}\ \Gamma \mid S_j \triangleright E[\mathbf{return}\ ()]{:}A \triangleleft \mathsf{end} \quad \{U_a\}\ \Gamma \vdash \kappa_1}{\Gamma; \Delta'_1 \vdash \langle a, E[\mathbf{return}\ ()], s[\mathbf{p}]\langle\tilde{\mathbf{r}}\rangle, \kappa_1\rangle} \quad \dfrac{b : \mathsf{Pid}(U_b) \in \Gamma \quad \{U_b\}\ \Gamma \mid T_j \triangleright E'[M_j\{V/x_j\}]{:}B \triangleleft \mathsf{end} \quad \{U_b\}\ \Gamma \vdash \kappa_2}{\Gamma; \Delta'_2 \vdash \langle b, E'[M_j\{V/x_j\}], s[\mathbf{q}]\langle\tilde{\mathbf{s}}\rangle, \kappa_2\rangle}}{\Gamma; \Delta' \vdash \langle a, E[\mathbf{return}\ ()], s[\mathbf{p}]\langle\tilde{\mathbf{r}}\rangle, \kappa_1\rangle \parallel \langle b, E'[M_j\{V/x_j\}], s[\mathbf{q}]\langle\tilde{\mathbf{s}}\rangle, \kappa_2\rangle}$$

with $\mathsf{safe}(\Delta')$, as required.

Case E-Disconn

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

Let $\Delta = a : U_a, s[\mathbf{p}]\langle \tilde{\mathbf{r}}\rangle{:}S, b : U_b, s[\mathbf{q}]\langle \mathbf{p}\rangle{:}T$.
Assumption:

$$\dfrac{\dfrac{\mathsf{Pid}(U_a) \in \Gamma \quad \{U_a\}\ \Gamma \mid S \triangleright E[\textbf{wait } \mathbf{q}]{:}B \triangleleft \mathsf{end} \quad \{U_a\}\ \Gamma \vdash \kappa_1}{\Gamma; a : U_a, s[\mathbf{p}]\langle \tilde{\mathbf{r}}, \mathbf{q}\rangle{:}S \vdash \langle a, E[\textbf{wait } \mathbf{q}], s[\mathbf{p}]\langle \tilde{\mathbf{r}}, \mathbf{q}\rangle, \kappa_1\rangle} \quad \dfrac{\mathsf{Pid}(U_b) \in \Gamma \quad \{U_b\}\ \Gamma \mid T \triangleright E'[\textbf{disconnect from } \mathbf{p}]{:}B \triangleleft \mathsf{end} \quad \{U_b\}\ \Gamma \vdash \kappa_2}{\Gamma; b : U_b, s[\mathbf{q}]\langle \mathbf{p}\rangle{:}T \vdash \langle b, E'[\textbf{disconnect from } \mathbf{p}], s[\mathbf{q}]\langle \mathbf{p}\rangle, \kappa_2\rangle}}{\Gamma; \Delta \vdash \langle a, E[\textbf{wait } \mathbf{q}], s[\mathbf{p}]\langle \tilde{\mathbf{r}}, \mathbf{q}\rangle, \kappa_1\rangle \parallel \langle b, E'[\textbf{disconnect from } \mathbf{p}], s[\mathbf{q}]\langle \mathbf{p}\rangle, \kappa_2\rangle}$$

By Lemma 23:

$$\dfrac{}{\{U_a\}\ \Gamma \mid \#{\uparrow}\mathbf{q} \,.\, S' \triangleright \textbf{wait } \mathbf{q}{:}\mathbf{1} \triangleleft S'}$$

Also by Lemma 23:

$$\dfrac{}{\{U_b\}\ \Gamma \mid \#{\downarrow}\mathbf{p} \triangleright \textbf{disconnect from } \mathbf{p}{:}\mathbf{1} \triangleleft T'}$$

Thus, $S = \#{\uparrow}\mathbf{q} \,.\, S'$ and $T = \#{\downarrow}\mathbf{q}$
We can show a reduction on runtime typing environments:

$$\dfrac{\dfrac{\dfrac{}{s[\mathbf{p}]\langle \tilde{\mathbf{r}}, \mathbf{q}\rangle{:}\#{\uparrow}\mathbf{q} \,.\, S' \xrightarrow{s:\mathbf{p}\#{\uparrow}\mathbf{q}} s[\mathbf{p}]\langle \tilde{\mathbf{r}}\rangle{:}S'} \quad \dfrac{}{s[\mathbf{q}]\langle \mathbf{p}\rangle{:}\#{\downarrow}\mathbf{p} \xrightarrow{s:\mathbf{q}\#{\downarrow}\mathbf{p}} \cdot}}{s[\mathbf{p}]\langle \tilde{\mathbf{r}}, \mathbf{q}\rangle{:}\#{\uparrow}\mathbf{q} \,.\, S', s[\mathbf{q}]\langle \mathbf{p}\rangle{:}\#{\downarrow}\mathbf{p} \xRightarrow{s:\mathbf{p}\#\mathbf{q}} S'}}{a : U_a, s[\mathbf{p}]\langle \tilde{\mathbf{r}}, \mathbf{q}\rangle{:}\#{\uparrow}\mathbf{q} \,.\, S', b : U_b, s[\mathbf{q}]\langle \mathbf{p}\rangle{:}\#{\downarrow}\mathbf{q} \xRightarrow{s:\mathbf{p}\#\mathbf{q}} a : U_a, s[\mathbf{p}]\langle \tilde{\mathbf{r}}\rangle{:}S', b : U_b}$$

Since $\mathsf{safe}(\Delta)$, and $\Delta \Longrightarrow \Delta'$, by the definition of safety, we have that $\mathsf{safe}(\Delta')$.
Noting that only top-level frames (i.e., F, F') can contain exception-handling frames, E, E' are pure. We can show that $\{U_a\}\ \Gamma \mid S' \triangleright \textbf{return } ()\text{:}\mathbf{1} \triangleleft S'$, so by Lemma 25, we have that $\{U_a\}\ \Gamma \mid S' \triangleright E[\textbf{return } ()]{:}A \triangleleft S'$.
By the same argument, $\{U_b\}\ \Gamma \mid \mathsf{end} \triangleright E'[\textbf{return } ()]{:}\mathbf{1} \triangleleft \mathsf{end}$.
Thus we can show (by T-UnconnectedActor, noting that end is a permissible precondition):

$$\dfrac{\mathsf{Pid}(U_a) \in \Gamma \quad \{U_a\}\ \Gamma \mid \mathsf{end} \triangleright E'[\textbf{return } ()]{:}B \triangleleft \mathsf{end} \quad \{U_a\}\ \Gamma \vdash \kappa_2}{\Gamma; b : U_a \vdash \langle b, E'[\textbf{return } ()], \bot, \kappa_2\rangle}$$

Recomposing:

$$\dfrac{\dfrac{a : \mathsf{Pid}(U_a) \in \Gamma \quad \{T\}\ \Gamma \mid S \triangleright E[\textbf{wait } \mathbf{q}]{:}B \triangleleft \mathsf{end} \quad \{T\}\ \Gamma \vdash \kappa_1}{\Gamma; a : U_a, s[\mathbf{p}]\langle \tilde{\mathbf{r}}\rangle{:}S' \vdash \langle a, E[\textbf{return } ()], s[\mathbf{p}]\langle \tilde{\mathbf{r}}\rangle, \kappa_1\rangle} \quad \dfrac{b : U_b \in \Gamma \quad \{U_b\}\ \Gamma \mid \mathsf{end} \triangleright E'[\textbf{return } ()]{:}B \triangleleft \mathsf{end} \quad \{U_b\}\ \Gamma \vdash \kappa_2}{\Gamma; b : U_b \vdash \langle b, E'[\textbf{return } ()], \bot, \kappa_2\rangle}}{\Gamma; a : U_a, s[\mathbf{p}]\langle \tilde{\mathbf{r}}\rangle{:}S', b : U_b \vdash \langle a, E[\textbf{return } ()], s[\mathbf{p}]\langle \tilde{\mathbf{r}}\rangle, \kappa_1\rangle \parallel \langle b, E'[\textbf{return } ()], \bot, \kappa_2\rangle}$$

with $\mathsf{safe}(a : U_a, s[\mathbf{p}]\langle\tilde{\mathbf{r}}\rangle{:}S', b : U_b)$, as required.

Case E-Complete

$(\nu s)(\langle a, \mathbf{return}\ V, s[\mathbf{p}]\langle\emptyset\rangle, \kappa\rangle) \longrightarrow \langle a, \mathbf{return}\ V, \bot, \kappa\rangle$

Assumption:

$$\frac{\dfrac{a : \mathsf{Pid}(T) \qquad \{T\}\ \Gamma\ \mid\ S \triangleright \mathbf{return}\ V{:}A \triangleleft \mathsf{end} \qquad \{T\}\ \Gamma \vdash \kappa}{\Gamma; a : T, s[\mathbf{p}]\langle\emptyset\rangle{:}S \vdash \langle a, \mathbf{return}\ V, s[\mathbf{p}]\langle\emptyset\rangle, \kappa\rangle}}{\Gamma; \cdot \vdash (\nu s)(\langle a, \mathbf{return}\ V, s[\mathbf{p}]\langle\emptyset\rangle, \kappa\rangle)}$$

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

$$\frac{a : \mathsf{Pid}(T) \qquad \{T\}\ \Gamma\ \mid\ \mathsf{end} \triangleright \mathbf{return}\ V{:}A \triangleleft \mathsf{end} \qquad \{T\}\ \Gamma \vdash \kappa}{\Gamma; a : T \vdash \langle a, \mathbf{return}\ V, s[\mathbf{p}]\langle\emptyset\rangle, \kappa\rangle}$$

as required.

Case E-CommRaise

$\langle a, E[M], s[\mathbf{p}]\langle\tilde{\mathbf{r}}\rangle, \kappa\rangle \parallel \lightning\, s[\mathbf{q}] \longrightarrow \langle a, E[\mathbf{raise}], s[\mathbf{p}]\langle\tilde{\mathbf{r}}\rangle, \kappa\rangle \parallel \lightning\, s[\mathbf{q}]$

where $\mathsf{subj}(M) = \mathbf{p}$.
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 $M = \mathbf{wait}\ \mathbf{q}$.
Assumption:

$$\frac{\dfrac{\{T\}\ \Gamma\ \mid\ \#{\uparrow}\mathbf{q}\,.\,S' \triangleright E[\mathbf{wait}\ \mathbf{q}]{:}A \triangleleft \mathsf{end} \qquad \{T\}\ \Gamma \vdash \kappa}{\Gamma; a : T, s[\mathbf{p}]\langle\tilde{\mathbf{r}}\rangle{:}\#{\uparrow}\mathbf{q}\,.\,S' \vdash \langle a, E[\mathbf{wait}\ \mathbf{q}], s[\mathbf{p}]\langle\tilde{\mathbf{r}}\rangle, \kappa\rangle} \qquad \dfrac{}{\Gamma; s[\mathbf{q}]\langle\tilde{\mathbf{s}}\rangle{:}U \vdash \lightning\, s[\mathbf{q}]}}{\Gamma; a : T, s[\mathbf{p}]\langle\tilde{\mathbf{r}}\rangle{:}\#{\uparrow}\mathbf{q}\,.\,S's[\mathbf{q}]\langle\tilde{\mathbf{s}}\rangle{:}U \vdash \langle a, E[\mathbf{wait}\ \mathbf{q}], s[\mathbf{p}]\langle\tilde{\mathbf{r}}\rangle, \kappa\rangle \parallel \lightning\, s[\mathbf{q}]}$$

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

$$\frac{}{\{T\}\ \Gamma\ \mid\ \#{\uparrow}\mathbf{q}\,.\,S' \triangleright \mathbf{wait}\ \mathbf{q}{:}\mathbf{1} \triangleleft S''}$$

By T-Raise, **raise** can have any precondition, return type, and postcondition. Therefore, by Lemma 25, $\{T\}\ \Gamma\ \mid\ \#{\uparrow}\mathbf{q} \triangleright E[\mathbf{raise}]{:}A \triangleleft \mathsf{end}$, and therefore:

$$\frac{\dfrac{\{T\}\ \Gamma\ \mid\ \#{\uparrow}\mathbf{q}\,.\,S' \triangleright E[\mathbf{raise}]{:}A \triangleleft \mathsf{end} \qquad \{T\}\ \Gamma \vdash \kappa}{\Gamma; a : T, s[\mathbf{p}]\langle\tilde{\mathbf{r}}\rangle{:}\#{\uparrow}\mathbf{q}\,.\,S' \vdash \langle a, E[\mathbf{wait}\ \mathbf{q}], s[\mathbf{p}]\langle\tilde{\mathbf{r}}\rangle, \kappa\rangle} \qquad \dfrac{}{\Gamma; s[\mathbf{q}]\langle\tilde{\mathbf{s}}\rangle{:}U \vdash \lightning\, s[\mathbf{q}]}}{\Gamma; a : T, s[\mathbf{p}]\langle\tilde{\mathbf{r}}\rangle{:}\#{\uparrow}\mathbf{q}\,.\,S's[\mathbf{q}]\langle\tilde{\mathbf{s}}\rangle{:}U \vdash \langle a, E[\mathbf{raise}], s[\mathbf{p}]\langle\tilde{\mathbf{r}}\rangle, \kappa\rangle \parallel \lightning\, s[\mathbf{q}]}$$

as required.

Case E-FailS

$\langle a, P[\mathbf{raise}], s[\mathbf{p}]\langle\tilde{\mathbf{r}}\rangle, \kappa\rangle \longrightarrow \langle a, \mathbf{raise}, \bot, \kappa\rangle \parallel \lightning\, s[\mathbf{p}]$

Assumption:

$$\frac{a : \mathsf{Pid}(T) \in \Gamma \qquad \{T\}\ \Gamma \mid S \triangleright P[\mathbf{raise}]{:}A \triangleleft \mathsf{end} \qquad \{T\}\ \Gamma \vdash \kappa}{\Gamma; a : T, s[\mathbf{p}]\langle\tilde{\mathbf{q}}\rangle{:}S \vdash \langle a, P[\mathbf{raise}], s[\mathbf{p}]\langle\tilde{\mathbf{q}}\rangle, \kappa\rangle}$$

Since **raise** is typable under any precondition and postcondition, and has an arbitrary return type, it follows that $\{T\}\ \Gamma \mid \mathsf{end} \triangleright \mathbf{raise}{:}A \triangleleft \mathsf{end}$.

Recomposing:

$$\frac{\dfrac{a : \mathsf{Pid}(T) \in \Gamma \qquad \{T\}\ \Gamma \mid \mathsf{end} \triangleright \mathbf{raise}{:}A \triangleleft \mathsf{end} \qquad \{T\}\ \Gamma \vdash \kappa}{\Gamma; a : T \vdash \langle a, \mathbf{raise}, s[\mathbf{p}]\langle\tilde{\mathbf{q}}\rangle, \kappa\rangle} \qquad \dfrac{}{\Gamma; s[\mathbf{p}]\langle\tilde{\mathbf{q}}\rangle{:}S \vdash \lightning\, s[\mathbf{p}]}}{\Gamma; a : T, s[\mathbf{p}]\langle\tilde{\mathbf{q}}\rangle{:}S \vdash \langle a, \mathbf{raise}, \bot, \kappa\rangle \parallel \lightning\, s[\mathbf{p}]}$$

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. ◀

C Progress

C.1 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).

We refine our notion of environment progress to include the possibility of failed sessions (Definition 35), and show that given a runtime environment Δ satisfying safety and progress, a derived exception-aware runtime environment Θ 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 restict 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 Ψ range over typing environments containing only runtime names: $\Psi ::= \cdot \mid \Psi, a : \mathsf{Pid}(S)$.

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

▶ **Lemma 26** (Progress (terms)). *If $\{T\}\ \Psi \mid S \triangleright M : A \triangleleft S'$, then either $M = \mathbf{return}\, V$ for some value V; there exists some N such that $M \longrightarrow_{\mathsf{M}} N$; 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.*

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

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

C.3 Canonical forms

▶ **Lemma 15** (Canonical forms). *If $\cdot;\cdot \vdash \mathcal{C}$, then $\exists \mathcal{D} \equiv \mathcal{C}$ where $\mathcal{D}$ 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 $\lightning\, s[\mathbf{p}]$, any other participant in the remainder of the session blocked on $\mathbf{p}$ can reduce. In order to account for this, we introduce the notion of exception-aware runtime typing environments, environment reduction, and runtime typing.

▶ **Definition 28** (Exception-aware runtime typing environments, environment reduction, and runtime typing). *An* exception-aware runtime typing environment Θ *is defined as follows:*

$$\Theta \quad ::= \quad a : S \mid s[\mathbf{p}]\langle\tilde{\mathbf{q}}\rangle{:}S \mid s[\mathbf{p}] : \lightning$$

The exception-aware runtime typing relation $\Gamma;\Theta \vdash_{\lightning} \mathcal{C}$ *is defined to be the standard runtime typing relation $\Gamma;\Delta \vdash \mathcal{C}$ but with rule T-Zap defined as:*

$$\textit{T-Zap}\quad \frac{}{\Gamma; s[\mathtt{p}] : \lightning \vdash_{\lightning} \lightning s[\mathtt{p}]}$$

We extend the set of labels ρ with a zapper label $s : \mathtt{p} \lightning \mathtt{q}$, which states that role $\mathtt{q}$ has attempted to interact with a cancelled role $\mathtt{p}$ and has become cancelled itself as a result.

The exception-aware environment reduction relation *is defined as the rules of the standard runtime typing relation $\Delta \Longrightarrow \Delta'$ with the addition of the following rules:*

$$\textit{ET-CommFail}\quad \frac{\exists j \in I . S_j = \mathtt{p}\dagger\ell_j(A_j) \,.\, S'_j}{s[\mathtt{p}] : \lightning, s[\mathtt{q}]\langle\tilde{\mathtt{r}}\rangle{:}\Sigma_{i\in I}(S_i) \xrightarrow{s:\mathtt{p}\lightning\mathtt{q}} s[\mathtt{p}] : \lightning, s[\mathtt{q}] : \lightning}$$

$$\textit{ET-DisconnFail}\quad \frac{}{s[\mathtt{p}] : \lightning, s[\mathtt{q}]\langle\mathtt{p}\rangle{:}\#{\downarrow}\mathtt{q} \xrightarrow{s:\mathtt{p}\lightning\mathtt{q}} s[\mathtt{p}] : \lightning, s[\mathtt{q}] : \lightning}$$

$$\textit{ET-WaitFail}\quad \frac{}{s[\mathtt{p}] : \lightning, s[\mathtt{q}]\langle\mathtt{p}\rangle{:}\#{\uparrow}\mathtt{q} \,.\, S \xrightarrow{s:\mathtt{p}\lightning\mathtt{q}} s[\mathtt{p}] : \lightning, s[\mathtt{q}] : \lightning}$$

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

$$\mathcal{S} = (\nu s)(\langle a_1, M_1, s[\mathtt{p}_1]\langle\tilde{\mathtt{q}}_1\rangle, \kappa_1\rangle \parallel \cdots \parallel \langle a_m, M_m, s[\mathtt{p}_m]\langle\tilde{\mathtt{q}}_m\rangle, \kappa_m\rangle \parallel \lightning s[\mathtt{p}_{m+1}] \parallel \cdots \parallel \lightning s[\mathtt{p}_n])$$

as $\mathtt{p}_{m+1}, \ldots, \mathtt{p}_n$.

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

$$\Delta = a_1 : S_1, \ldots, a_l : S_l, s[\mathtt{p}_1]\langle\tilde{\mathtt{r}}_1\rangle{:}T_1, \ldots s[\mathtt{p}_m]\langle\tilde{\mathtt{r}}_m\rangle{:}T_m, s[\mathtt{q}_1]\langle\tilde{\mathtt{r}}_1\rangle{:}T'_1, \ldots s[\mathtt{q}_n]\langle\tilde{\mathtt{r}}_n\rangle{:}T'_n$$

and a set of roles $\tilde{\mathtt{q}} = \mathtt{q}_1, \ldots, \mathtt{q}_n$, the zapped environment $\Theta = \mathit{zap}(\Delta, \tilde{\mathtt{q}})$ *is defined as:*

$$\Delta = a_1 : S_1, \ldots, a_l : S_l, s[\mathtt{p}_1]\langle\tilde{\mathtt{r}}_1\rangle{:}T_1, \ldots s[\mathtt{p}_m]\langle\tilde{\mathtt{r}}_m\rangle{:}T_m, s[\mathtt{q}_1] : \lightning, \ldots s[\mathtt{q}_n] : \lightning$$

▶ **Definition 31** (Failed environment). *An exception-aware runtime typing environment Θ is a* failed environent*, written $\mathit{failed}(\Theta)$, if Θ is of the form $s[\mathtt{p}_1] : \lightning, \ldots, s[\mathtt{p}_n] : \lightning$.*

▶ **Lemma 32** (Zapped Typeability). *If $\Gamma; \Delta \vdash \mathcal{S}$ where $\mathcal{S} = (\nu s : \Delta')\mathcal{C}$, the zapped roles of $\mathcal{S}$ are $\tilde{\mathtt{p}}$, and $\Theta = \mathit{zap}(\Delta', \tilde{\mathtt{p}})$, then $\Gamma; \Delta, \Theta \vdash_{\lightning} \mathcal{C}$.*

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

▶ **Corollary 33.** *If $\cdot; \cdot \vdash \mathcal{C}$ where $\mathcal{C}$ is in canonical form, then $\cdot; \cdot \vdash_{\lightning} \mathcal{C}$.*

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

▶ **Corollary 34** (Exception-aware session typability). *If $\cdot; \cdot \vdash_{\lightning} \mathcal{G}[\mathcal{S}]$, then there exist Ψ, Δ such that Δ only consists of entries of the form $a : S$ and $\cdot; \cdot \vdash_{\lightning} \mathcal{S}$.*

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

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

- *(Role progress) for each $s[\mathtt{p}_i]\langle\tilde{\mathtt{q}}_i\rangle{:}S_i \in \Theta$ such that $\mathsf{active}(S_i)$, it is the case that $\Theta \longrightarrow^* \Theta' \xrightarrow{\gamma}$ with $\mathtt{p} \in \mathit{roles}(\gamma)$.*
- *(Eventual communication) if $\Theta \Longrightarrow^* \Theta' \xrightarrow{s:\mathtt{p}!\mathtt{q}::\ell(A)}$, then either $\Theta' \xRightarrow{\overrightarrow{\rho}} \Theta'' \xrightarrow{s:\mathtt{q}?\mathtt{p}::\ell(A)}$, or $\Theta' \xRightarrow{\overrightarrow{\rho}} \Theta'' \xrightarrow{s:\mathtt{q}\lightning\mathtt{p}}$, where $\mathtt{p} \notin \mathit{roles}(\overrightarrow{\rho})$*
- *(Correct termination) $\Theta \Longrightarrow^* \Theta' \not\Longrightarrow$ implies either $\mathsf{end}(\Theta)$ or $\mathsf{failed}(\Theta)$.*

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

▶ **Lemma 36** (Exception-aware environments preserve safety and progress). *Suppose* $\mathsf{safe}(\Delta)$ *and* $\mathsf{prog}(\Delta)$ *for some* Θ. *If* $\Theta = \mathsf{zap}(\Delta, \tilde{\mathtt{p}})$ *for some set of roles* $\tilde{\mathtt{p}}$, *then* $\mathsf{safe}(\Theta)$ *and* $\mathsf{prog}(\Theta)$.

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 $\mathsf{safe}(\Delta)$ then $\mathsf{safe}(\Theta)$.

Progress. For (role progress), we know that any active role in Δ 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-DisconnFail, 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 $\Delta \Longrightarrow^* \Delta' \xrightarrow{s:\mathtt{p}!\mathtt{q}::\ell(A)}$, then $\Delta' \xRightarrow{\vec{\rho}} \Delta'' \xrightarrow{s:\mathtt{q}?\mathtt{p}::\ell(A)} \Delta'''$, where $\mathtt{p} \notin \mathsf{roles}(\vec{\rho})$.

Suppose $\Theta \Longrightarrow^* \Theta' \xrightarrow{s:\mathtt{p}!\mathtt{q}::\ell(A)}$. If Θ' contains zapped roles, either they are irrelevant to $\mathtt{p}$'s reduction and we can use the original reduction sequence to show $\Theta \Longrightarrow^* \Theta'' \xrightarrow{s:\mathtt{q}?\mathtt{p}::\ell(A)} \Delta'''$, or they are relevant to reduction and the exception propagates, resulting in $\Theta' \xRightarrow{\vec{\rho}} \Theta'' \xRightarrow{s:\mathtt{q}\not\downarrow\mathtt{p}} \Delta'''$.

For (correct termination), we know that $\Delta \not\Longrightarrow$ implies that $\mathsf{end}(\Delta)$. By (1), we know that each active role in Θ will eventually reduce, so it follows that either $\mathsf{end}(\Theta)$ (if all communications occur successfully) or $\mathsf{failed}(\Theta)$ 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* $\Sigma_{i\in I}(\alpha_i \,.\, S_i)$ *is a* output-directed *if each* α_i *is either of the form* $\mathtt{p}_i!\ell_i(A_i)$.

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

For convenience, we write $S^\bullet$ *and* $\Sigma^\bullet_{i\in I}(\alpha_i \,.\, S_i)$ *to denote an output-directed choice, and* S° *and* $\Sigma^\circ_{i\in I}(\alpha_i \,.\, S_i)$ *to denote an input-directed choice.*

▶ **Definition 38** (Output-flat session type). *An output-directed session type* $\Sigma^\bullet_{i\in I}(S_i \,.\, S_i)$ *is* output-flat *if it is a unary choice, i.e., can be written* $\mathtt{p}!\ell(A) \,.\, S$ *or* $\mathtt{p}!!\ell(A) \,.\, S$.

An exception-aware runtime typing environment Θ *is* output-flat, *written* $\mathsf{flat}(\Theta)$, *if each output-directed choice type in* Θ *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* $S = \Sigma^\bullet_{i\in I}(\alpha_i \,.\, T_i)$ *if* $S' = \alpha_j \,.\, T_j$ *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:*

$$\Theta = s[\mathtt{p}_1]\langle\tilde{\mathtt{r}}_1\rangle{:}S^\bullet_1, \ldots, s[\mathtt{p}_l]\langle\tilde{\mathtt{r}}_l\rangle{:}S^\bullet_l, s[\mathtt{q}_1]\langle\tilde{\mathtt{s}}_1\rangle{:}T_1, \ldots, s[\mathtt{q}_m]\langle\tilde{\mathtt{s}}_m\rangle{:}T_m, s[\mathtt{t}_1] : \not\downarrow, \ldots, s[\mathtt{t}_n] : \not\downarrow$$

where each T *is not an output-directed choice, we say that* Θ' *is a* flattening *of* Θ *if:*

$$\Theta = s[\mathtt{p}_1]\langle\tilde{\mathtt{r}}_1\rangle{:}{S^\bullet}'_1, \ldots, s[\mathtt{p}_l]\langle\tilde{\mathtt{r}}_l\rangle{:}{S^\bullet}'_l, s[\mathtt{q}_1]\langle\tilde{\mathtt{s}}_1\rangle{:}T_1, \ldots, s[\mathtt{q}_m]\langle\tilde{\mathtt{s}}_m\rangle{:}T_m, s[\mathtt{t}_1] : \not\downarrow, \ldots, s[\mathtt{t}_n] : \not\downarrow$$

where each ${S^\bullet}'_i$ *is a flattening of* $S^\bullet_i$.

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

▶ **Definition 41** (Ready). *A configuration* $\mathcal{C}$ *is* ready, *written* $\mathsf{ready}(\mathcal{C})$, *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* $\cdot;\cdot \vdash_{\not\downarrow} \mathcal{G}[\mathcal{S}]$ *where* $\mathcal{S} = (\nu s : \Theta)\mathcal{C}$ *and* $\mathsf{ready}(\mathcal{C})$*, then there exists some* Θ' *such that* $\mathsf{flat}(\Theta')$ *where* Θ' *is a flattening of* Θ*, and* $\cdot;\cdot \vdash_{\not\downarrow} \mathcal{G}[(\nu s : \Theta')\mathcal{C}]$.

Proof. By Lemma 34, we have that there exist Ψ, Δ such that Δ does not contain session entries, and $\Psi; \Delta \vdash_{\not\downarrow} \mathcal{S}$.

By T-Session, $\Psi; \Delta, \Theta \vdash_{\not\downarrow} \mathcal{C}$.

The result follows by induction on the derivation of $\Psi; \Delta, \Theta \vdash_{\not\downarrow} \mathcal{C}$. In the case of T-Zap, the result follows immediately.

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

1. $\mathcal{C} = \langle a, M, s[\mathbf{p}]\langle \tilde{\mathbf{r}} \rangle, \kappa \rangle$
2. $\Psi; a : T, s[\mathbf{p}]\langle \tilde{\mathbf{r}} \rangle : S \vdash_{\not\downarrow} \mathcal{C}$
3. $\{T\}\ \Psi \mid S \triangleright M : A \triangleleft \mathsf{end}$

Since $\mathsf{ready}(\mathcal{C})$, 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, $\{T\}\ \Psi \mid S \triangleright N : B \triangleleft S'$ 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 $M = \mathbf{connect}\ \ell_j(V_j)\ \mathbf{to}\ W\ \mathbf{as}\ \mathbf{q}$, by T-Connect it must be the case that $S = \Sigma^{\bullet}_{i \in I}(\alpha_i \,.\, S'_i)$ where $\mathbf{q}!!\ell_j(A_j) \in \{\alpha_i\}_{i \in I}$ and $\Psi \vdash V_j : A_j$. Again by T-Connect and Lemma 24, we can show that $\{T\}\ \Psi \mid \mathbf{q}!!\ell_j(A_j) \,.\, S'_i \triangleright E[\mathbf{connect}\ \ell_j(V_j)\ \mathbf{to}\ W\ \mathbf{as}\ \mathbf{q}] : B \triangleleft S'$, where $\mathbf{q}!!\ell_j(A_j) \,.\, S'_i$ is a flattening of S, as required.

The same argument holds for send actions. ◀

▶ **Lemma 43** (Flattening preserves reducibility). *Suppose* $\mathsf{safe}(\Theta)$*,* $\mathsf{prog}(\Theta)$*, and* $\Theta \Longrightarrow$.
If $\hat{\Theta}$ *is a flattening of* Θ*, then* $\hat{\Theta} \Longrightarrow$.

Proof. Assume for the sake of contradiction that $\hat{\Theta} \not\Longrightarrow$.

For this to be the case, each ρ such that $\Theta \overset{\rho}{\Longrightarrow}$ must either be $s{:}\mathbf{p}, \mathbf{q}{::}\ell$ or $s{:}\mathbf{p} \twoheadrightarrow \mathbf{q}{::}\ell$, but instead, $\hat{\Theta} \xrightarrow{s:\mathbf{p}!\mathbf{q}'::\ell'(A)}$ for some $\mathbf{q}' \neq \mathbf{q}$. Note that ρ cannot be a disconnection action, since these are unaffected by flattening, and it cannot be the case that $\hat{\Theta} \xrightarrow{s:\mathbf{p} \twoheadrightarrow \mathbf{q}'::\ell' A}$, since this is a synchronisation action and could reduce.

By *(eventual communication)*, either $\Theta \overset{\vec{\rho}}{\Longrightarrow} \Theta' \xrightarrow{s:\mathbf{p}?\mathbf{q}'::\ell'(A)}$ or $\Theta \overset{\vec{\rho}}{\Longrightarrow} \Theta' \xrightarrow{\mathbf{q}':\mathbf{p}\not\downarrow}$.

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

It follows that either:

- $\hat{\Theta} \overset{\vec{\rho}}{\Longrightarrow} \Theta' \xrightarrow{s:\mathbf{p}!\mathbf{q}'::\ell'(A)} \xrightarrow{s:\mathbf{q}'?\mathbf{p}::\ell'(A)}$ meaning $\hat{\Theta} \overset{\vec{\rho}}{\Longrightarrow} \Theta' \xrightarrow{s:\mathbf{p},\mathbf{q}'::\ell'}$, a contradiction; or
- $\Theta \overset{\vec{\rho}}{\Longrightarrow} \Theta' \xrightarrow{\mathbf{q}':\mathbf{p}\not\downarrow}$, also a contradiction.

◀

C.6 Session progress

▶ **Lemma 44** (Readiness). *Suppose* $\cdot;\cdot \vdash_{\not\downarrow} \mathcal{C}$ *where* $\mathcal{C} = \mathcal{G}[\mathcal{S}]$*,* $\mathcal{C}$ *does not contain unmatched discovers,* $\mathcal{S}$ *is not a failed session,* $\mathcal{S} = (\nu s : \Theta)\mathcal{D}$*, and* $\mathsf{prog}(\Theta)$.
Either $\mathcal{C} \longrightarrow$ *or* $\mathsf{ready}(\mathcal{D})$.

Proof. For each $\langle a, M, s[\mathbf{p}]\langle \tilde{\mathbf{q}} \rangle, \kappa \rangle$, by Lemma 26, M can either reduce or is either a value, adaptation construct, or communication construct.

- If M can reduce, then the configuration can reduce by E-LiftM.
- If $M = E[\mathbf{raise}]$, 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.

- If M is a value, then by T-ConnectedActor, $\{T\}\ \Psi \mid \mathsf{end} \triangleright \mathbf{return}\ V{:}A \triangleleft \mathsf{end}$. As a consequence of $\mathsf{prog}(\Theta)$, a must be the only actor in the session and thus the configuration could reduce by E-Complete.
- It follows that all terms must be evaluating communication actions, satisfying $\mathsf{ready}(\mathcal{D})$.

◀

▶ **Lemma 45** (Exception-aware session progress). *If* $\cdot;\cdot \vdash_{\lightning} \mathcal{C}$ *where:*

1. $\mathcal{C} \equiv \mathcal{G}[\mathcal{S}]$,
2. $\mathcal{S} = (\nu s : \Theta)\mathcal{D}$,
3. $\mathit{flat}(\Theta)$,
4. $\mathit{ready}(\mathcal{C})$,
5. $\Theta \Longrightarrow$

then $\mathcal{C} \longrightarrow$.

Proof. Since $\Theta \Longrightarrow$, there exists some environment Θ' and label ρ such that $\Theta \overset{\rho}{\Longrightarrow} \Theta'$.

The proof is by induction on the derivation of $\Theta \overset{\rho}{\Longrightarrow} \Theta'$.

Case ET-Conn

$$\frac{\exists j \in I.S_j = \mathbf{q}!!\ell_j(A_j)\,.\,S'_j \qquad \mathsf{ty}(\mathbf{q}) = \Sigma_{k\in K}(\mathbf{p}??\ell_k(B_k)\,.\,T_k) \qquad j \in K \qquad A_j = B_j}{s[\mathbf{p}]\langle\tilde{\mathbf{r}}\rangle{:}\Sigma_{i\in I}(S_i) \xrightarrow{s:\mathbf{p}\to\mathbf{q}::\ell_j} s[\mathbf{p}]\langle\tilde{\mathbf{r}},\mathbf{q}\rangle{:}S'_j, s[\mathbf{q}]\langle\mathbf{p}\rangle{:}T_j}$$

Since $\mathsf{flat}(\Theta)$, we have that $s[\mathbf{p}]\langle\tilde{\mathbf{r}}\rangle{:}\mathbf{q}!!\ell_j(A_j)\,.\,S'_j$. Thus, $\mathcal{C}$ must contain an actor evaluating $E[\mathbf{connect}\ \ell_j(A_j)\ \mathbf{to}\ V\ \mathbf{as}\ b]$ 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 E-Conn or E-ConnFail.

Case ET-Comm

$$\text{ET-Comm}\ \frac{\Theta_1 \xrightarrow{s:\mathbf{p}!\mathbf{q}::\ell(A)} \Theta'_1 \qquad \Theta_2 \xrightarrow{s:\mathbf{q}?\mathbf{p}::\ell(A)} \Theta'_2}{\Theta_1, \Theta_2 \xrightarrow{s:\mathbf{p},\mathbf{q}::\ell} \Theta'_1, \Theta'_2}$$

By ET-Act, Θ_1 must contain $s[\mathbf{p}]\langle\tilde{\mathbf{r}}\rangle{:}S$; since $\mathsf{flat}(\Theta)$, $S = \mathbf{q}!\ell(A)$.
Also by ET-Act, Θ_2 must contain $s[\mathbf{p}]\langle\tilde{\mathbf{r}}\rangle{:}\Sigma_{i\in I}(\beta_i\,.\,T'_i)$ with $\mathbf{p}?\ell(A) \in \{\beta_i\}$.
Thus by typing, $\mathcal{C}$ must contain an actor $\langle a, E[\mathbf{send}\ \ell(V)\ \mathbf{to}\ \mathbf{q}], s[\mathbf{p}]\langle\tilde{\mathbf{r}}\rangle, \kappa_a\rangle$, and another actor $\langle b, E'[\mathbf{receive\ from}\ \mathbf{p}\ \{\ell_i(x_i) \mapsto M_i\}], s[\mathbf{q}]\langle\tilde{\mathbf{s}}\rangle, \kappa_b\rangle$, which would reduce by E-Comm.

Case ET-CommFail

$$\text{ET-CommFail}\ \frac{\exists j \in I.S_j = \mathbf{p}\dagger\ell_j(A_j)\,.\,S'_j}{s[\mathbf{p}] : \lightning, s[\mathbf{q}]\langle\tilde{\mathbf{r}}\rangle{:}\Sigma_{i\in I}(\alpha_i\,.\,S'_i) \xrightarrow{s:\mathbf{p}\lightning\mathbf{q}} s[\mathbf{p}] : \lightning, s[\mathbf{q}] : \lightning}$$

We prove the case where $\dagger = !$. Since $\mathsf{flat}(\Theta)$, by typing we have that $\Sigma_{i\in I}(\alpha_i\,.\,S'_i) = \ell!A(\mathbf{q})\,.\,S'$ and $\mathcal{C}$ must contain an actor $\langle a, E[\mathbf{send}\ \ell(V)\ \mathbf{to}\ \mathbf{q}], s[\mathbf{p}]\langle\tilde{\mathbf{r}}\rangle, \kappa_a\rangle$ and a zapper thread $\lightning\, s[\mathbf{q}]$, 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. ◀

▶ **Lemma 46.** *If* $\mathit{prog}(\Theta)$ *and neither* $\mathit{end}(\Theta)$ *nor* $\mathit{failed}(\Theta)$*, then* $\Theta \Longrightarrow$.

Proof. By contradiction. Assume that $\mathsf{prog}(\Theta)$ and neither $\mathsf{end}(\Theta)$ nor $\mathsf{failed}(\Theta)$.

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

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

Otherwise, by the definition of $\mathsf{prog}(\Theta)$ for each $s[\mathsf{p}]\langle\tilde{\mathsf{q}}\rangle{:}S$ there must exist some $\overrightarrow{\rho}$ such that $\Theta \xrightarrow{\overrightarrow{\rho}} {}^*\Theta' \xRightarrow{\rho}$ where $\mathsf{p} \in \mathsf{subj}(\rho)$: a contradiction. ◀

▶ **Lemma 47.** *If $\cdot;\cdot \vdash_{\lightning} \mathcal{S}$ where $\mathcal{S} = (\nu s : \Theta)\mathcal{C}$ and $\mathsf{ready}(\mathcal{C})$, then Θ is not final.*

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

▶ **Lemma 18** (Session Progress). *If $\cdot;\cdot \vdash \mathcal{C}$ where $\mathcal{C}$ does not contain an unmatched discover, $\mathcal{C} \equiv \mathcal{G}[\mathcal{S}]$ and $\mathcal{S} = (\nu s : \Delta)\mathcal{D}$ with $\mathsf{prog}(\Delta)$, and $\mathcal{S}$ is not a failed session, then $\mathcal{C} \longrightarrow$.*

Proof.
- By Lemma 27, $\exists.\Psi, \Delta'$ such that $\Psi;\Delta' \vdash \mathcal{S}$.
- By definition, $\mathcal{S} = (\nu s : \Delta)\mathcal{D}$, where $\mathcal{D} = \langle a_1, M_1, s[\mathsf{p}_1]\langle\tilde{\mathsf{q}}_1\rangle, \kappa_1\rangle \parallel \cdots \parallel \langle a_m, M_m, s[\mathsf{p}_m]\langle\tilde{\mathsf{q}}_m\rangle, \kappa_m\rangle \parallel \lightning\, s[\mathsf{p}_{m+1}] \parallel \cdots \parallel \lightning\, s[\mathsf{p}_n]$.
- By T-Session, $\mathsf{safe}(\Delta)$.
- By definition, the zapped roles of $\mathcal{C}$ are $\mathsf{p}_{m+1}, \ldots, \mathsf{p}_n$. Let us denote this set as $\tilde{\mathsf{p}}$.
- Let $\Theta = \mathsf{zap}(\Delta, \tilde{\mathsf{p}})$.
- By Lemma 32, $\Psi;\Delta', \Theta \vdash_{\lightning} \mathcal{D}$ and so by Corollary 33, $\cdot;\cdot \vdash_{\lightning} \mathcal{G}[\mathcal{S}]$.
- By Lemma 36, $\mathsf{safe}(\Theta)$ and $\mathsf{prog}(\Theta)$.
- By Lemma 44, either $\mathcal{C} \longrightarrow$ or $\mathsf{ready}(\mathcal{D})$. As $\mathcal{C} \longrightarrow$ satisfies the theorem statement, we proceed assuming $\mathsf{ready}(\mathcal{D})$.
- By Lemma 42, there exists some $\hat{\Theta}$ such that $\hat{\Theta}$ is a flattening of Θ and $\cdot;\cdot \vdash_{\lightning} \mathcal{G}[(\nu s : \hat{\Theta})\mathcal{D}]$.
- By Lemma 47, $\hat{\Theta}$ is not final.
- By Lemma 43, $\hat{\Theta} \Longrightarrow$.
- Thus, by Lemma 45, $\mathcal{C} \longrightarrow$ as required.

◀

C.7 Progress

▶ **Theorem 19** (Progress). *Suppose $\cdot;\cdot \vdash \mathcal{C}$ where $\mathcal{C}$ is in canonical form. If $\mathcal{C}$ does not contain an unmatched discover, either $\exists\mathcal{D}$ such that $\mathcal{C} \longrightarrow \mathcal{D}$, or $\mathcal{C} \equiv \mathbf{0}$, or $\mathcal{C} \equiv (\nu b_1 \cdots \nu b_n)(\langle b_1, N_1, \bot, \kappa_1\rangle \parallel \cdots \parallel \langle b_n, N_n, \bot, \kappa_n\rangle)$ where each b_i is terminated or accepting.*

Proof.
- First, eliminate all failed sessions via the equivalence $(\nu s)(\lightning\, s\mathsf{p}_1 \parallel \cdots \parallel \lightning\, s[\mathsf{p}_n]) \parallel \mathcal{C} \equiv \mathcal{C}$. Heneceforth assume no sessions are failed.
- By repeated application of Lemma 18, either $\mathcal{C}$ can reduce or it contains no sessions.
- Thus we only need to consider disconnected actors. For each disconnected actor $\langle a, M, \bot, \kappa\rangle$, by Lemma 26, either:
  1. $M = \mathbf{return}\, V$. If $\kappa = N$, then the configuration can reduce by E-Loop. Otherwise, if $\kappa = \mathbf{stop}$, then the actor is terminated, as required.
  2. $M = E[\mathbf{raise}]$. If E contains an exception handler, the configuration can reduce by E-TryRaise. If not, E is pure: if $\kappa = N$, then the configuration can reduce by E-FailLoop. Otherwise, if $\kappa = \mathbf{stop}$, then the actor is terminated, as required.
  3. $M = E[N]$ where N is a communication or concurrency action, which, following the logic in Lemma 18, can either reduce by a configuration reduction rule or is a communication action. Since:
     - by T-UnconnectedActor each disconnected actor must either have type end or be following its statically-defined session type
     - by well-formedness, each statically-defined session type must correspond to a role in a protocol
     - by well-formedness, each protocol must have a unique initiator

- 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.

◀