Session Types without Tiers

SIMON FOWLER, University of Edinburgh
SAM LINDLEY, University of Edinburgh
J. GARRETT MORRIS, University of Kansas
SÁRA DECOVA, University of Edinburgh

Session types statically guarantee that concurrent communication complies with a protocol. However, most accounts of session typing do not account for failure, which means they are of limited use in real applications—especially distributed applications—where failure is pervasive.

We present the first formal integration of asynchronous session types with exception handling in a functional programming language. We define a core calculus which satisfies preservation and progress properties, is deadlock free, confluent, and terminating.

We provide the first implementation of session types with exception handling for a fully-fledged functional programming language, by extending the Links web programming language; our implementation draws on existing work on algebraic effects and effect handlers. We illustrate our approach through a running example of two-factor authentication, and a larger example of a session-based chat application where communication occurs over session-typed channels and disconnections are handled gracefully.

1 INTRODUCTION

With the growth of the internet and mobile devices, as well as the failure of Moore's law, concurrency and distribution have become central to many applications. Writing correct concurrent and distributed code requires effective tools for reasoning about communication protocols. While data types do provide an effective tool for reasoning about the shape of data communicated, protocols also require us to reason about the order in which messages are transmitted.

Session types [Honda 1993; Honda et al. 1998] are types for protocols. They describe both the shape and order of messages. If a program type-checks according to its session type, then it is statically guaranteed to comply with the corresponding protocol.

Alas, most accounts of session types do not handle failure, which means they are of limited use in distributed settings where failure is pervasive. Inspired by work of Mostrous and Vasconcelos [2014], we present the first account of asynchronous session types in a functional programming language, which smoothly handles both distribution and failure. We present both a core calculus enjoying strong metatheoretical correctness properties and a practical implementation as an extension of the Links web programming language [Cooper et al. 2007].

1.1 Session Types

We illustrate session types with a basic example of two-factor authentication. A user inputs their credentials. If the login attempt is from a known device, then they are authenticated and may proceed to perform privileged actions. If the login attempt is from an unrecognised device, then the user is sent a challenge code. They enter the challenge code into a hardware key which yields a response code. If the user responds with the correct response code, then they are authenticated.

A session type specifies the communication behaviour of one endpoint of a communication channel participating in a dialogue (or *session*) with the other endpoint of the channel. Fig. 1 shows the session types of two channel endpoints connecting a client and a server. Fig. 1a shows the session type for the server which first receives (?) a pair of a username and password from the

Authors' addresses: Simon Fowler, University of Edinburgh, simon.fowler@ed.ac.uk; Sam Lindley, University of Edinburgh, sam.lindley@ed.ac.uk; J. Garrett Morris, University of Kansas, garrett@ittc.ku.edu; Sára Decova, University of Edinburgh, sara.decova@gmail.com.

```
TwoFactorServer ≜
  ?(Username, Password).⊕{
    Authenticated : ServerBody,
    Challenge : !ChallengeKey.?Response.
      ⊕{ Authenticated : ServerBody,
         AccessDenied : End},
    AccessDenied : End}
```

(a) Server Session Type

```
TwoFactorClient ≜
  !(Username, Password).&{
    Authenticated : ClientBody,
    Challenge : ?ChallengeKey.!Response.
      &{ Authenticated : ClientBody,
         AccessDenied : End},
    AccessDenied : End}
```

(b) Client Session Type

Fig. 1. Two-factor Authentication Session Types

client. Next, the server selects ($\oplus$) whether to authenticate the client, issue a challenge, or reject the credentials. If the server decides to issue a challenge, then it sends (!) the challenge string, awaits the response, and either authenticates or rejects the client. The ServerBody type abstracts over the remainder of the interactions, for example making a deposit or withdrawal.

The client implements the *dual* session type, shown in Fig. 1b. Whenever the server receives a value, the client sends a value, and vice versa. Whenever the server makes a selection, the client offers a choice (&), and vice versa. This *duality* between client and server ensures that each communication is matched by the other party. We denote duality with an overbar; thus TwoFactorClient = $\overline{\text{TwoFactorServer}}$ and TwoFactorServer = $\overline{\text{TwoFactorClient}}$.

Implementing Two-factor Authentication. Let us suppose we have constructs for sending and receiving along, and for closing, an endpoint.

send $M\ N : S$	where M has type A, and N is an endpoint with session type $!A.S$
receive $M : (A \times S)$	where M is an endpoint with session type $?A.S$
close $M : \mathbf{1}$	where M is an endpoint with session type End

Let us also suppose we have constructs for selecting and offering a choice:

select $\ell_j\ M : S_j$	where M is an endpoint with session type $\oplus\{\ell_i : S_i\}_{i \in I}$, and $j \in I$
offer $M\ \{\ell_i(x_i) \mapsto N_i\}_{i \in I} : A$	where M is an endpoint with session type $\&\{\ell_i \mapsto S_i\}_{i \in I}$, each x_i binds an endpoint with session type S_i, and each N_i has type A

We can now write a client implementation:

```
twoFactorClient : (Username × Password × TwoFactorClient) ⊸ 1
twoFactorClient(username, password, s) ≜
  let s = send (username, password) s in
  offer s {
    Authenticated(s) ↦ clientBody(s)
    Challenge(s) ↦
      let (key, s) = receive (s) in
      let s = send generateResponse(key) s in
      offer s {
        Authenticated(s) ↦ clientBody(s)
        AccessDenied(s) ↦ close s; loginFailed
      }
    AccessDenied(s) ↦ close s; loginFailed
  }
```

The twoFactorClient function takes a username, password, and an endpoint s of type TwoFactorClient as its arguments. It sends the username and password along the endpoint, before offering three branches depending on whether the server authenticates the user, sends a two-factor challenge, or rejects the authentication attempt. Note that the rejection of an authentication attempt is part of the protocol, and *not* exceptional behaviour. In the case that the server authenticates the user, then the program progresses to the main application (denoted here by clientBody(s)). If the server sends a challenge, the client receives the challenge key, and sends the response, calculated by generateResponse. It then offers two branches based on whether the challenge response was successful. The server implementation is similarly straightforward:

```
twoFactorServer : TwoFactorServer ⊸ 1
twoFactorServer(s) ≜
  let ((username, password), s) = receive s in
  if checkDetails(username, password) then
    let s = select Authenticated s in serverBody(s)
  else
    let s = select AccessDenied s in close s
```

The twoFactorServer function takes an endpoint of type TwoFactorServer, receives a username and password, and which are checked using the checkDetails function. If the check passes, then the server authenticates the client and proceeds to the application body (denoted here by serverBody(s)); if not, then the server notifies the client by selecting the AccessDenied branch. Note that this particular server implementation opts to never send a challenge request.

To successfully implement session types, one necessarily needs a substructural type system. We discuss three options: linear types, affine types, and linear types with explicit cancellation.

1.2 Linear Types

Simply providing constructs for sending and receiving values, and for selecting and offering choices, is not quite enough to safely implement session types. Consider the following client:

```
wrongClient : TwoFactorClient ⊸ 1
wrongClient(s) ≜
  let t = send ("Alice", "hunter2") s in
  let t = send ("Bob", "letmein") s in ...
```

Reuse of s allows a (username, password) pair to be sent along the same endpoint twice, violating the fundamental property of *session fidelity*, which states that in a well-typed program the communication taking place over an endpoint matches its session type. In order to maintain session fidelity and ensure that all communication actions in a session type occur, session type systems typically require that endpoints are used *linearly*—each endpoint must be used exactly once.

Exceptions. In real programs, linear session types are unrealistic. Thus far, we have assumed that checkDetails always succeeds, which may be plausible if the server is checking against an in-memory store, but certainly not if it is contacting a remote database. One approach would be to simply have checkDetails return false should the request fail, but this approach loses information. Instead, let us suppose we have a basic try – catch exception handling construct.

As a first attempt, we might try to write:

```
exnServer1 : TwoFactorClient ⊸ 1
exnServer1(s) ≜
  let ((username, password), s) = receive (s) in
  try
    if checkDetails(username, password) then
      let s = select Authenticated s in serverBody(s)
    else
      let s = select AccessDenied s in close s
  catch log("Database Error")
```

However, the above code does not type-check and is unsafe. Linear endpoint s is not used in the **catch** block and yet it may still be open if an exception is raised by checkDetails.

As a second attempt, we may decide to localise exception handling to the call to checkDetails. We introduce checkDetailsOpt, which returns Some(*result*) if the call is successful and None if not.

```
checkDetailsOpt :
  (Username × Password) ⊸ Option(Bool)
checkDetailsOpt(username, password) ≜
  try Some(checkDetails(username, password))
  catch None
```

```
exnServer2 : TwoFactorServer ⊸ 1
exnServer2(s) ≜
  let ((username, password), s) = receive (s) in
  case checkDetailsOpt(username, password) of
    Some(res) ↦
      if res then
        let s = select Authenticated s in
        serverBody(s)
      else
        let s = select AccessDenied s in
        close s
    None ↦ log("Database Error")
```

Still the code is unsafe as it does not use s in the None branch of the case-split. However, we do now have more precise information about the type of s, since it is unused in the **try** block. One

solution could be to adapt the protocol by adding an **InternalError** branch:

$$
\begin{aligned}
&\mathsf{TwoFactorServerExn} \triangleq \\
&\quad ?(\mathsf{Username}, \mathsf{Password}).\oplus\{ \\
&\qquad \mathsf{Authenticated} : \mathsf{ServerBody}, \\
&\qquad \mathsf{Challenge} : !\mathsf{ChallengeKey}.?\mathsf{Response}. \\
&\qquad\quad \oplus\{\mathsf{Authenticated} : \mathsf{ServerBody}, \mathsf{AccessDenied} : \mathsf{End}\}, \\
&\qquad \mathsf{AccessDenied} : \mathsf{End}, \\
&\qquad \mathsf{InternalError} : \mathsf{End}\}
\end{aligned}
$$

With this modification in place, we could use **select** InternalError *s* in the None branch and write a type-correct program. But this approach is still rather unsatisfactory as it entails cluttering both the protocol and the implementation with failure points, even though they may occur rarely.

Disconnection. The problem of failure is compounded by the possibility of disconnection. On a single machine it may be plausible to assume that communication always succeeds. In a distributed setting this assumption is unrealistic as parties may disconnect without warning. The problem is particularly acute in web applications as a client may close the browser at any point. In order to adequately handle failure we must incorporate some mechanism for detecting disconnection.

1.3 Affine Types

We began by assuming linear types—each endpoint must be used *exactly* once. One might consider relaxing linear types to *affine types*—each endpoint must be used *at most* once. Statically checked affine types underlie the existing Rust implementation of session types [Jespersen et al. 2015] and dynamically checked affine types underlie the FuSe [Padovani 2017] and `lchannels` [Scalas and Yoshida 2016] session type implementations in OCaml and Scala respectively.

However, affine types present two quandaries, both arising from endpoints being silently discarded. First, a developer receives no feedback if they *accidentally* forget to finish the implementation of a protocol. Second, if an exception is raised then the peer may be left waiting forever if an open endpoint appears in the evaluation context of the raised exception.

1.4 Linear Types with Explicit Cancellation

Mostrous and Vasconcelos [2014] address the difficulties outlined above through an *explicit* discard (or *cancellation*) operator. (They characterise their sessions as *affine*, but it is important not to confuse their system with affine type systems, as in §1.3, which allow variables to be discarded *implicitly*.) Their approach boils down to three key principles: endpoints can be explicitly discarded; an *exception* is thrown if a communication cannot succeed because a peer endpoint has been cancelled; and endpoint cancellations are *propagated* when endpoints become inaccessible due to an exception being thrown. They introduce a process calculus including the term $a\lightning$ ("cancel a"), which indicates that endpoint a may no longer be used to perform communications. Mostrous and Vasconcelos provide an exception handling construct which attempts a communication action, running an exception handler if the communication action fails, and show that explicit cancellation is *well-behaved*: their calculus is sound, satisfies *global progress*—no session gets stuck even in the presence of cancellation—and confluent.

Explicit cancellation neatly addresses the problem of failure while ruling out the problem of accidentally incomplete implementations and providing a mechanism for notifying peers when an exception is raised. In this paper we take advantage of explicit cancellation to formalise and implement asynchronous session types with failure handling in a distributed functional programming

language. Doing so is not simply a routine adaptation of the ideas of Mostrous and Vasconcelos for the following reasons:

- They present a *process calculus*, but we work in a *functional programming language*.
- Communication in their system is *synchronous*, depending on a rendezvous between sender and receiver. We require *asynchronous* communication, which is more amenable to implementation in a distributed setting.
- Their exception handling construct is over a *single* communication action, and does not allow nested exception handling. This design is difficult to reconcile with a functional language, as it is inherently non-compositional.

We define a core concurrent λ-calculus, *Exceptional GV*, with asynchronous session-typed communication and exception handling. As with the calculus of Mostrous and Vasconcelos, an exception is raised when a communication action fails. But our compositional exception handling construct can be arbitrarily nested, and allows exception handling over multiple communication actions.

Using EGV, we may implement the two factor authentication server as follows:

```
exnServer3 : TwoFactorServer ⊸ 1
exnServer3(s) ≜
  let ((username, password), s) = receive (s) in
  try checkDetails(username, password) as res in
    if res then let s = select Authenticated s in serverBody(s)
    else
      let s = select AccessDenied s in close s
  otherwise
    cancel (s); log("Database Error")
```

Following Benton and Kennedy [2001], exception handlers **try** L **as** x **in** M **otherwise** N take an explicit success continuation M as well as the usual failure continuation N. If checkDetails fails with an exception, then we safely discard s using **cancel**, which takes an endpoint and returns the unit value. Disconnection is handled by cancelling all endpoints associated with a client. If a peer tries to read along a cancelled endpoint, then an exception is thrown.

We implement the constructs described by EGV as an extension to Links [Cooper et al. 2007], a functional programming language for the web. Our implementation is based on a minimal translation to effect handlers [Plotkin and Pretnar 2013].

1.5 Contributions

This paper makes five main contributions:

(1) *Exceptional GV* (§2), a core linear lambda calculus extended with asynchronous session-typed channels and exception handling. We prove (§3) that the calculus enjoys preservation, progress, a strong form of confluence called the *diamond property*, and termination.
(2) Extensions to EGV supporting exception payloads, unrestricted types, and access points (§4).
(3) The design and implementation of an extension of the Links web programming language to support tierless web applications which can communicate using session-typed channels (§5).
(4) Client and server backends for Links implementing session typing with exception handling (§5.5), drawing on connections with effect handlers [Plotkin and Pretnar 2013].
(5) Example applications using the infrastructure (§6). In addition to our two-factor authentication workflow we outline the implementation of a chat server.

Links is open-source and freely-available. The website can be found at http://www.links-lang.org and the source at http://www.github.com/links-lang/links. Users of the `opam` tool can install Links

```
try
  let s =
    fork (λt.cancel t) in
  let (res, s) = receive s in
  close s; res
as res in
  print ("Result: " + res)
otherwise print "Error!"
```

(a) Cancellation and Exceptions

```
let s =
  fork (λt.
    let (res, t) = receive t in
    close t; res) in
let u = fork (λv.cancel v) in
let u = send s u in
close u
```

(b) Delegation

```
let f = (λx.send x s) in
raise;
f(5)
```

(c) Closures

Fig. 2. Failure Examples

by invoking `opam install links`. Examples described in the paper, and the Links source code, are included in the supplementary material.

The rest of the paper is structured as follows: §2 presents Exceptional GV and §3 its metatheory; §4 discusses extension to Exceptional GV; §5 describes the implementation; §6 presents a chat application written in Links; §7 discusses related work; and §8 concludes.

2 EXCEPTIONAL GV

In this section, we introduce Exceptional GV (henceforth EGV). GV is a core session-typed linear λ-calculus with a tight correspondence with classical linear logic [Lindley and Morris 2015; Wadler 2014]. EGV is an asynchronous variant of GV with support for failure handling.

2.1 Integrating Sessions with Exceptions, by Example

Safely integrating session types with failure handling into a higher-order functional language requires some care. In this section, we illustrate three important cases we must consider (Fig. 2): cancellation and exceptions, delegation, and closures.

In order to initiate a session, we adopt the **fork** primitive of Lindley and Morris [2015]. Given a term M of type $S \multimap \mathbf{1}$, the term **fork** M of type $\overline{S}$ creates a fresh channel with endpoints a of type S and b of type $\overline{S}$, forks a child thread that executes $M\ a$, and returns endpoint b.

Cancellation and Exceptions. The code in Fig. 2a illustrates the basic case of forking off a thread and immediately cancelling the endpoint of the child thread. Variables s and t are bound to peer endpoints. The child thread immediately discards t using **cancel**. Meanwhile, the parent thread attempts to receive from s, but the message it is waiting for can never arrive so an exception is raised, and the **otherwise** clause is evaluated.

Delegation. A key concept in the π-calculus is *mobility* of names. In session calculi, sending an endpoint is known as *session delegation*. The code in Fig. 2b begins by forking a thread and returning endpoint s. The child thread is passed endpoint t on which it blocks receiving. Next, the parent thread forks a second child thread, obtaining the endpoint u. The second child thread is passed endpoint v, which it immediately discards using **cancel**. Now the parent thread sends endpoint s along u. At this point endpoint s will never be received since the peer endpoint v of u has been cancelled. In turn, this renders s irretrievable. Consequently, an exception is thrown in the first child thread, as it can now never receive a value.

Types	$A, B, C ::= \mathbf{1} \mid A \multimap B \mid A + B \mid A \times B \mid S$
Session Types	$S ::= !A.S \mid ?A.S \mid \mathsf{End}$
Variables	x, y, z
Terms	$L, M, N ::= x \mid \lambda x.M \mid M\ N$
	$\mid () \mid \mathbf{let}\ () = M\ \mathbf{in}\ N$
	$\mid (M, N) \mid \mathbf{let}\ (x, y) = M\ \mathbf{in}\ N$
	$\mid \mathbf{inl}\ M \mid \mathbf{inr}\ M \mid \mathbf{case}\ L\ \mathbf{of}\ \{\mathbf{inl}\ x \mapsto M; \mathbf{inr}\ y \mapsto N\}$
	$\mid \mathbf{fork}\ M \mid \mathbf{send}\ M\ N \mid \mathbf{receive}\ M \mid \mathbf{close}\ M$
	$\mid \mathbf{cancel}\ M \mid \mathbf{raise} \mid \mathbf{try}\ L\ \mathbf{as}\ x\ \mathbf{in}\ M\ \mathbf{otherwise}\ N$
Type Environments	$\Gamma ::= \cdot \mid \Gamma, x : A$

Fig. 3. Syntax

Closures. It is crucial that cancellation plays nicely with closures. The code in Fig. 2c constructs a function f which takes an argument x which is sent along s. Next, an exception is explicitly raised. As s appears in the closure bound to f, which appears in the continuation and so has been discarded, s must now be cancelled.

2.2 Syntax and Typing Rules for Terms

Fig. 3 gives the syntax of EGV. Types include unit ($\mathbf{1}$), linear functions ($A \multimap B$), linear sums ($A + B$), linear tensor products ($A \times B$), and session types (S).

Terms include variables (x) and the usual introduction and elimination forms for linear functions, unit, products, and sums. We write $M; N$ as syntactic sugar for $\mathbf{let}\ () = M\ \mathbf{in}\ N$ and $\mathbf{let}\ x = M\ \mathbf{in}\ N$ for $(\lambda x.N)\ M$. The standard session typing primitives [Lindley and Morris 2015] are as follows: $\mathbf{fork}\ M$ creates a fresh channel with endpoints a of type S and b of type $\overline{S}$, forks a child thread that executes $M\ a$, and returns endpoint b; $\mathbf{send}\ M\ N$ sends M along endpoint N; $\mathbf{receive}\ M$ receives along endpoint M; and $\mathbf{close}\ M$ closes an endpoint when a session is complete.

We introduce three new term constructs to support session typing with failure handling: $\mathbf{cancel} M$ explicitly discards session endpoint M; $\mathbf{raise}$ raises an exception; and $\mathbf{try}\, L\, \mathbf{as}\, x\, \mathbf{in}\, M\, \mathbf{otherwise}\, N$ is used for exception handling in the style of Benton and Kennedy's $\mathbf{try}{-}\mathbf{in}{-}\mathbf{unless}$ construct [Benton and Kennedy 2001]. The ability to distinguish between a possibly-failing term and an explicit success continuation is convenient both for typing and for implementation, as we will see in §5.5.

Though our implementation supports **select** and **offer** directly, and we use them in examples, we omit them from the core calculus (following Lindley and Morris [2015, 2017]) as they can be encoded using sums and delegation [Dardha et al. 2017; Kobayashi 2003].

Type environments are standard.

Typing. Fig. 4 gives the typing rules for EGV. As usual, linearity is enforced by splitting type environments when typing subterms and by ensuring that leaf rules T-Var, T-Unit, and T-Raise take an empty type environment. The bulk of the rules are standard for a linear λ-calculus. Session types are related by *duality*. The T-Fork rule forks a thread connected by dual endpoints of a channel. The rules T-Send, T-Recv, and T-Close capture session-typed communication.

As exceptions do not return values, the rule T-Raise allows an exception to be given any type A. The rule T-Try is similar to that of Mostrous and Vasconcelos [2014]:

$$\frac{\Gamma, a : S \vdash \rho \qquad \Gamma \vdash P \qquad \mathsf{subject}(\rho) = a}{\Gamma, a : S \vdash \mathsf{do}\ \rho\ \mathsf{catch}\ P}$$

Term Typing $\boxed{\Gamma \vdash M : A}$

$$\text{T-Var}\ \frac{}{x : A \vdash x : A} \qquad \text{T-Abs}\ \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \multimap B} \qquad \text{T-App}\ \frac{\Gamma_1 \vdash M : A \multimap B \qquad \Gamma_2 \vdash N : A}{\Gamma_1, \Gamma_2 \vdash M\ N : B}$$

$$\text{T-Unit}\ \frac{}{\cdot \vdash () : \mathbf{1}} \qquad \text{T-LetUnit}\ \frac{\Gamma_1 \vdash M : \mathbf{1} \qquad \Gamma_2 \vdash N : A}{\Gamma_1, \Gamma_2 \vdash \mathbf{let}\ () = M\ \mathbf{in}\ N : A} \qquad \text{T-Pair}\ \frac{\Gamma_1 \vdash M : A \qquad \Gamma_2 \vdash N : B}{\Gamma_1, \Gamma_2 \vdash (M, N) : A \times B}$$

$$\text{T-LetPair}\ \frac{\Gamma_1 \vdash M : A \times A' \qquad \Gamma_2, x : A, y : A' \vdash N : B}{\Gamma_1, \Gamma_2 \vdash \mathbf{let}\ (x, y) = M\ \mathbf{in}\ N : B}$$

$$\text{T-Inl}\ \frac{\Gamma \vdash M : A}{\Gamma \vdash \mathbf{inl}\ M : A + B} \qquad \text{T-Inr}\ \frac{\Gamma \vdash M : B}{\Gamma \vdash \mathbf{inr}\ M : A + B} \qquad \text{T-Case}\ \frac{\Gamma_1 \vdash L : A + B \qquad \Gamma_2, x : A \vdash M : C \qquad \Gamma_2, x : B \vdash N : C}{\Gamma_1, \Gamma_2 \vdash \mathbf{case}\ L\ \mathbf{of}\ \{\mathbf{inl}\ x \mapsto M; \mathbf{inr}\ x \mapsto N\} : C}$$

$$\text{T-Fork}\ \frac{\Gamma \vdash M : S \multimap \mathbf{1}}{\Gamma \vdash \mathbf{fork}\ M : \overline{S}} \qquad \text{T-Send}\ \frac{\Gamma_1 \vdash M : A \qquad \Gamma_2 \vdash N : !A.S}{\Gamma_1, \Gamma_2 \vdash \mathbf{send}\ M\ N : S} \qquad \text{T-Recv}\ \frac{\Gamma \vdash M : ?A.S}{\Gamma \vdash \mathbf{receive}\ M : (A \times S)} \qquad \text{T-Close}\ \frac{\Gamma \vdash M : \mathsf{End}}{\Gamma \vdash \mathbf{close}\ M : \mathbf{1}}$$

$$\text{T-Cancel}\ \frac{\Gamma \vdash M : S}{\Gamma \vdash \mathbf{cancel}\ M : \mathbf{1}} \qquad \text{T-Try}\ \frac{\Gamma_1 \vdash L : A \qquad \Gamma_2, x : A \vdash M : B \qquad \Gamma_2 \vdash N : B}{\Gamma_1, \Gamma_2 \vdash \mathbf{try}\ L\ \mathbf{as}\ x\ \mathbf{in}\ M\ \mathbf{otherwise}\ N : B} \qquad \text{T-Raise}\ \frac{}{\cdot \vdash \mathbf{raise} : A}$$

Duality $\boxed{\overline{S}}$

$$\overline{!A.S} = ?A.\overline{S} \qquad \overline{?A.S} = !A.\overline{S} \qquad \overline{\mathsf{End}} = \mathsf{End}$$

Fig. 4. Term Typing and Duality

They allow exception handling over a *single* communication action ρ over name a. If the communication action fails, then control moves to the exception handling process P which is typeable *without* a. In order to allow exception handling over multiple communication actions we take a different approach. Embracing explicit success continuations as advocated by Benton and Kennedy [2001], instead of *subtracting* a linear name from a context upon failure, we *bind* a result in M if L evaluates successfully.

The T-Cancel rule explicitly discards an endpoint. Naïvely implemented, cancellation violates progress: a thread could discard an endpoint, leaving a peer waiting forever. However, our integration of cancellation and exceptions ensures that we retain strong progress guarantees (§3).

2.3 Operational Semantics

We now give a small-step operational semantics for EGV.

Runtime Syntax. Fig. 5 shows the runtime syntax of EGV. We introduce the type $S^\sharp$ as the type of a channel which can be split into two endpoints of types S and $\overline{S}$. We extend the syntax of terms to include names ranged over by a, b, c. Depending on context, a name a is variously used to identify a channel of type $S^\sharp$ and each of its endpoints of type S and $\overline{S}$. Values are standard. The semantics makes use of *configurations*, which are similar to processes in the π-calculus: $(\nu a)\mathcal{C}$ binds name a in configuration $\mathcal{C}$; and $\mathcal{C} \parallel \mathcal{D}$ is the parallel composition of configurations $\mathcal{C}$ and $\mathcal{D}$.

Program threads take the form ϕM, where ϕ is a thread flag identifying whether the term is the *main thread* ($\bullet$), meaning that it may return a result, or a *child thread* ($\circ$), which may not. A configuration may have at most one main thread.

Types	$A ::= \cdots \mid S^\sharp$
Names	a, b, c
Terms	$M ::= \cdots \mid a$
Values	$U, V, W ::= x \mid a \mid \lambda x.M \mid () \mid (V, W) \mid \mathbf{inl}\, V \mid \mathbf{inr}\, V$
Configurations	$\mathcal{C}, \mathcal{D}, \mathcal{E} ::= (\nu a)\mathcal{C} \mid \mathcal{C} \parallel \mathcal{D} \mid \phi M \mid \mathbf{halt} \mid \lightning\, a \mid a(\overrightarrow{V}) \rightsquigarrow b(\overrightarrow{W})$
Thread Flags	$\phi ::= \bullet \mid \circ$
Top-level threads	$\mathcal{T} ::= \bullet M \mid \mathbf{halt}$
Auxiliary threads	$\mathcal{A} ::= \circ M \mid \lightning\, a \mid a(\overrightarrow{V}) \rightsquigarrow b(\overrightarrow{W})$
Type Environments	$\Gamma ::= \cdots \mid \Gamma, a : S$
Runtime Type Environments	$\Delta ::= \cdot \mid \Delta, a : S \mid \Delta, a : S^\sharp$
Evaluation Contexts	$E ::= [\,] \mid E\, M \mid V\, E$
	$\mid \mathbf{let}\, () = E\, \mathbf{in}\, M \mid \mathbf{let}\, (x, y) = E\, \mathbf{in}\, M \mid (E, V) \mid (V, E)$
	$\mid \mathbf{inl}\, E \mid \mathbf{inr}\, E \mid \mathbf{case}\, E\, \mathbf{of}\, \{\mathbf{inl}\, x \mapsto M; \mathbf{inr}\, x \mapsto N\}$
	$\mid \mathbf{fork}\, E \mid \mathbf{send}\, E\, M \mid \mathbf{send}\, V\, E \mid \mathbf{receive}\, E \mid \mathbf{close}\, E$
	$\mid \mathbf{cancel}\, E \mid \mathbf{try}\, E\, \mathbf{as}\, x\, \mathbf{in}\, M\, \mathbf{otherwise}\, N$
Pure Contexts	$P ::= [\,] \mid P\, M \mid V\, P$
	$\mid \mathbf{let}\, () = P\, \mathbf{in}\, M \mid \mathbf{let}\, (x, y) = P\, \mathbf{in}\, M \mid (P, V) \mid (V, P)$
	$\mid \mathbf{inl}\, P \mid \mathbf{inr}\, P \mid \mathbf{case}\, P\, \mathbf{of}\, \{\mathbf{inl}\, x \mapsto M; \mathbf{inr}\, x \mapsto N\}$
	$\mid \mathbf{fork}\, P \mid \mathbf{send}\, P\, M \mid \mathbf{send}\, V\, P \mid \mathbf{receive}\, P \mid \mathbf{close}\, P$
	$\mid \mathbf{cancel}\, P$
Thread Contexts	$\mathcal{F} ::= \phi E$
Configuration Contexts	$\mathcal{G} ::= [\,] \mid (\nu a)\mathcal{G} \mid \mathcal{G} \parallel \mathcal{C}$

Syntactic Sugar

$$\lightning\, V \triangleq \lightning\, a_1 \parallel \cdots \parallel \lightning\, a_n \text{ where } \mathsf{fn}(V) = \{a_i\}_i$$
$$\lightning\, P \triangleq \lightning\, a_1 \parallel \cdots \parallel \lightning\, a_n \text{ where } \mathsf{fn}(P) = \{a_i\}_i$$
$$\lightning\, E \triangleq \lightning\, a_1 \parallel \cdots \parallel \lightning\, a_n \text{ where } \mathsf{fn}(E) = \{a_i\}_i$$

Fig. 5. Runtime Syntax

In addition to program threads, configurations include three special forms of thread. A *zapper thread* ($\lightning\, a$) manages an endpoint a that has been cancelled, and is used to propagate failure. A *halted thread* (**halt**) arises when the main thread has crashed due to an uncaught exception. A *buffer thread* ($a(\overrightarrow{V}) \rightsquigarrow b(\overrightarrow{W})$) is used to model asynchrony, where $\overrightarrow{V}$ is a sequence of values ready to be received along endpoint a, and $\overrightarrow{W}$ is a sequence of values ready to be received along endpoint b. We will sometimes find it useful to distinguish top-level threads $\mathcal{T}$ (main threads and halted threads) from auxiliary threads $\mathcal{A}$ (child threads, zapper threads, and buffer threads).

Environments. We extend type environments Γ to include runtime names of session type and introduce runtime type environments Δ, which type both buffer endpoints of session type and channels of type $S^\sharp$ for some S, but not variables.

Contexts. Evaluation contexts E are set up for standard left-to-right call-by-value evaluation. Pure contexts P are those evaluation contexts that include no exception handling frames. Thread contexts $\mathcal{F}$ support reduction in program threads. Configuration contexts $\mathcal{G}$ support reduction under ν-binders and parallel composition.

Free Names. We let the meta operation $\mathsf{fn}(-)$ denote the set of free names in a term, type environment, buffer environment, value, configuration, pure context, or evaluation context.

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

$$
\begin{array}{lrcl}
\text{E-Lam} & (\lambda x.M)\ V & \longrightarrow_{\mathsf{M}} & M\{V/x\} \\
\text{E-Unit} & \mathbf{let}\ () = ()\ \mathbf{in}\ M & \longrightarrow_{\mathsf{M}} & M \\
\text{E-Pair} & \mathbf{let}\ (x,y) = (V,W)\ \mathbf{in}\ M & \longrightarrow_{\mathsf{M}} & M\{V/x, W/y\} \\
\text{E-Inl} & \mathbf{case}\ \mathbf{inl}\ V\ \mathbf{of}\ \{\mathbf{inl}\ x \mapsto M; \mathbf{inr}\ x \mapsto N\} & \longrightarrow_{\mathsf{M}} & M\{V/x\} \\
\text{E-Inr} & \mathbf{case}\ \mathbf{inr}\ V\ \mathbf{of}\ \{\mathbf{inl}\ x \mapsto M; \mathbf{inr}\ x \mapsto N\} & \longrightarrow_{\mathsf{M}} & N\{V/x\} \\
\text{E-Val} & \mathbf{try}\ V\ \mathbf{as}\ x\ \mathbf{in}\ M\ \mathbf{otherwise}\ N & \longrightarrow_{\mathsf{M}} & M\{V/x\} \\
\text{E-Lift} & E[M] & \longrightarrow_{\mathsf{M}} & E[M'], \quad \text{if } M \longrightarrow_{\mathsf{M}} M'
\end{array}
$$

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

$$
\mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E}) \equiv (\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E} \qquad \mathcal{C} \parallel \mathcal{D} \equiv \mathcal{D} \parallel \mathcal{C} \qquad (\nu a)(\nu b)\mathcal{C} \equiv (\nu b)(\nu a)\mathcal{C}
$$

$$
\mathcal{C} \parallel (\nu a)\mathcal{D} \equiv (\nu a)(\mathcal{C} \parallel \mathcal{D}), \quad \text{if } a \notin \mathsf{fn}(\mathcal{C}) \qquad \mathcal{G}[\mathcal{C}] \equiv \mathcal{G}[\mathcal{D}], \quad \text{if } \mathcal{C} \equiv \mathcal{D}
$$

$$
a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W}) \equiv b(\overrightarrow{W}) \leftrightsquigarrow a(\overrightarrow{V}) \qquad \circ () \parallel \mathcal{C} \equiv \mathcal{C} \qquad (\nu a)(\nu b)(\lightning a \parallel \lightning b \parallel a(\epsilon) \leftrightsquigarrow b(\epsilon)) \parallel \mathcal{C} \equiv \mathcal{C}
$$

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

$$
\begin{array}{lrcl}
\text{E-Fork} & \mathcal{F}[\mathbf{fork}\ (\lambda x.M)] & \longrightarrow & (\nu a)(\nu b)(\mathcal{F}[a] \parallel \circ M\{b/x\} \parallel a(\epsilon) \leftrightsquigarrow b(\epsilon)), \\
 & & & \quad \text{where } a, b \text{ are fresh} \\
\text{E-Send} & \mathcal{F}[\mathbf{send}\ U\ a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W}) & \longrightarrow & \mathcal{F}[a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W} \cdot U) \\
\text{E-Receive} & \mathcal{F}[\mathbf{receive}\ a] \parallel a(U \cdot \overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W}) & \longrightarrow & \mathcal{F}[(U,a)] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W}) \\
\text{E-Close} & (\nu a)(\nu b)(\mathcal{F}[\mathbf{close}\ a] \parallel \mathcal{F}'[\mathbf{close}\ b] \parallel a(\epsilon) \leftrightsquigarrow b(\epsilon)) & \longrightarrow & \mathcal{F}[()] \parallel \mathcal{F}'[()] \\
\text{E-Cancel} & \mathcal{F}[\mathbf{cancel}\ a] & \longrightarrow & \mathcal{F}[()] \parallel \lightning a \\
\text{E-Zap} & \lightning a \parallel a(U \cdot \overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W}) & \longrightarrow & \lightning a \parallel \lightning U \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W}) \\
\text{E-CloseZap} & (\nu a)(\nu b)(\mathcal{F}[\mathbf{close}\ a] \parallel \lightning b \parallel a(\epsilon) \leftrightsquigarrow b(\epsilon)) & \longrightarrow & \mathcal{F}[()] \\
\text{E-ReceiveZap} & \mathcal{F}[\mathbf{receive}\ a] \parallel \lightning b \parallel a(\epsilon) \leftrightsquigarrow b(\overrightarrow{W}) & \longrightarrow & \mathcal{F}[\mathbf{raise}] \parallel \lightning a \parallel \lightning b \parallel a(\epsilon) \leftrightsquigarrow b(\overrightarrow{W}) \\
\text{E-Raise} & \mathcal{F}[\mathbf{try}\ P[\mathbf{raise}]\ \mathbf{as}\ x\ \mathbf{in}\ M\ \mathbf{otherwise}\ N] & \longrightarrow & \mathcal{F}[N] \parallel \lightning P \\
\text{E-RaiseChild} & \circ P[\mathbf{raise}] & \longrightarrow & \lightning P \\
\text{E-RaiseMain} & \bullet P[\mathbf{raise}] & \longrightarrow & \mathbf{halt} \parallel \lightning P \\
\text{E-LiftC} & \mathcal{G}[\mathcal{C}] & \longrightarrow & \mathcal{G}[\mathcal{D}], \quad \text{if } \mathcal{C} \longrightarrow \mathcal{D} \\
\text{E-LiftM} & \phi M & \longrightarrow & \phi M', \quad \text{if } M \longrightarrow_{\mathsf{M}} M'
\end{array}
$$

Fig. 6. Reduction and Equivalence for Terms and Configurations

Syntactic Sugar. We follow the standard convention that parallel composition of configurations associates to the right. We write $\lightning V$, $\lightning P$, and $\lightning E$, as shorthand for the parallel composition of zapper threads for each free name in values V, pure contexts P, and evaluation contexts E, respectively.

Following prior work on linear functional languages with session types [Gay and Vasconcelos 2010; Lindley and Morris 2015, 2016, 2017], we present the semantics of EGV via a deterministic reduction relation on terms ($\longrightarrow_{\mathsf{M}}$), an equivalence relation on configurations ($\equiv$), and a nondeterministic reduction relation on configurations ($\longrightarrow$). We write $\Longrightarrow$ for the relation $\equiv \longrightarrow \equiv$. Fig. 6 presents reduction and equivalence rules for terms and configurations.

Term Reduction. Reduction on terms is standard call-by-value β-reduction.

Configuration Equivalence. A running program can make use of the standard structural π-calculus equivalence rules [Milner 1999] of associativity and commutativity of parallel composition and name restriction; scope extrusion; and congruence. We incorporate a rule to allow buffers to be

treated symmetrically and two garbage collection rules, allowing completed child threads and cancelled empty buffers to be discarded.

Communication and Concurrency. The E-Fork rule creates two fresh names for each endpoint of a channel, returning one name and substituting the other in the body of the spawned thread, as well as creating a channel with two empty buffers. The E-Send and E-Receive rules send to and receive from a buffer. The E-Close rule discards an empty buffer once a session is complete.

Cancellation. The E-Cancel rule cancels an endpoint by creating a zapper thread. The E-Zap rule ensures that when an endpoint is cancelled, all other endpoints stored in the buffer of the cancelled endpoint are also cancelled: it dequeues a value from the head of the buffer and cancels any endpoints contained within the dequeued value. It is applied repeatedly until the buffer is empty. The E-CloseZap rule closes an endpoint whose peer endpoint has been cancelled.

Raising Exceptions. Following Mostrous and Vasconcelos [2014], an exception is raised when it would be otherwise impossible for a communication action to succeed. The E-ReceiveZap rule raises an exception if an attempt is made to receive along a cancelled endpoint whose buffer is empty. There is no rule for the case where a thread tries to send a value along a cancelled endpoint; the free names in the communicated value must eventually be cancelled, but this is achieved through E-Zap. We choose not to raise an exception in this case since to do so would violate confluence, which we discuss in more detail in §3.5. Not raising exceptions on message sends is standard for languages such as Erlang.

Handling Exceptions. The E-Raise rule invokes the **otherwise** clause if an exception is raised, while also cancelling all endpoints in the enclosing pure context. If an unhandled exception occurs in a child thread, then all free endpoints in the evaluation context are cancelled and the thread is terminated (E-RaiseChild). If the exception is in the main thread then all free endpoints are cancelled and the main thread reduces to **halt** (E-RaiseMain).

3 METATHEORY

Even in the presence of channel cancellation and exceptions, EGV retains all of GV's strong metatheoretic properties [Lindley and Morris 2015].

The central property of session-typed systems is session fidelity: all communication along session channels follows the prescribed session types. Session fidelity follows as a corollary of the preservation of configuration typeability under reduction.

Session calculi with roots in linear logic exhibit deadlock-freedom since interpreting the logical cut rule as a combination of a name restriction and parallel composition necessarily ensures acyclicity of configurations. Coupled with appropriate reduction rules, it is also possible to use deadlock-freedom to derive a global progress result. We prove that global progress holds even in the presence of channel cancellation. (Our proof is direct, not requiring catalyser processes [Carbone et al. 2014; Mostrous and Vasconcelos 2014].) We also prove that EGV is confluent and terminating.

3.1 Runtime Typing

In order to state our main results we require typing rules for names and configurations. These are given in Fig. 7. The configuration typing judgement has the shape $\Gamma; \Delta \vdash^{\phi} C$, which states that under type environment Γ, buffer environment Δ, and thread flag ϕ, configuration C is well-typed. Thread flags ensure that there can be at most one top-level thread which can return a value: $\bullet$ denotes a configuration with a top-level thread and $\circ$ denotes a configuration without. The main thread returns the result of running a program. Any configuration C such that $\Gamma; \Delta \vdash^{\bullet} C$ has exactly one main thread or halted thread as a subconfiguration.

Term Typing $\boxed{\Gamma \vdash M : A}$

T-Name
$$\frac{}{a : S \vdash a : S}$$

Session Slicing $\boxed{S/\overrightarrow{A}}$

$$S/\epsilon = S$$
$$!A.S/A \cdot \overrightarrow{A} = S/\overrightarrow{A}$$

Queue Typing $\boxed{\Gamma \vdash \overrightarrow{V} : \overrightarrow{A}}$

$$\frac{}{\cdot \vdash \epsilon : \epsilon} \qquad \frac{\Gamma_1 \vdash V : A \qquad \Gamma_2 \vdash \overrightarrow{V} : \overrightarrow{A}}{\Gamma_1, \Gamma_2 \vdash V \cdot \overrightarrow{V} : A \cdot \overrightarrow{A}}$$

Configuration Typing $\boxed{\Gamma; \Delta \vdash^{\phi} C}$

T-Nu
$$\frac{\Gamma; \Delta, a : S^{\sharp} \vdash^{\phi} C}{\Gamma; \Delta \vdash^{\phi} (\nu a) C}$$

T-Connect$_1$
$$\frac{\Gamma_1, a : S; \Delta_1 \vdash^{\phi_1} C \qquad \Gamma_2; \Delta_2, a : \overline{S} \vdash^{\phi_2} \mathcal{D}}{\Gamma_1, \Gamma_2; \Delta_1, \Delta_2, a : S^{\sharp} \vdash^{\phi_1 + \phi_2} C \parallel \mathcal{D}}$$

T-Connect$_2$
$$\frac{\Gamma_1; \Delta_1, a : S \vdash^{\phi_1} C \qquad \Gamma_2, a : \overline{S}; \Delta_2 \vdash^{\phi_2} \mathcal{D}}{\Gamma_1, \Gamma_2; \Delta_1, \Delta_2, a : S^{\sharp} \vdash^{\phi_1 + \phi_2} C \parallel \mathcal{D}}$$

T-Mix
$$\frac{\Gamma_1; \Delta_1 \vdash^{\phi_1} C \qquad \Gamma_2; \Delta_2 \vdash^{\phi_2} \mathcal{D}}{\Gamma_1, \Gamma_2; \Delta_1, \Delta_2 \vdash^{\phi_1 + \phi_2} C \parallel \mathcal{D}}$$

T-Main
$$\frac{\Gamma \vdash M : A}{\Gamma; \cdot \vdash^{\bullet} \bullet M}$$

T-Child
$$\frac{\Gamma \vdash M : \mathbf{1}}{\Gamma; \cdot \vdash^{\circ} \circ M}$$

T-Halt
$$\frac{}{\cdot; \cdot \vdash^{\bullet} \mathbf{halt}}$$

T-Zap
$$\frac{}{a : S; \cdot \vdash^{\circ} \lightning a}$$

T-Buffer
$$\frac{S/\overrightarrow{A} = \overline{S'/\overrightarrow{B}} \qquad \Gamma_1 \vdash \overrightarrow{V} : \overrightarrow{A} \qquad \Gamma_2 \vdash \overrightarrow{W} : \overrightarrow{B}}{\Gamma_1, \Gamma_2; a : S, b : S' \vdash^{\circ} a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}$$

Flag Combination $\boxed{\phi_1 + \phi_2 = \phi_3}$

$\bullet + \circ = \bullet$ $\qquad \circ + \bullet = \bullet$
$\circ + \circ = \circ$ $\qquad \bullet + \bullet$ undefined

Session Type Reduction $\boxed{S \longrightarrow S'}$

$$?A.S \longrightarrow S \qquad !A.S \longrightarrow S$$

Environment Reduction $\boxed{\Gamma; \Delta \longrightarrow \Gamma'; \Delta'}$

$$\frac{S \longrightarrow S'}{\Gamma, a : S; \Delta \longrightarrow \Gamma, a : S'; \Delta} \qquad \frac{S \longrightarrow S'}{\Gamma; \Delta, a : S \longrightarrow \Gamma; \Delta, a : S'} \qquad \frac{S \longrightarrow S'}{\Gamma; \Delta, a : S^{\sharp} \longrightarrow \Gamma; \Delta, a : S'^{\sharp}}$$

Fig. 7. Runtime Typing

The T-Nu rule introduces a channel name. The T-Connect$_1$ and T-Connect$_2$ rules connect two configurations over a channel. The T-Mix rule composes in parallel two independent configurations that share no channels. The three rules for parallel composition use the + operator to combine the flags from the subconfigurations.

The T-Main and T-Child rules introduce main and child threads. Child threads always return the unit value. The T-Halt rule types the **halt** configuration, which signifies that an unhandled exception has occurred in the main thread. The T-Zap rule types a zapper thread, given a single name in the type environment. The T-Buffer rule ensures that buffers contain values corresponding to the session types of their endpoints. This is the only rule that consumes names from the buffer environment. Buffers rely on two auxiliary judgements. The queue typing judgement $\Gamma \vdash \overrightarrow{V} : \overrightarrow{A}$ states that under type environment Γ, the sequence of values $\overrightarrow{V}$ have types $\overrightarrow{A}$. The session slicing operator $S/\overrightarrow{A}$ captures reasoning about session types discounting values contained in the buffer. The session types of two buffer endpoints are compatible if they are dual up to values contained in the buffer. The partiality of the slicing operator ensures that one queue in a buffer is empty.

3.2 Preservation

Preservation for the functional fragment of EGV is standard.

Lemma 3.1 (Preservation (Terms)). *If* $\Gamma \vdash M : A$ *and* $M \longrightarrow_M M'$, *then* $\Gamma \vdash M' : A$.

Preservation of typing by configuration reduction holds only for closed configurations.

Relation Notation. Given a relation R, we write $R^?$ for its reflexive closure.
We write Ψ for the restriction of type environments Γ to contain runtime names but no variables:

$$\Psi ::= \cdot \mid \Psi, a : S$$

Theorem 3.2 (Preservation). *If* $\Psi; \Delta \vdash^\phi C$ *and* $C \longrightarrow C'$, *then there exist* Ψ', Δ' *such that* $\Psi; \Delta \longrightarrow^? \Psi'; \Delta'$ *and* $\Psi'; \Delta' \vdash^\phi C'$.

Proof. By induction on the derivation of $C \longrightarrow C'$, making use of Lemma 3.1, and lemmas for subconfiguration typeability and replacement. The proof cases can be found in §A.1 of the supplementary material. □

Typing and Configuration Equivalence. As is common in functional languages inspired by logical accounts of session-typed concurrency [Lindley and Morris 2015, 2017], typeability of configurations is *not* preserved by equivalence. As an example, consider $\Gamma; \Delta \vdash^\phi (\nu a)(\nu b)(C \parallel (\mathcal{D} \parallel \mathcal{E}))$ where $a \in \mathsf{fn}(C)$, $b \in \mathsf{fn}(\mathcal{D})$, and $a, b \in \mathsf{fn}(\mathcal{E})$. However, $\Gamma; \Delta \nvdash^\phi (\nu a)(\nu b)((C \parallel \mathcal{D}) \parallel \mathcal{E})$, since only a or b would be present in the type environment when typing $\mathcal{E}$.

Fortunately the looseness of the equivalence relation is not a problem, as any reduction sequence that passes through an ill-typed configuration is equivalent to one that does not.

Theorem 3.3 (Preservation Modulo Equivalence). *If* $\Psi; \Delta \vdash^\phi C$, $C \equiv \mathcal{D}$, *and* $\mathcal{D} \longrightarrow \mathcal{D}'$, *then:*
(1) There exists some $\mathcal{E}$ *such that* $\mathcal{D} \equiv \mathcal{E}$, *and* $\Psi; \Delta \vdash^\phi \mathcal{E}$, *and* $\mathcal{E} \longrightarrow \mathcal{E}'$
(2) There exist Ψ', Δ' *such that* $\Psi; \Delta \longrightarrow^? \Psi'; \Delta'$ *and* $\Psi'; \Delta' \vdash^\phi \mathcal{E}'$
(3) $\mathcal{D}' \equiv \mathcal{E}'$

Proof sketch. The only non-trivial cases of reduction are those that involve a synchronisation with a buffer (E-Send, E-Receive, E-Close, E-Zap, E-CloseZap, E-ReceiveZap). The only equivalence rule that can lead to an ill-typed configuration is associativity of parallel composition

$$C \parallel (\mathcal{D} \parallel \mathcal{E}) \equiv (C \parallel \mathcal{D}) \parallel \mathcal{E}$$

where both compositions arise from the T-Connect$_1$ and T-Connect$_2$ rules. The only reason to apply the associativity rule from left-to-right is in order to enable threads inside C and $\mathcal{D}$ to synchronise. But for synchronisation to be possible there must exist a name a such that $a \in \mathsf{fn}(C)$ and $a \in \mathsf{fn}(\mathcal{D})$. Because the left-hand-side is well-typed, we know that C and $\mathcal{E}$ have no names in common, that $\mathcal{D}$ and $\mathcal{E}$ share a name, and that the right-hand-side must be well-typed as there is still exactly one channel connecting each of the parallel compositions. The argument for the case of applying the rule from right-to-left is symmetric. In summary, any ill-typed use of equivalence is useless, as it does not enable any more reductions. □

3.3 Deadlock-freedom

Due to its correspondence with linear logic, GV is naturally deadlock-free. As none of the new constructs introduce cycles between threads, the same holds for EGV. The proof of deadlock-freedom is thus similar to that for GV [Lindley and Morris 2015].

We begin by classifying the notion of a *ready thread*: that is, a thread which is ready to perform an action on some channel endpoint.

Definition 3.4. We say that term M is *ready to perform an action on name a* if M is about to send on, receive on, close, or cancel a. Formally:

$$\mathsf{ready}(a, M) \triangleq \exists E.(M = E[\mathbf{send} V\, a]) \vee (M = E[\mathbf{receive} a]) \vee (M = E[\mathbf{close} a]) \vee (M = E[\mathbf{cancel} a])$$

Given the notion of a ready thread, we may characterise the notion of a dependency between communication actions.

Definition 3.5. Let C be a buffer such that a and b are not bound by C. We say that *a depends on b in C*, written $\mathsf{depends}(a, b, C)$, if C is a buffer connecting a and b, or a appears in some thread ready to perform an action on b, or if a depends on some name c which depends on b. Formally:

- $\mathsf{depends}(a, b, a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W}))$
- $\mathsf{depends}(a, b, b(\overrightarrow{W}) \leftrightsquigarrow a(\overrightarrow{V}))$
- $\mathsf{depends}(a, b, \phi M) \triangleq \mathsf{ready}(b, M) \wedge a \in \mathsf{fn}(M)$
- $\mathsf{depends}(a, b, C) \triangleq \exists \mathcal{D}, \mathcal{E}, c. C \equiv \mathcal{G}[\mathcal{D} \parallel \mathcal{E}] \wedge \mathsf{depends}(a, c, \mathcal{D}) \wedge \mathsf{depends}(c, b, \mathcal{E})$

Remark. The above definition of dependency is an overapproximation to the intuitive notion, as a buffer need not have dependencies in both directions, but for our purposes this does not matter. □

Definition 3.6. We say that a configuration is *deadlocked* if it contains cyclic dependencies:

$$\mathsf{deadlocked}(C) \triangleq \exists \mathcal{D}, \mathcal{E}, a, b.\ C \equiv \mathcal{G}[\mathcal{D} \parallel \mathcal{E}] \wedge \mathsf{depends}(a, b, \mathcal{D}) \wedge \mathsf{depends}(b, a, \mathcal{E})$$

With these definitions in place, we can show that EGV terms are deadlock-free. The first step is to show that at most one name is shared between two configurations.

Lemma 3.7. *If $\Gamma; \Delta \vdash^{\phi} C$ and $\exists \mathcal{D}, \mathcal{E}.\ C = \mathcal{G}[\mathcal{D} \parallel \mathcal{E}]$, then $\mathsf{fn}(\mathcal{D}) \cap \mathsf{fn}(\mathcal{E})$ is either $\emptyset$ or $\{a\}$ for some name a.*

Proof. By induction on the derivation of $\Gamma; \Delta \vdash^{\phi} C$, due to the partitioning of the type environment and buffer environment in the typing rules for parallel composition. The T-Connect$_1$ and T-Connect$_2$ rules allow exactly one name to be shared, whereas T-Mix forbids sharing of names. □

We use this lemma to prove that well-typed configurations are deadlock-free.

Theorem 3.8. *If $\Gamma; \Delta \vdash C$, then $\neg\mathsf{deadlocked}(C)$.*

Proof. By contradiction. By the definition of deadlocked, we know that there must be some cyclic dependency, which would be ill-typed due to Lemma 3.7. □

3.4 Progress

Deadlock-freedom alone is not sufficient to prove progress, especially in the presence of channel cancellation. In order to prove that EGV enjoys a strong notion of progress we identify a notion of *canonical form* for configurations. We prove that every well-typed configuration is equivalent to a well-typed configuration one in canonical form, and that irreducible configurations that do not reduce and contain no free channels are equivalent to either a value or to **halt**.

The functional fragment of EGV satisfies a progress property. The proof is a standard induction on typing derivations.

Lemma 3.9 (Progress: Term Reduction). *If $\Psi \vdash M : A$, then either:*

- *M is a value;*
- *there exists some M' such that $M \longrightarrow_M M'$; or*

- *M has the form $E[M']$, where M' is a session typing primitive of the form:* $\mathbf{fork}\ V$, $\mathbf{send}\ V\ W$, $\mathbf{receive}\ V$, $\mathbf{close}\ V$, *or* $\mathbf{cancel}\ V$.

To reason about progress of configurations, we characterise *canonical forms*, which make explicit the property that at most one name is shared between threads. Recall that $\mathcal{A}$ ranges over auxiliary threads and $\mathcal{T}$ over top-level threads (Fig. 5). Let $\mathcal{M}$ range over configurations of the form:

$$\mathcal{A}_1 \parallel \cdots \parallel \mathcal{A}_m \parallel \mathcal{T}$$

Definition 3.10 (Canonical Form). A configuration C is in *canonical form* if there is a sequence of names $a_1, \ldots, a_n$, a sequence of configurations $\mathcal{A}_1, \ldots, \mathcal{A}_n$, and a configuration $\mathcal{M}$, such that:

$$C = (\nu a_1)(\mathcal{A}_1 \parallel (\nu a_2)(\mathcal{A}_2 \parallel \cdots \parallel (\nu a_n)(\mathcal{A}_n \parallel \mathcal{M}) \ldots))$$

Each thread $\mathcal{A}_i$ appearing next to a ν-binder is composed using either T-Connect$_1$ or T-Connect$_2$, whereas each thread in $\mathcal{M}$ is composed using T-Mix. All well-typed configurations can be written in canonical form.

Theorem 3.11 (Canonical Forms). *Given C such that $\Gamma; \Delta \vdash^\bullet C$, there exists some $C' \equiv C$ such that $\Gamma; \Delta \vdash^\bullet C'$ and C' is in canonical form.*

Proof. By induction on the count of ν-bound variables, following Lindley and Morris [2015]. The additional features of EGV do not change the essential argument. The full proof can be found in §A.2 of the supplementary material. □

Armed with the notion of a canonical form, we can proceed to classify the nature of configurations which do not reduce.

Theorem 3.12. *Suppose $\Psi; \Delta \vdash^\bullet C$ where C is in canonical form and $C \not\Longrightarrow$.*
Let $C = (\nu a_1)(\mathcal{A}_1 \parallel (\nu a_2)(\mathcal{A}_2 \parallel \cdots \parallel (\nu a_n)(\mathcal{A}_n \parallel \mathcal{M})) \ldots))$.
(1) For $1 \le i \le n$, each thread in $\mathcal{A}_i$ is either:
  (a) a child thread $\circ M$ for which there exists $a \in \{a_j \mid 1 \le j \le i\} \cup \mathsf{fn}(\Psi)$ such that $\mathsf{ready}(a, M)$;
  (b) a zapper thread $\lightning a$ such that $a \in \{a_j \mid 1 \le j \le i\} \cup \mathsf{fn}(\Psi)$; or
  (c) a buffer.
(2) $\mathcal{M} = \mathcal{A}'_1 \parallel \cdots \parallel \mathcal{A}'_m \parallel \mathcal{T}$ such that for $1 \le j \le m$:
  (a) $\mathcal{A}'_j$ is either:
    (i) a child thread $\circ N$ such that N is a value or $\mathsf{ready}(a, N)$ for some $a \in \{a_i \mid 1 \le i \le n\} \cup \mathsf{fn}(\Psi)$;
    (ii) a zapper thread $\lightning a$ for some $a \in \{a_i \mid 1 \le i \le n\} \cup \mathsf{fn}(\Psi)$; or
    (iii) a buffer.
  (b) Either:
    (i) $\mathcal{T} = \bullet N$, where either N is a value, or $\mathsf{ready}(a, N)$ for some $a \in \{a_i \mid 1 \le i \le n\} \cup \mathsf{fn}(\Psi)$; or
    (ii) $\mathcal{T} = \mathbf{halt}$.

The proof considers the form of each subconfiguration. By Lemma 3.9, we know that each thread must either be a value or of the form $E[M]$, where M is a communication or concurrency action. This action cannot be **fork** since it could reduce, thus the thread must be ready to perform an action on a channel, which by the typing rules must either be in the environment or a preceding ν-bound variable. As a corollary, we obtain a more precise result for closed configurations.

Corollary 3.13. *Suppose $\cdot; \cdot \vdash^\bullet C$ where C is in canonical form and $C \not\Longrightarrow$.*
Let $C = (\nu a_1)(\mathcal{A}_1 \parallel (\nu a_2)(\mathcal{A}_2 \parallel \cdots \parallel (\nu a_n)(\mathcal{A}_n \parallel \mathcal{M}) \ldots))$.
Then:

(1) *For* $1 \le i \le n$, *every thread in* $\mathcal{A}_i$ *is either:*
   (a) *a child thread* $\circ M$ *for some* M *such that* $\mathsf{ready}(a_i, M)$*; or*
   (b) *a zapper thread* $\lightning a_i$*; or*
   (c) *a buffer.*
(2) $\mathcal{M} = \mathcal{A}'_1 \parallel \cdots \parallel \mathcal{A}'_m \parallel \mathcal{T}$ *such that for* $1 \le j \le m$*:*
   (a) *Each* $\mathcal{A}'_j$ *is either a fully-reduced child thread* $\circ V$*, a zapper thread* $\lightning a_i$ *for some* $1 \le i \le n$*, or a buffer.*
   (b) *Either* $\mathcal{T} = \bullet W$*, for some value* W*, or* $\mathcal{T} = \mathbf{halt}$*.*

We now come to our main result: if the top-level thread has reduced to a value which contains no free channels, then the entire configuration is equivalent to a value. If the top-level thread has halted, we can show that the entire configuration is equivalent to **halt**. These results are established as a consequence of Corollary 3.13, along with the garbage collection congruences of Fig. 6.

Theorem 3.14 (Global Progress). *Let* $\cdot;\cdot \vdash^{\bullet} C$ *and* $C \Longrightarrow$*, and let* $\mathcal{D}$ *be the top-level thread of* C*. If* $\mathcal{D} = \bullet V$ *and* $\mathsf{fn}(V) = \emptyset$*, then* $C \equiv \bullet V$*. If* $\mathcal{D} = \mathbf{halt}$*, then* $C \equiv \mathbf{halt}$*.*

3.5 Confluence

EGV enjoys a strong form of confluence known as the *diamond property* [Barendregt 1984].

Theorem 3.15 (Diamond Property).
If $\Psi; \Delta \vdash^{\phi} C$*, and* $C \Longrightarrow \mathcal{D}_1$*, and* $C \Longrightarrow \mathcal{D}_2$*, then either* $\mathcal{D}_1 \equiv \mathcal{D}_2$*, or there exists some* $\mathcal{D}_3$ *such that* $\mathcal{D}_2 \Longrightarrow \mathcal{D}_3$ *and* $\mathcal{D}_2 \Longrightarrow \mathcal{D}_3$*.*

Proof. First, note that $\longrightarrow_{\mathsf{M}}$ is entirely deterministic and hence confluent due to the call-by-value, left-to-right ordering imposed by evaluation contexts. By linearity, we know that endpoints to different buffers may not be shared, so it follows that communication actions on different channels may be performed in any order. Nevertheless, two critical pairs arise due to asynchrony. The first arises when it is possible to send to or receive from a buffer. There is a choice of whether the send or the receive happens first. Both cases reduce to the same configuration after a single further step.

$$\begin{array}{ccc} & \mathcal{F}[\mathbf{send}\ U\ a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(V \cdot \overrightarrow{W}) \parallel \mathcal{F}'[\mathbf{receive}\ b] & \\ \swarrow & & \searrow \\ \mathcal{F}[a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(V \cdot \overrightarrow{W} \cdot U) \parallel \mathcal{F}'[\mathbf{receive}\ b] & & \mathcal{F}[\mathbf{send}\ U\ a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W}) \parallel \mathcal{F}'[(W, b)] \\ \searrow & & \swarrow \\ & \mathcal{F}[a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W} \cdot U) \parallel \mathcal{F}'[(W, b)] & \end{array}$$

The second critical pair arises when sending to a buffer where the peer endpoint has a non-empty buffer and has been cancelled. There is a choice over whether the value at the head of the queue is cancelled before or after the send takes place. Again, both cases reduce to the same configuration after a single further step.

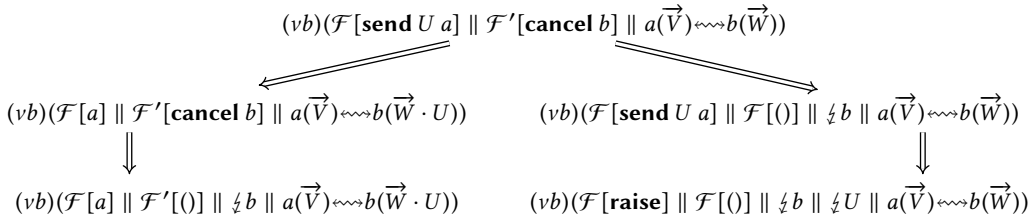
$$\begin{array}{ccc} & \mathcal{F}[\mathbf{send}\ U\ a] \parallel \lightning b \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(V \cdot \overrightarrow{W}) & \\ \swarrow & & \searrow \\ \mathcal{F}[a] \parallel \lightning b \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(V \cdot \overrightarrow{W} \cdot U) & & \mathcal{F}[\mathbf{send}\ U\ a] \parallel \lightning b \parallel \lightning V \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W}) \\ \searrow & & \swarrow \\ & \mathcal{F}[a] \parallel \lightning b \parallel \lightning V \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W} \cdot U) & \end{array}$$

□

Remark. The system becomes non-confluent if we choose to raise an exception when sending to a cancelled buffer. Say we were to replace E-Send with the following two rules:

$$(\nu b)(\mathcal{F}[\mathbf{send}\ U\ a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W}) \parallel \phi M) \longrightarrow (\nu b)(\mathcal{F}[a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W} \cdot U) \parallel \phi M)$$
$$\mathcal{F}[\mathbf{send}\ U\ a] \parallel \lightning b \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W}) \longrightarrow \mathcal{F}[\mathbf{raise}] \parallel \lightning b \parallel \lightning U \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$$

Then, configuration $(\nu b)(\mathcal{F}[\mathbf{send} U a] \parallel \mathcal{F}'[\mathbf{cancel} b] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W}))$ results in a non-convergent critical pair, as follows:



In either case, the endpoints contained in U will still eventually be cancelled, thus preservation and global progress still hold. However, the lack of confluence affects exactly *when* the exception is raised in context $\mathcal{F}$. This decision has practical relevance, in that it characterises the race between sending a message and propagating a cancellation notification in the distributed setting. □

3.6 Termination

As EGV is linear, it has an elementary strong normalisation proof.

Theorem 3.16 (Strong normalisation). *If $\Psi; \Delta \vdash^{\phi} C$, then there are no infinite $\Longrightarrow$ reduction sequences from C.*

Proof. Let the size of a configuration be the sum of the sizes of the abstract syntax trees of all of the terms contained in its main threads, child threads, and buffers, modulo garbage collection (i.e. exhaustively applying the rule $\circ() \parallel C \equiv C$ from left-to-right). The size of a configuration is invariant under $\equiv$ and strictly decreases under $\longrightarrow$, hence $\Longrightarrow$ reduction must always terminate. □

We conjecture that the strong normalisation result continues to hold in the presence of unrestricted types or shared channels for session initiation, but the proof technique is necessarily more involved. We believe that a logical relations argument along the lines of Pérez et al. [2012] or a CPS translation along the lines of Lindley and Morris [2016] would suffice.

4 EXTENSIONS

4.1 User-defined Exceptions with Payloads

In order to focus on the interplay between exceptions and session types we have thus far considered the simplest case of exception handling, in which there is a single kind of exception. In practice it can be useful to distinguish between multiple kinds of user-defined exception, each of which may carry a payload.

Consider again the example of handling the exception in checkDetails. Two reasons why the exception may arise are that the database is corrupt, or that there are too many connections. We

Syntax

Types	$A, B ::= \cdots \mid \mathsf{Exn}$
Terms	$L, M, N ::= \cdots \mid X(M) \mid \mathbf{raise}\ M \mid \mathbf{try}\ L\ \mathbf{as}\ x\ \mathbf{in}\ M\ \mathbf{unless}\ H$
Exception Handlers	$H ::= \{X_i(x_i) \mapsto N_i\}_i$

Runtime Syntax

Evaluation Contexts $E ::= \cdots \mid \mathbf{raise}\ E \mid \mathbf{try}\ E\ \mathbf{as}\ x\ \mathbf{in}\ M\ \mathbf{unless}\ H$

Term typing $\boxed{\Sigma(X) = A}$ $\boxed{\Gamma \vdash M : A}$

$$\text{TP-Exn}\ \frac{\Sigma(X) = A \qquad \Gamma \vdash M : A}{\Gamma \vdash X(M) : \mathsf{Exn}} \qquad \text{TP-Raise}\ \frac{\Gamma \vdash M : \mathsf{Exn}}{\Gamma \vdash \mathbf{raise}\ M : A}$$

$$\text{TP-Try}\ \frac{\Gamma_1 \vdash L : A \qquad \Gamma_2, x : A \vdash M : B \qquad (\Gamma_2, y_i : \Sigma(X_i) \vdash N_i : B)_i}{\Gamma_1, \Gamma_2 \vdash \mathbf{try}\ L\ \mathbf{as}\ x\ \mathbf{in}\ M\ \mathbf{unless}\ \{X_i(y_i) \mapsto N_i\}_i : B}$$

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

$$\text{EP-Val} \quad \mathbf{try}\ V\ \mathbf{as}\ x\ \mathbf{in}\ M\ \mathbf{unless}\ H \ \longrightarrow_{\mathsf{M}}\ M\{V/x\}$$

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

EP-Raise	$\mathcal{F}[\mathbf{try}\ E[\mathbf{raise}\ X(V)]\ \mathbf{as}\ x\ \mathbf{in}\ M\ \mathbf{unless}\ H]$	$\longrightarrow$	$\mathcal{F}[N\{V/y\}] \parallel \lightning E$	where $X \notin \mathsf{handled}(E)$ $(X(y) \mapsto N) \in H$
EP-RaiseChild	$\circ\, E[\mathbf{raise}\ X(V)]$	$\longrightarrow$	$\lightning E \parallel \lightning V$	where $X \notin \mathsf{handled}(E)$
EP-RaiseMain	$\bullet\, E[\mathbf{raise}\ X(V)]$	$\longrightarrow$	$\mathbf{halt} \parallel \lightning E \parallel \lightning V$	where $X \notin \mathsf{handled}(E)$

Fig. 8. User-defined Exceptions with Payloads

might like to handle each case separately:

```
exnServer4(s) ≜
  let ((username, password), s) = receive (s) in
  try checkDetails(username, password) as res in
    if res then let s = select Authenticated s in serverBody(s)
    else
      let s = select AccessDenied s in close s
  unless
    DBCorrupt(y)          ↦ cancel (s); log("Database Corrupt: " + y)
    TooManyConnections(y) ↦ cancel (s); log("DB Error: Too many connections: " + y)
```

An exception in checkDetails might be raised by the term **raise** DatabaseCorrupt(*filename*), for example. Our approach generalises straightforwardly to handle this example.

Syntax. Figure 8 shows the extensions to EGV to accommodate exceptions with payloads. We introduce a type of exceptions, Exn. We assume a countably infinite set $X \in \mathbb{E}$ of exception names, and a type schema function $\Sigma(X) = A$ mapping exception names to their payload types. We extend **raise** to take a term of type Exn as its argument. Finally, we generalise **try** L **as** x **in** M **otherwise** N to **try** L **as** x **in** M **unless** H, where H is an exception handler with clauses $\{X_i(y_i) \mapsto N_i\}_i$, such that X_i is an exception name; y_i binds the payload; and N_i is the clause to be evaluated when the exception is raised.

Typing Rules. The TP-Exn rule ensures that an exceptions payload matches its expected type. The TP-Raise and TP-Try are the natural extensions of T-Raise and T-Try to accommodate exceptions with payloads.

Semantics. Our formulation is similar to operational accounts of handlers for algebraic effects; the formulation here is inspired by that of Hillerström et al. [2017]. To define the semantics of the generalised exception handling construct, we first introduce the auxiliary function $\mathsf{handled}(H)$, which defines the exceptions handled in a given evaluation context:

$$\mathsf{handled}(P) = \emptyset \qquad \mathsf{handled}(\mathbf{try}\ E\ \mathbf{as}\ x\ \mathbf{in}\ M\ \mathbf{unless}\ H) = \mathsf{handled}(E) \cup \mathsf{dom}(H)$$

$$\mathsf{handled}(E) = \mathsf{handled}(E'), \quad \text{if } E \text{ is not a } \mathbf{try} \text{ and } E' \text{ is the immediate subcontext of } E$$

The EP-Raise rule handles an exception. The side conditions ensure that the exception is caught by the nearest matching handler and is handled by the appropriate clause. As with plain EGV, all free names are safely discarded. The EP-RaiseChild and EP-RaiseMain rules cover the cases where an exception is unhandled. Note that due to the use of the handled function we no longer required pure contexts. All of EGV's metatheoretic properties (preservation, global progress, confluence, and termination) adapt straightforwardly to this extension.

4.2 Unrestricted Types and Access Points

Unrestricted (intuitionistic) types allow some values to be used in a non-linear fashion. Access points [Gay and Vasconcelos 2010] provide a more flexible method of session initiation than **fork**, allowing two threads to dynamically establish a session. Both features are useful in practice: unrestricted types for obvious reasons, and access points because they admit cyclic communication topologies supporting, for instance, racey stateful servers such as chat servers.

Access points decouple spawning a thread from establishing a session. An access point has the unrestricted type $\mathsf{AP}(S)$. The interface for access points is given by the following operations:

$\mathbf{spawn}\ M : \mathbf{1}$	where M has type $\mathbf{1}$
$\mathbf{new}_S : \mathsf{AP}(S)$	
$\mathbf{request}\ M : \overline{S}$	where M has type $\mathsf{AP}(S)$
$\mathbf{accept}\ M : S$	where M has type $\mathsf{AP}(S)$

The $\mathbf{spawn}\ M$ construct spawns M as a new thread; $\mathbf{new}_S$ creates a fresh access point; $\mathbf{request}\ M$ and $\mathbf{accept}\ M$ generate fresh endpoints that are matched up nondeterministically to form channels. With access points we can macro-express **fork**:

$$\begin{aligned}\mathbf{fork}\ M \triangleq\ & \mathbf{let}\ ap = \mathbf{new}_S\ \mathbf{in} \\ & \mathbf{spawn}\ (M\ (\mathbf{accept}\ ap)); \\ & \mathbf{request}\ ap\end{aligned}$$

By decoupling process and channel creation we lose the guarantee that the communication topology is acyclic, and therefore introduce the possibility of deadlock. Nevertheless, a weak form of progress still holds: the only way of getting stuck is deadlock. In the presence of access points the confluence and termination properties no longer hold (access points are nondeterministic and can encode higher-order state and hence Landin's knot).

5 IMPLEMENTATION

In this section we describe our extensions to the Links programming language to incorporate the exception handling functionality of EGV as well as extensions to the Links concurrency runtimes to support distribution. Links [Cooper et al. 2007] is a statically-typed, ML-inspired, impure functional

```
TwoFactorServer ≜
  ?(Username, Password).⊕{
    Authenticated : ServerBody,
    Challenge : !ChallengeKey.?Response.
      ⊕{ Authenticated : ServerBody,
         AccessDenied : End}
    AccessDenied : End}

TwoFactorClient ≜ ¯TwoFactorServer

exnServer3 : TwoFactorServer ⊸ 1
exnServer3(s) ≜
  let ((username, password), s) = receive (s) in
  try checkDetails(username, password) as auth in
    if auth then
      let s = select Authenticated s in
      serverBody(s)
    else
      let s = select AccessDenied s in
      close s
  otherwise
    cancel (s);
    log("Database Error")
```

(a) EGV

```
typename TwoFactorServer =
  ?(Username, Password).[+|
    Authenticated: ServerMain,
    Challenge: !ChallengeKey.?Response.
      [+| Authenticated: ServerMain,
          AccessDenied: End |+],
    AccessDenied: End |+];

typename TwoFactorClient = ~TwoFactorServer;

sig exnServer3 : (TwoFactorServer) ~> ()
fun exnServer3(s) {
  var ((username, password), s) = receive(s);
  try checkDetails(username, password) as auth in {
      if (auth) {
        var s = select Authenticated s;
        serverBody(s)
      } else {
        var s = select AccessDenied s;
        close(s)
      }
  } otherwise {
      cancel(s);
      log("Database Error")
  }
}
```

(b) Links

Fig. 9. Two-factor Authentication Session Types in EGV and Links

programming language designed for the web. Links is designed to allow code for all "tiers" of a web application—client, server, and database—to be written in a single uniform language. Lindley and Morris [2017] extend Links with first-class session types, relying on lightweight linear typing as described by Mazurak et al. [2010] and row polymorphism [Rémy 1994]. We extend their work to account for distributed web applications, which amongst other things necessitates handling failure.

5.1 From Exceptional GV to Links

First, let us see how we may realise the ideas from the core calculus in the Links language itself. Figure 9a shows the definition of the client of the two-factor authentication workflow in the core calculus, and Figure 9b shows the definition in Links. The mapping is rather direct; type aliases are introduced by `typename`, and selection is denoted by `[+| ... |+]`. Although not appearing in the example, offering a choice is denoted by `[&| ... |&]`. Duality is denoted by a tilde (~). A difference between the core calculus and the implementation is that to concentrate on the metatheory, we have a purely linear calculus, and explicitly require endpoints to be closed in EGV. Links allows both linear and unrestricted types. In order to concentrate on programmability in Links, endpoints of type End are unrestricted, and can be implicitly discarded. In the example, `close` is defined as `fun close(_) { () }`.

Type signatures in Links are introduced by `sig`—we assign an explicit type signature to `twoFactorClient`, but Links also supports type reconstruction. The `~>` function arrow denotes that the function is impure and thus cannot be run as part of a database query. Let-bindings are expressed with the `var` keyword.

5.2 The Links Model

Links provides a uniform language for web applications. Client code is compiled to JavaScript, server code is interpreted, and database queries are compiled to SQL. Each client and server has its own concurrency runtime, providing lightweight processes and message passing communication.

Earlier versions of Links [Cooper et al. 2007] invoke a fresh copy of the server per server request and communication between client and server is via RPC calls which invoke a fresh copy of the server, relying on serialisation of continuations in order to maintain server state. However, the web has moved on: in addition to the traditional-request model, *single-page applications* are web applications which are usable without reloading a web page. Advances such as WebSockets allow socket-like bidirectional asynchronous communication between client and server, in turn allowing richer applications where data (for example, comments on a GitHub pull request) flows more freely between client and server. Moving to a model based on lightweight threads and session-typed channels avoids the inversion of control inherent in RPC-style systems, and allows development to be driven by the communication protocol.

In order to better support dynamic single-page applications and multi-user applications such as chat, Links now adopts an application server model, in which the server persists. On top of this we have implemented communication between client and server using session-typed channels. Since channels are a location-transparent abstraction, we also optionally allow the abstraction of client-to-client communication, routed through the server. In future, we are also interested in investigating the use of WebRTC [Bergkvist et al. 2012] for purely client-to-client communication.

5.3 Concurrency

Links provides typed, actor-style concurrency, where processes have a single incoming message queue and can send asynchronous messages. Lindley and Morris [2017] extended Links with session-typed channels, using Links' process-based model but replacing actor mailboxes with session-typed channels. We extend their implementation to support distribution and failure handling.

The client relies on continuation-passing style (CPS), trampolining, and co-operative threading. Client code is compiled to CPS, and explicit `yield` instructions are inserted at every function application. When a process has yielded a given number of times, the continuation is pushed to the back of a queue, and the next process is pulled from the front of the queue. While modern browsers are beginning to integrate tail-recursion, and we have updated the Links library to support it, adoption is not yet widespread. Thus, we periodically discard the call stack using a trampoline. Cooper [2009] discusses the Links client concurrency model in depth. The server implements concurrency on top of the OCaml `lwt` library [Vouillon 2008], which provides lightweight co-operative threading. At runtime, a channel is represented as a pair of endpoint identifiers:

```
(Peer endpoint, Local endpoint)
```

Endpoint identifiers are unique. If a channel (a, b) exists at a given location, then that location should contain a buffer for b.

5.4 Distributed Communication

In order to support bidirectional communication between client and server we use WebSockets [Fette and Melnikov 2011]. A WebSocket connection is established by a client. When a request is made

and a web page is generated, each client is assigned a unique identifier, which it uses to establish a WebSocket connection. Any messages the server attempts to send prior to a WebSocket connection being established are buffered and delivered once the connection is established. Once the WebSocket connection is established, we use a JSON protocol to communicate messages such as access point operations, remote session messages, and endpoint cancellation notifications.

Due to delegation, it is possible that one client will hold one endpoint of a channel, and another client will hold the other endpoint. In order to provide the illusion of client-to-client communication, we route the communication between the two clients via the server. The server maintains a map

$$\texttt{Endpoint ID} \mapsto \texttt{Location}$$

where `Location` is either `Server` or `Client(ID)`, where `ID` identifies a particular client. The map is updated if: a connection is established using `fork` or an access point; an endpoint is sent as part of a message; or a client disconnects. The server also maintains a map

$$\texttt{Client ID} \mapsto [\texttt{Channel}]$$

associating each client with the publicly-facing channels residing on that client, where `Channel` is a pair of endpoints (a, b) such that b is the endpoint residing on the client. Much like TCP connections, WebSocket connections raise an event when a connection is disconnected. Upon receiving such an event, all channels associated with the client are cancelled, and exceptions are invoked as per the exception handling mechanism described further in §2 and §5.5.

Session Delegation. It is possible to send endpoints as part of a message. Session delegation in the presence of distributed communication has intricacies in ensuring that messages are delivered to the correct participant; our implementation adapts the algorithms described by Hu et al. [2008]. Further details can be found in §B of the supplementary material.

5.5 Session Typing with Failure Handling

Handlers for Algebraic Effects. Algebraic effects [Plotkin and Power 2001] and their handlers [Plotkin and Pretnar 2013] are a modular abstraction for programming with user-defined effects. Exception handlers are in fact a special case of handlers for algebraic effects. Consequently, we leverage the existing implementation of effect handlers in Links [Hillerström and Lindley 2016; Hillerström et al. 2017]. In §4 we generalise **try**−**as**−**in**−**otherwise**− to accommodate user defined exceptions. Effect handlers generalise further to support what amounts to *resumable exceptions* in which the handler not only has access to a payload, but also to the delimited continuation (i.e. evaluation context) from the point at which the exception was raised up to the handler, allowing effect handlers to implement arbitrary side-effects; not just exceptions.

Adopting the syntax of Hillerström and Lindley's $\lambda^{\rho}_{\text{eff}}$ calculus [Hillerström and Lindley 2016], we translate exception handling as follows:

$$\begin{aligned}
&[\![\textbf{raise}]\!] = \textbf{do}\ \text{raise}\\
&[\![\textbf{try}\ L\ \textbf{as}\ x\ \textbf{in}\ M\ \textbf{otherwise}\ N]\!] = \textbf{handle}\ [\![L]\!]\ \textbf{with}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{return}\ x \mapsto [\![M]\!]\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{raise}\ r \quad \mapsto \textbf{cancel}\ r; [\![N]\!]
\end{aligned}$$

The introduction form **do** op invokes an operation op (which may represent raising an exception or some other effect). The elimination form **handle** M **with** H runs effect handler H on the computation M. In general an effect handler H consists of a *return clause* of the form **return** $x \mapsto N$, which behaves just like the success continuation (x **in** N) of an exception handler, and a collection of *operation clauses*, each of the form op $\vec{p}\ r \mapsto N$. Each clause specifies how to handle each operation analogously to how exception handler clauses specify how to handle each exception, except that as

well as binding payload parameters $\vec{p}$, each operation clause also binds a *resumption* parameter r. The resumption r binds a closure that reifies the continuation up to the nearest enclosing effect handler, allowing control to pass back to the program after handling the effect. In the case of our translation, the fail operation has no payload, and rather than invoking the resumption r we cancel it, assuming the natural extension of cancellation to arbitrary linear values, whereby all free names in the value are cancelled (r being bound to the current evaluation context reified as a value).

As a preprocessing step, before translating to effect handlers, we insert a dummy exception handler around each forked thread

$$(\!|\mathbf{fork}\ M|\!)^\star = \mathbf{fork}\ (\mathbf{try}\ (\!|M|\!)^\star\ \mathbf{as}\ x\ \mathbf{in}\ ()\ \mathbf{otherwise}\ ())$$

which has the effect of simulating the E-RaiseChild rule, ensuring that unhandled exceptions are trapped and all endpoints in the context are cancelled if an exception is raised.

As we are targeting *linear* effect handlers, the sharing of linear variables between the success and failure continuations of an exception handler is problematic since there is no reason, *a priori*, to assume that operations should not be handled more than once. The issue can be resolved by restricting the typing rule for **try** in order to disallow any free variables in the continuations:

$$\text{T-TryRestricted}\quad \frac{\Gamma \vdash L : A \qquad x : A \vdash M : B \qquad \cdot \vdash N : B}{\Gamma \vdash \mathbf{try}\ L\ \mathbf{as}\ x\ \mathbf{in}\ M\ \mathbf{otherwise}\ N : B}$$

This rule may look overly restrictive, but in fact it still allows us to simulate the unrestricted rule via a simple macro translation using a Maybe type:

$$\begin{aligned}(\!|\mathbf{try}\ L\ \mathbf{as}\ x\ \mathbf{in}\ M\ \mathbf{otherwise}\ N|\!)^\dagger = \mathbf{case}\ \mathbf{try}\ (\!|L|\!)^\dagger\ \mathbf{as}\ x\ \mathbf{in}\ \mathsf{Some}\ x\ \mathbf{otherwise}\ \mathsf{None}\ \mathbf{of}\\ \mathsf{Some}\ x \mapsto (\!|M|\!)^\dagger\\ \mathsf{None} \mapsto (\!|N|\!)^\dagger\end{aligned}$$

Links performs this translation as another preprocessing step.

Raising Exceptions. An exception may be raised either explicitly through an invocation of **raise** (desugared to **do** raise), or through a blocked **receive** call where the partner endpoint has been cancelled. Thus, we know statically where any exceptions may be raised.

In order to support cancellation of closures on the client, we adorn closures with an explicit environment field that can be directly inspected. Currently, Links does not closure convert continuations on the client, so we use a workaround in order to simulate cancelling a resumption (as required by the translation $[\![-]\!]$). When compiling client code, for each occurrence of **do** raise, we compile a function that inspects all affected variables and cancels any affected endpoints in the continuation. For each occurrence of **receive**, we compile a continuation to cancel affected endpoints to be invoked by the runtime system if the receive operation fails.

Distributed Exceptions. We require surprisingly little additional machinery in order to provide distributed exceptions and provide an answer to the question: "Well, what happens if a client closes the browser window?". We maintain a mapping from client IDs to the list of channels contained on that client and we add a special message type to notify a client that the peer endpoint of a channel it owns has been cancelled. WebSockets—much like TCP sockets—raise a *closed* event when they are closed. Consequently, when a channel is closed, we look up the channels owned by the terminated client and notify all clients containing the peer endpoints of the cancelled channels.

```
typename ChatClient = !Nickname.
  [&| Join:
        ?(Topic, [Nickname], ClientReceive).ClientSend,
      Nope:End |&];

typename ClientReceive =
  [&| Join     : ?Nickname           .ClientReceive,
      Chat     : ?(Nickname, Message).ClientReceive,
      NewTopic : ?Topic              .ClientReceive,
      Leave    : ?Nickname           .ClientReceive
  |&];

typename ClientSend =
  [+| Chat  : ?Message.ClientSend,
      Topic : ?Topic  .ClientSend |+];

typename ChatServer = ~ChatClient;
typename WorkerSend = ~ClientReceive;
typename WorkerReceive = ~ClientSend;
```

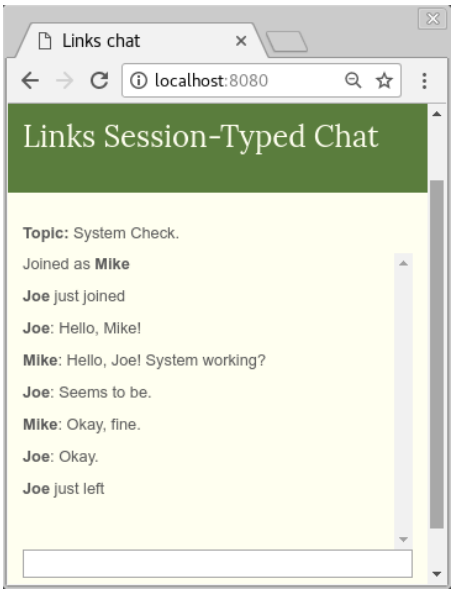


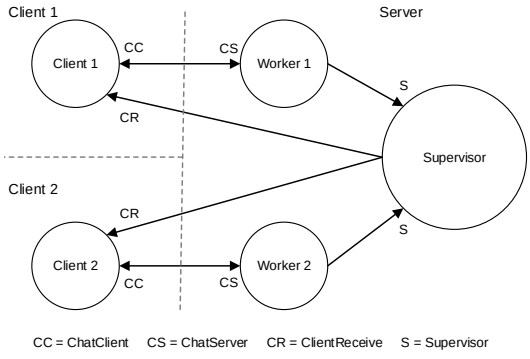
Fig. 10. Chat Application Session Types

6 EXAMPLE: A CHAT APPLICATION

In this section we outline the design and implementation of a web-based chat application in Links making use of distributed session-typed channels. Informally, we write the following specification:

- To initialise, a client must:
  - connect to the chat server; then
  - send a nickname; then
  - receive the current topic and list of nicknames.
- After initialisation the client is connected and can:
  - send a chat message to the room; or
  - change the room's topic; or
  - receive messages from other users; or
  - receive changes of topic from other users.
- Clients cannot connect with a nickname that is already in use in the room.
- All participants should be notified whenever a participant joins or leaves the room.

Session Types. We can encode much of the specification more precisely as a session type, as shown in Figure 10. The client begins by sending a nickname, and then offers the server a choice of a `Join` message or a `Nope` message. In the former case, the client then receives a triple containing the current topic, a list of existing nicknames, and an endpoint (of type `ClientReceive`) for receiving further updates from the server; and may then continue to send messages to the server as a connected client endpoint (of type `ClientSend`). (Observe the essential use of session delegation.) In the latter case, communication is terminated. The intention is that the server will respond with `Nope` if a client with the supplied nickname is already in the chat room (the details of this check are part of the implementation, not part of the communication protocol).



(a) Architecture

```
sig worker : (Nickname, WorkerReceive) ~> ()
fun worker(nick, c) {
  try {
    offer(c) {
      case Chat(c) ->
        var (msg, c) = receive(c);
        chat(nick, msg);
        c
      case NewTopic(c) ->
        var (topic, c) = receive(c);
        newTopic(topic);
        c
    }
  } as (c) in {
    worker(nick, c)
  } otherwise {
    leave(nick)
  }
}
```

(b) Worker Implementation

Fig. 11. Chat Application Architecture and Worker Implementation

The `ClientReceive` endpoint allows the client to offer a choice of four different messages: `Join`, `Chat`, `NewTopic`, or `Leave`. In each case the client then receives a payload (depending on the choice, a nickname, pair of nickname and chat message, or topic change) before offering another choice. The `ClientSend` endpoint allows the client to select between two different messages: `Chat` and `NewTopic`. In each case the client subsequently sends a payload (a chat message or a new topic) before selecting another choice. The chat server communicates with the client along endpoints with dual types.

Architecture. Figure 11a depicts the architecture of the chat server application. Each client has a process which sends messages over a distributed session channel of type `ClientSend` to its own worker process on the server, which in turn sends internal messages to a supervisor process containing the state of the chat room. In turn, these messages trigger the supervisor process to broadcast a message to all chat clients over a channel of type `~ClientReceive`. As is evident from the figure, the communication topology is cyclic; in order to construct this topology the code (see the supplementary material) makes essential use of access points.

Disconnection. Figure 11b shows the implementation of a worker process which receive messages from a client. The worker takes the nickname of the client, as well as a channel endpoint of type `WorkerReceive` (which is the dual of `ClientSend`). The server offers the client a choice of sending a message (`Chat`), or changing topic (`NewTopic`); in each case, the associated data is received and an appropriate message dispatched to the supervisor process by calling `chat` or `newTopic`. The client may leave the chat room at any time by closing the browser window. All other participants are notified when a participant joins or leaves. When a client closes its connection to the server, all associated endpoints are cancelled. Consequently, an exception will be raised when evaluating the `offer` or `receive` expressions where the user has closed their browser window. To handle disconnection, we wrap the function in an exception handler, which recursively calls `worker` if the interaction is successful, and notifies the server that the user has left via a call to `leave` if an exception is raised.

7 RELATED WORK

7.1 Session Types with Failure Handling

Carbone et al. [2008] provide the first formal basis for exceptions in a session-typed process calculus. Our approach provides significant simplifications: zapper threads provide a simpler semantics and remove the need for their queue levels, meta-reduction relation, and liveness protocol.

Our work draws on that of Mostrous and Vasconcelos [2014], who introduce the idea of cancellation. Our work differs from theirs in several key ways. Their system is a process calculus; ours is a λ-calculus. Their channels are synchronous; ours are asynchronous. Their exception handling construct scopes over a single action; ours scopes over an arbitrary computation.

Caires and Pérez [2017] describe a core, logically-inspired process calculus supporting nondeterminism and abortable behaviours encoded via a nondeterminism modality. Processes may either provide or not provide a prescribed behaviour; if a process attempts to consume a behaviour that is not provided, then its linear continuation is safely discarded by propagating the failure of sessions contained within the continuation. Their approach is similar in spirit to our zapper threads. Additionally, they give a core λ-calculus with abortable behaviours and exception handling, and define a type-preserving translation into their core process calculus.

Our approach differs in several important ways. First, our semantics is asynchronous, handling the intricacies involved with cancelling values contained in message queues. Second, we give a direct semantics to EGV, whereas Caires and Pérez rely on a translation into their underlying process calculus. Third, to handle the possibility of disconnection, our calculus allows *any* channel to be discarded (including the ability to handle uncaught exceptions), whereas the authors opt for an approach more closely resembling checked exceptions, aided by a monadic presentation.

These works are all theoretical; backed by our theoretical development, our implementation integrates session types and exceptions, extending Links.

Multiparty Session Types. Fowler [2016] describes an Erlang implementation of the Multiparty Session Actor framework proposed by Neykova and Yoshida [2017b] with a limited form of failure recovery; Neykova and Yoshida [2017a] present a more comprehensive approach, based on refining existing Erlang supervision strategies. Chen et al. [2016] introduce a formalism based on multiparty session types [Honda et al. 2016] that handles partial failures by transforming programs to detect possible failures at a set of statically determined synchronisation points. These approaches rely on a fixed communication topology, using mechanisms such as dependency graphs or synchronisation points to determine which participants are affected when one participant fails. For binary channels, delegation implies location transparency, thus we must consider dynamic topologies.

7.2 Session Types and Distribution

Hu et al. [2008] introduce Session Java (SJ), which allows distributed session-based communication in the Java programming language, making use of the Polyglot framework [Nystrom et al. 2003] to statically check session types. Hu et al. are the first to present the challenges of distributed delegation along with distributed algorithms which address those challenges. We adapt their algorithms to web applications. SJ restricts communication to a fixed set of simple types; Links allows arbitrary values to be sent. SJ provides statically scoped exception handling, propagating exceptions to ensure liveness (but this feature is not formalised).

Scalas and Yoshida [2016] introduce `lchannels`, a lightweight implementation of session types in Scala. To maximise applicability of their approach and not require any modifications to Scala, their approach makes use of affine types, with duplicate outputs detected at runtime. By virtue of the translation into the linear π-calculus introduced by Kobayashi [2003] and later expanded on by Dardha et al. [2017], `lchannels` is particularly amenable to distribution. Scalas et al. [2017]

build upon this approach to translate a multiparty session calculus into the linear π-calculus to provide the first distributed implementation of multiparty session types to support delegation.

7.3 Session Types via Affine Types

Rust [Matsakis and Klock II 2014] provides *ownership types* [Clarke 2003], ensuring that an object has at most one owner. Jespersen et al. [2015] use Rust's ownership types to encode affine session types. It would be interesting to explore whether our exception handling methodology can be applied to ensure progress in the Rust implementation using Rust's destructor mechanism.

8 CONCLUSION AND FUTURE WORK

Session types allow conformance to a protocol to be checked statically. Designing languages with session types requires a substructural type system, and the prevailing consensus has hitherto been to require linear use of endpoints to enforce session fidelity and to prevent premature discarding of open channels.

We have argued that in order to write realistic applications in the presence of distribution and failure, linearity should be supplemented with an *explicit* cancellation operation. We show that, even in the presence of channel cancellation, our core calculus is well-behaved, being deadlock-free, type sound, confluent, and terminating.

In tandem with the formal development, we have developed a practical extension of the Links web programming language to support distributed session-based communication for web applications, thus providing the first implementation of asynchronous session types and exceptions in a functional programming language. Our implementation takes advantage of recent work on handlers for algebraic effects, paving the way for further study of linear effect handlers.

8.1 Future Work

Linear Effect Handlers. Our implementation combines linearity and effect handlers. Linear effect handlers are new, and a ripe area of study in their own right; in the near future we aim to formalise session-typed concurrency directly in terms of linear effect handlers.

Resources. We have discussed explicit cancellation and exception handling for session types. It would be interesting to investigate whether the approach scales to the more general case of exception/effect handling when using substructural types to enforce protocols in the presence of resources (e.g. files).

Multiparty Session Types. The theory of multiparty session types [Honda et al. 2016] has led to a multitude of implementations, but multiparty session types are yet to be incorporated as *first-class* constructs in a core functional language. A natural starting point would be a λ-calculus into which we can translate the MCP calculus of Carbone et al. [2016], which is based on classical linear logic.

REFERENCES

H. P. Barendregt. 1984. *The Lambda Calculus Its Syntax and Semantics* (revised ed.). Vol. 103. North Holland.

Nick Benton and Andrew Kennedy. 2001. Exceptional Syntax. *Journal of Functional Programming* 11, 4 (2001), 395–410.

Adam Bergkvist, Daniel C Burnett, Cullen Jennings, Anant Narayanan, and Bernard Aboba. 2012. WebRTC 1.0: Real-time Communication Between Browsers. (2012).

Luís Caires and Jorge A Pérez. 2017. Linearity, control effects, and behavioral types. In *ESOP*. Springer, 229–259.

Marco Carbone, Ornela Dardha, and Fabrizio Montesi. 2014. Progress as compositional lock-freedom. In *COORDINATION*. Springer, 49–64.

Marco Carbone, Kohei Honda, and Nobuko Yoshida. 2008. Structured interactional exceptions in session types. In *CONCUR*. Springer, 402–417.

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

Tzu-Chun Chen, Malte Viering, Andi Bejleri, Lukasz Ziarek, and Patrick Eugster. 2016. A Type Theory for Robust Failure Handling in Distributed Systems. In *FORTE (Lecture Notes in Computer Science)*, Vol. 9688. Springer, 96–113.

David Gerard Clarke. 2003. *Object Ownership and Containment*. Ph.D. Dissertation. New South Wales, Australia. AAI0806678.

Ezra Cooper. 2009. *Programming Language Features for Web Application Development*. Ph.D. Dissertation. University of Edinburgh.

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2007. Links: Web programming without tiers. In *FMCO*. Springer, 266–296.

Ornela Dardha, Elena Giachino, and Davide Sangiorgi. 2017. Session types revisited. *Inf. Comput.* 256 (2017), 253–286.

Ian Fette and Alexey Melnikov. 2011. *The WebSocket Protocol*. RFC 6455. RFC Editor. 70 pages. http://www.rfc-editor.org/rfc/rfc6455.txt

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

Simon J Gay and Vasco T Vasconcelos. 2010. Linear type theory for asynchronous session types. *Journal of Functional Programming* 20, 1 (2010), 19–50.

Daniel Hillerström and Sam Lindley. 2016. Liberating Effects with Rows and Handlers. In *TyDe@ICFP*. ACM, 15–27.

Daniel Hillerström, Sam Lindley, Robert Atkey, and K. C. Sivaramakrishnan. 2017. Continuation Passing Style for Effect Handlers. In *FSCD (LIPIcs)*, Vol. 84. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 18:1–18:19.

Kohei Honda. 1993. Types for dyadic interaction. In *CONCUR*. Springer, 509–523.

Kohei Honda, Vasco T Vasconcelos, and Makoto Kubo. 1998. Language primitives and type discipline for structured communication-based programming. In *ESOP*. Springer, 122–138.

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2016. Multiparty asynchronous session types. *Journal of the ACM (JACM)* 63, 1 (2016), 9.

Raymond Hu, Nobuko Yoshida, and Kohei Honda. 2008. Session-based distributed programming in Java. In *ECOOP*. Springer, 516–541.

Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. 2015. Session types for Rust. In *WGP*. ACM, 13–22.

Naoki Kobayashi. 2003. *Type Systems for Concurrent Programs*. Springer Berlin Heidelberg, Berlin, Heidelberg, 439–453.

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

Sam Lindley and J Garrett Morris. 2016. Talking bananas: structural recursion for session types. In *ICFP*. ACM, 434–447.

Sam Lindley and J Garrett Morris. 2017. Lightweight Functional Session Types. In *Behavioural Types: from Theory to Tools*. River Publishers, 265–286.

Nicholas D. Matsakis and Felix S. Klock II. 2014. The Rust Language. In *HILT*. ACM, 103–104.

Karl Mazurak, Jianzhou Zhao, and Steve Zdancewic. 2010. Lightweight Linear Types in System F°. In *TLDI*. ACM, 77–88.

Robin Milner. 1999. *Communicating and mobile systems: the pi calculus*. Cambridge university press.

Dimitris Mostrous and Vasco Thudichum Vasconcelos. 2014. Affine Sessions. In *COORDINATION*. Springer, 115–130.

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

Rumyana Neykova and Nobuko Yoshida. 2017b. Multiparty Session Actors. *Logical Methods in Computer Science* 13, 1 (2017).

Nathaniel Nystrom, Michael Clarkson, and Andrew Myers. 2003. Polyglot: An extensible compiler framework for Java. In *CC*. Springer, 138–152.

Luca Padovani. 2017. A simple library implementation of binary sessions. *Journal of Functional Programming* 27 (2017), e4.

Jorge A Pérez, Luís Caires, Frank Pfenning, and Bernardo Toninho. 2012. Linear logical relations for session-based concurrency. In *European Symposium on Programming*. Springer, 539–558.

Gordon Plotkin and John Power. 2001. Adequacy for algebraic effects. In *FoSSaCS*. Springer, 1–24.

Gordon D. Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science* 9, 4 (2013).

Didier Rémy. 1994. Type inference for records in a natural extension of ML. In *Theoretical Aspects Of Object-Oriented Programming*, Carl A. Gunter and John C. Mitchell (Eds.). MIT Press, Cambridge, MA, 67–95.

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

Alceste Scalas and Nobuko Yoshida. 2016. Lightweight Session Programming in Scala. In *ECOOP (LIPIcs)*, Vol. 56. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 21:1–21:28.

Jérôme Vouillon. 2008. Lwt: a cooperative thread library. In *ML*. ACM, 3–12.

Philip Wadler. 2014. Propositions as sessions. *Journal of Functional Programming* 24, 2-3 (2014), 384–418.

A SELECTED FULL PROOFS

A.1 Preservation

Firstly, typing of terms is preserved by substitution.

Lemma A.1 (Substitution). *If* $\Gamma_1 \vdash M : B$ *and* $\Gamma_2, x : B \vdash N : A$, *then* $\Gamma_1, \Gamma_2 \vdash N\{M/x\} : A$.

Proof. By induction on the derivation of $\Gamma_2, x : B \vdash N : A$. □

Lemma A.2 shows that a subterm of a well-typed evaluation context H (and therefore also a pure evaluation context E) is typeable with a subset of the type environment. Lemma A.3 states that the subterm of a well-typed evaluation context can be replaced. Both follow the formulation of Gay and Vasconcelos [2010].

Lemma A.2 (Typeability of subterms). *If* $\boldsymbol{D}$ *is a derivation of* $\Gamma \vdash E[M] : A$, *then there exist* Γ_1, Γ_2 *and* B *such that* $\Gamma = \Gamma_1, \Gamma_2$, *that* $\boldsymbol{D}$ *has a subderivation* $\boldsymbol{D}'$ *that concludes* $\Gamma_2 \vdash M : B$, *and the position of* $\boldsymbol{D}'$ *in* $\boldsymbol{D}$ *corresponds to the position of the hole in* E.

Proof. By induction on the structure of E. □

Lemma A.3 (Replacement (evaluation contexts)). *If:*

- $\boldsymbol{D}$ *is a derivation of* $\Gamma_1, \Gamma_2 \vdash E[M] : A$
- $\boldsymbol{D}'$ *is a subderivation of* $\boldsymbol{D}$ *concluding* $\Gamma_2 \vdash M : B$
- *The position of* $\boldsymbol{D}'$ *in* $\boldsymbol{D}$ *corresponds to that of the hole in* E
- $\Gamma_3 \vdash N : B$

then $\Gamma_1, \Gamma_3 \vdash E[N] : B$.

Proof. By induction on the structure of E. □

To prove preservation on configurations, we must first establish some auxiliary results on configuration contexts. Lemma A.4 states how we may type subconfigurations.

Lemma A.4 (Typeability of subconfigurations). *If* $\boldsymbol{D}$ *is a derivation of* $\Gamma; \Delta \vdash^{\phi} \mathcal{G}[C]$, *then there exist* Γ', Δ', ϕ' *such that* $\boldsymbol{D}$ *has a subderivation* $\boldsymbol{D}'$ *that concludes* $\Gamma'; \Delta' \vdash^{\phi'} C$, *and the position of* $\boldsymbol{D}'$ *in* $\boldsymbol{D}$ *corresponds to the position of the hole in* $\mathcal{G}$.

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

Lemma A.5 states that we may replace a subconfiguration of a configuration context. The lemma is slightly complicated by the fact that $(\nu a)\mathcal{G}$ binds a variable a, but replacement is safe if the typing environments are related by the environment reduction relation.

Lemma A.5 (Replacement (configurations)). *If:*

- $\boldsymbol{D}$ *is a derivation of* $\Gamma; \Delta \vdash^{\phi} \mathcal{G}[C]$
- $\boldsymbol{D}'$ *is a subderivation of* $\boldsymbol{D}$ *concluding that* $\Gamma'; \Delta' \vdash^{\phi'} C$ *for some* Γ', Δ', ϕ'
- $\Gamma''; \Delta'' \vdash^{\phi'} C'$ *for some* Γ'', Δ'' *such that* $\Gamma'; \Delta' \longrightarrow^{?} \Gamma''; \Delta''$
- *The position of* $\boldsymbol{D}$ *in* $\boldsymbol{D}'$ *corresponds to that of the hole in* $\mathcal{G}$

then there exist some Γ''', Δ''' *such that* $\Gamma'''; \Delta''' \vdash^{\phi} \mathcal{G}[C']$ *and* $\Gamma; \Delta \longrightarrow^{?} \Gamma'''; \Delta'''$.

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

Theorem 3.2 (Preservation (Configurations))
Assume Γ *only contains entries of the form* $a_i : S_i$.
If $\Gamma; \Delta \vdash^{\phi} C$ *and* $C \longrightarrow C'$, *then there exist* Γ', Δ' *such that* $\Gamma; \Delta \longrightarrow^{?} \Gamma'; \Delta'$ *and* $\Gamma'; \Delta' \vdash^{\phi} C'$.

Proof. We proceed by induction on the derivation of $C \longrightarrow C'$. If there is a choice on the value of ϕ, (for example, if the configuration contains a thread), we prove the cases where $\phi = \bullet$. The cases where $\phi = \circ$ are similar (but use T-Thread instead of T-Main in the inversion and construction steps for that thread).

Case E-Fork

$$\mathcal{F}[\mathbf{fork}\,\lambda x.M] \longrightarrow (va)(vb)(\mathcal{F}[a] \parallel M\{b/x\} \parallel a(\epsilon) \leftrightsquigarrow b(\epsilon))$$

Assumption: $\Gamma; \Delta \vdash^{\bullet} \mathcal{F}[\mathbf{fork}\,\lambda x.M]$
By definition of $\mathcal{F}$: $\exists E.\mathcal{F} = \bullet(E[\mathbf{fork}\,\lambda x.M])$
By T-Main:

- $\Delta = \cdot$
- $\Gamma \vdash E[\mathbf{fork}\,\lambda x.M] : A$

Let $\mathbf{D}$ be the derivation of $\Gamma \vdash^{\bullet} E[\mathbf{fork}\,\lambda x.M] : A$.
By Lemma A.2:

- $\exists \Gamma_1, \Gamma_2.\Gamma = \Gamma_1, \Gamma_2$
- $\exists \mathbf{D}'$ such that $\mathbf{D}'$ is a subderivation of $\mathbf{D}$ concluding $\Gamma_2 \vdash \mathbf{fork}\,\lambda x.M : B$
- The position of $\mathbf{D}'$ in $\mathbf{D}$ corresponds to the position of the hole in E.

By inversion on T-Fork:

- $\Gamma_2 \vdash \mathbf{fork}\,\lambda x.M : \overline{S}$
- $\Gamma_2 \vdash \lambda x.M : S \multimap \mathsf{End}$

By Lemma A.3:

- $\Gamma_1, a : \overline{S} \vdash E[a] : A$

By inversion on T-Abs:

- $\Gamma_2, x : S \vdash M : \mathbf{1}$

By Lemma A.1:

- $\Gamma_2, b : S \vdash M\{b/x\} : \mathbf{1}$

By T-Thread:

- $\Gamma_2, b : S \vdash \circ M\{b/x\}$

By T-Buffer:

- $\cdot; a : S, b : \overline{S} \vdash^{\circ} a(\epsilon) \leftrightsquigarrow b(\epsilon)$

By T-Connect$_1$:

- $\Gamma_2; a : S, b : S^{\sharp} \vdash^{\circ} M\{b/x\} \parallel a(\epsilon) \leftrightsquigarrow b(\epsilon)$

By T-Connect$_1$:

- $\Gamma_1, \Gamma_2; a : \overline{S}^{\sharp}, b : S^{\sharp} \vdash^{\bullet} E[a] \parallel \circ M\{b/x\} \parallel a(\epsilon) \leftrightsquigarrow b(\epsilon)$

By T-Nu:

- $\Gamma_1, \Gamma_2; a : \overline{S}^{\sharp} \vdash^{\bullet} (vb)(E[a] \parallel \circ M\{b/x\} \parallel a(\epsilon) \leftrightsquigarrow b(\epsilon))$

By T-Nu:

- $\Gamma_1, \Gamma_2; \cdot \vdash^{\bullet} (va)(vb)(E[a] \parallel \circ M\{b/x\} \parallel a(\epsilon) \leftrightsquigarrow b(\epsilon))$

Since $\Gamma = \Gamma_1, \Gamma_2$ and $\Delta = \cdot$:

- $\Gamma; \cdot \vdash^{\bullet} (va)(vb)(E[a] \parallel \circ M\{b/x\} \parallel a(\epsilon) \leftrightsquigarrow b(\epsilon))$

as required.

Case E-Send

$$\mathcal{F}[\mathbf{send}\,V'\,a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W}) \longrightarrow \mathcal{F}[a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W} \cdot V')$$

Assumption: $\Gamma; \Delta \vdash^{\bullet} \mathcal{F}[\mathbf{send}\ V'\ a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$
By inversion on T-Connect$_1$:

- $\exists \Gamma_1, \Gamma_2 . \Gamma = \Gamma_1, \Gamma_2$
- $\exists \Delta_1, \Delta_2 . \Delta = \Delta_1, \Delta_2, a : S^\sharp$
- $\Gamma_1, a : S; \Delta_1 \vdash^{\bullet} \mathcal{F}[\mathbf{send}\ V'\ a]$
- $\Gamma_2; \Delta_2, a : \overline{S} \vdash^{\circ} a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$

By the definition of $\mathcal{F}$ and knowledge that $\phi = \bullet$:

- $\exists E . \mathcal{F} = \bullet(E[\mathbf{send}\ V'\ a])$

By T-Main:

- $\Delta_1 = \cdot$
- $\Gamma_1, a : S \vdash E[\mathbf{send}\ V'\ a] : C$

Let $\mathbf{D}$ be the derivation of $\Gamma_1, a : S \vdash E[\mathbf{send}\ V'\ a] : C$.
By Lemma A.2:

- $\exists \Gamma_3, \Gamma_4 . \Gamma_1, a : S = \Gamma_3, \Gamma_4, a : S$
- $\exists \mathbf{D}'$ such that $\mathbf{D}'$ is a subderivation of $\mathbf{D}$ concluding that $\Gamma_4, a : S \vdash \mathbf{send}\ V'\ a : B$
- The position of $\mathbf{D}'$ in $\mathbf{D}$ corresponds to the position of the hole in E.

By inversion on T-Send:

- $B = S'$
- $S = !A.S'$
- $\Gamma_4 \vdash V' : A$

By knowledge that $S = !A.S'$:

- $\Gamma_2; \Delta_2, a : \overline{!A.S'} \vdash^{\circ} a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$

By the definition of duality:

- $\Gamma_2; \Delta_2, a : ?A.\overline{S'} \vdash^{\circ} a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$

By inversion on T-Buffer:

- $\exists \Gamma_5, \Gamma_6 . \Gamma_2 = \Gamma_5, \Gamma_6$
- $\Delta_2 = b : T$
- $\Gamma_5 \vdash \overrightarrow{V} : \overrightarrow{A}$
- $\Gamma_6 \vdash \overrightarrow{W} : \overrightarrow{B}$
- $?A.\overline{S'} / \overrightarrow{A} = \overline{T / \overrightarrow{B}}$

By the definition of the slicing function:

- $\overline{T / \overrightarrow{B}} = \overline{!B_1. \cdots .!B_n.!A.S}$ for each $B_i \in \overrightarrow{B}$.

It follows that:

- $\overline{S'} / \overrightarrow{A} = \overline{T / \overrightarrow{B} \cdot A}$

By the definition of $\Gamma \vdash \overrightarrow{V} : \overrightarrow{A}$:

- $\Gamma_4, \Gamma_6 \vdash \overrightarrow{W} \cdot V' : \overrightarrow{B} \cdot A$

Thus by T-Buffer:

- $\Gamma_2, \Gamma_4; a : \overline{S'}, b : T \vdash^{\circ} a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W} \cdot V')$

By Lemma A.3:

- $\Gamma_3, a : S' \vdash E[a] : C$

By T-Main:

- $\Gamma_3, a : S'; \cdot \vdash^{\bullet} \bullet(E[a])$

By the definition of $\mathcal{F}$:

- $\mathcal{F}[a] = \bullet E[a]$

So

- $\Gamma_3, a : S'; \cdot \vdash^\bullet \mathcal{F}[a]$

By T-Connect$_1$:

- $\Gamma_2, \Gamma_3, \Gamma_4; a : S'^\sharp, b : T \vdash^\bullet (\mathcal{F}[a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W} \cdot V'))$

We have that $\Gamma = \Gamma_2, \Gamma_3, \Gamma_4$, and that $\Delta = a : S^\sharp, b : T$. Since $S = !A.S'$ and $(!A.S')^\sharp \longrightarrow S'^\sharp$, we have that $\Gamma; \Delta \longrightarrow \Gamma; a : S'^\sharp; b : T$, with

- $\Gamma; a : S'^\sharp, b : T \vdash^\bullet \mathcal{F}[a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W} \cdot V')$

as required.

Case E-Receive

$\mathcal{F}[\textbf{receive}\ a] \parallel a(V' \cdot \overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W}) \longrightarrow$
$\mathcal{F}[(V', a)] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$

Assumption: $\Gamma; \Delta \vdash^\bullet \mathcal{F}[\textbf{receive}\ a] \parallel a(V' \cdot \overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$

By inversion on T-Connect$_1$:

- $\exists \Gamma_1, \Gamma_2. \Gamma = \Gamma_1, \Gamma_2$
- $\exists \Delta_1, \Delta_2. \Delta = \Delta_1, \Delta_2, a : S^\sharp$
- $\Gamma_1, a : S; \Delta_1 \vdash^\bullet \mathcal{F}[\textbf{receive}\ a]$
- $\Gamma_2; \Delta_2, a : \overline{S} \vdash^\circ a(V' \cdot \overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$

By the definition of F:

- $\exists E.F = \bullet(E[\textbf{receive}\ a])$

By T-Main:

- $\Delta_1 = \cdot$
- $\Gamma_1, a : S \vdash E[\textbf{receive}\ a] : C$

Let $\mathbf{D}$ be the derivation of $\Gamma_1, a : S \vdash E[\textbf{receive}\ a] : C$.

By Lemma A.2:

- $\exists \Gamma_3, \Gamma_4. \Gamma_1, a : S = \Gamma_3, \Gamma_4, a : S$
- $\exists \mathbf{D'}$ such that $\mathbf{D'}$ is a subderivation of $\mathbf{D}$ concluding $\Gamma_4, a : S \vdash \textbf{receive}\ a : B$
- The position of $\mathbf{D'}$ in $\mathbf{D}$ corresponds to the position of the hole in E.

By inversion on T-Recv:

- $\Gamma_4 = \cdot$
- $S = ?A.S'$
- $B = (A \times S')$

Since $S = ?A.S'$, by duality we have that:

- $\Gamma_2; \Delta_2, a : !A.\overline{S'} \vdash^\circ a(V' \cdot \overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$

By inversion on T-Buffer:

- $\exists \Gamma_5, \Gamma_6. \Gamma_2 = \Gamma_5, \Gamma_6$
- $\Delta_2 = b : T$
- $\Gamma_5 \vdash V' \cdot \overrightarrow{V} : A \cdot \overrightarrow{A}$
- $\Gamma_6 \vdash \overrightarrow{W} : \overrightarrow{B}$
- $!A.\overline{S'}/A \cdot \overrightarrow{A} = \overline{T/\overrightarrow{B}}$

By the definition of $\Gamma_5 \vdash V' \cdot \overrightarrow{V} : A \cdot \overrightarrow{A}$:

- $\exists \Gamma_7, \Gamma_8. \Gamma_5 = \Gamma_7, \Gamma_8$
- $\Gamma_7 \vdash V' : A$

- $\Gamma_8 \vdash \overrightarrow{V} : \overrightarrow{A}$

By Lemma A.3:

- $\Gamma_3, \Gamma_7, a : S' \vdash E[(V', a)] : C$

By T-Main:

- $\Gamma_3, \Gamma_7, a : S'; \cdot \vdash^\bullet E[(V', a)] : C$

By the definition of the slicing function:

- $\overline{S'}/\overrightarrow{A} = \overline{T/\overrightarrow{B}}$

By T-Buffer:

- $\Gamma_6, \Gamma_8; a : \overline{S'}, b : T \vdash^\circ a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$

By T-Connect$_1$:

- $\Gamma_3, \Gamma_6, \Gamma_7, \Gamma_8; a : S'^\sharp, b : T \vdash^\bullet E[(V', a)] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$

We know that $\Gamma = \Gamma_3, \Gamma_6, \Gamma_7, \Gamma_8$ and $\Delta = a : S^\sharp, b : T$ and that $S = ?A.S'$. Since $?A.S' \longrightarrow S'$, we have that $(?A.S')^\sharp \longrightarrow S'^\sharp$. Thus:

- $\Gamma; a : S, b : T \longrightarrow \Gamma; a : S', b : T$

Therefore:

- $\Gamma; a : S', b : T \vdash^\bullet E[(V', a)] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$

as required.

Case E-Close

$$(\nu a)(\nu b)(\mathcal{F}[\mathbf{close}\ a] \parallel \mathcal{F}'[\mathbf{close}\ b] \parallel a(\epsilon) \leftrightsquigarrow b(\epsilon)) \longrightarrow \mathcal{F}[()] \parallel \mathcal{F}'[()]$$

Assumption: $\Gamma; \Delta \vdash^\bullet (\nu a)(\nu b)(\mathcal{F}[\mathbf{close}\ a] \parallel \mathcal{F}'[\mathbf{close}\ b] \parallel a(\epsilon) \leftrightsquigarrow b(\epsilon))$.

For the sake of the proof, we assume that $\mathcal{F}[\mathbf{close}\ a]$ is a main thread, and $\mathcal{F}'[\mathbf{close}\ b]$ is a child thread. Proving the other combinations ($\mathcal{F}'[\mathbf{close}\ b]$ being the main thread, or both being child threads) follows exactly the same pattern.

By inversion on T-Nu, twice:

- $\Gamma; \Delta, a : S^\sharp, b : T^\sharp \vdash^\bullet \mathcal{F}[\mathbf{close}\ a] \parallel \mathcal{F}'[\mathbf{close}\ b] \parallel a(\epsilon) \leftrightsquigarrow b(\epsilon)$

By inversion on T-Connect$_1$:

- $\exists \Gamma_1, \Gamma_2 . \Gamma = \Gamma_1, \Gamma_2$
- $\exists \Delta_1, \Delta_2 . \Delta = \Delta_1, \Delta_2$
- $\Gamma_1, a : S; \Delta_1 \vdash^\bullet \mathcal{F}[\mathbf{close}\ a]$
- $\Gamma_2; \Delta_2, a : \overline{S} \vdash^\circ \mathcal{F}'[\mathbf{close}\ b] \parallel a(\epsilon) \leftrightsquigarrow b(\epsilon)$

By inversion on T-Connect$_1$ and T-Buffer:

- $\Delta_1 = \cdot$ and $\Delta_2 = \cdot$
- $\Gamma_2, b : T \vdash^\circ \mathcal{F}'[\mathbf{close}\ b]$
- $\cdot; a : \overline{S}, b : \overline{T} \vdash^\circ a(\epsilon) \leftrightsquigarrow b(\epsilon)$

By inversion on T-Main:

- $\Delta_1 = \cdot$
- $\exists E, C . \Gamma_1, a : S \vdash E[\mathbf{close}\ a] : C$

By Lemma A.2 and T-Close:

- $\exists \mathbf{D}'$ such that $\mathbf{D}'$ is a subderivation of $\mathbf{D}$ concluding that $a : S \vdash \mathbf{close}\ a : \mathbf{1}$
- $S = \mathsf{End}$
- The position of $\mathbf{D}'$ in $\mathbf{D}$ corresponds to the position of the hole in E.

By Lemma A.3 and T-Unit:

- $\Gamma_1; \cdot \vdash^\bullet \mathcal{F}[()]$

By inversion on T-Child:

- $\Delta_2 = \cdot$
- $\exists E' \Gamma_2, b : T \vdash E'[\textbf{close}\ b] : \mathbf{1}$

By Lemma A.2 and T-Close:

- $\exists \mathbf{E}'$ such that $\mathbf{E}'$ is a subderivation of $\mathbf{E}$ concluding that $b : T \vdash \textbf{close}\ b : \mathbf{1}$
- $T = \mathsf{End}$
- The position of $\mathbf{E}'$ in $\mathbf{E}$ corresponds to the position of the hole in E.

By Lemma A.3 and T-Unit:

- $\Gamma_2; \cdot \vdash^{\circ} \mathcal{F}'[()]$

By T-Mix, it follows that $\Gamma; \Delta \vdash^{\bullet} \mathcal{F}[()] \parallel \mathcal{F}'[()]$ as required.

Case E-ReceiveZap

$$\mathcal{F}[\textbf{receive}\ a] \parallel \lightning b \parallel a(\epsilon) \leftrightsquigarrow b(\overrightarrow{V}) \longrightarrow \mathcal{F}[\textbf{raise}] \parallel \lightning a \parallel \lightning b \parallel a(\epsilon) \leftrightsquigarrow b(\overrightarrow{V})$$

Assumption: $\Gamma; \Delta \vdash^{\bullet} \mathcal{F}[\textbf{receive}\ a] \parallel a(\epsilon) \leftrightsquigarrow b(\overrightarrow{V})$

By inversion on T-Connect$_1$:

- $\exists \Gamma_1, \Gamma_2 . \Gamma = \Gamma_1, \Gamma_2$
- $\exists \Delta_1, \Delta_2 . \Delta = \Delta_1, \Delta_2, a : S^{\sharp}$
- $\Gamma_1, a : S; \Delta_1 \vdash \mathcal{F}[\textbf{receive}\ a]$
- $\Gamma_2; \Delta_2, a : \overline{S} \vdash \lightning b \parallel a(\epsilon) \leftrightsquigarrow b(\overrightarrow{V})$

By definition, $\exists E.F = \bullet(E[\textbf{receive}\ a])$.

By inversion on T-Main:

- $\Delta_1 = \cdot$

By Lemma A.2:

- $\exists \Gamma_3, \Gamma_4 . \Gamma_1, a : S = \Gamma_3, \Gamma_4, a : S$
- $\exists \mathbf{D}'$ such that $\mathbf{D}'$ is a subderivation of $\mathbf{D}$ concluding that $\Gamma_4, a : S \vdash \textbf{receive}\ a : B'$
- The position of $\mathbf{D}'$ in $\mathbf{D}$ corresponds to the position of the hole in E.

By inversion on T-Recv:

- $\Gamma_4 = \cdot$
- $S = ?A.S'$

By Lemma A.3:

- $\Gamma_3 \vdash E[\textbf{raise}] : B$

By T-Main:

- $\Gamma_3; \cdot \vdash^{\bullet} \mathcal{F}[\textbf{raise}]$

By T-Zap:

- $a : S; \cdot \vdash \lightning a$

By T-Connect$_1$:

- $\Gamma_2; \Delta_2, a : S^{\sharp} \vdash^{\bullet} \lightning a \parallel \lightning b \parallel a(\epsilon) \leftrightsquigarrow b(\overrightarrow{V})$

By T-Mix:

- $\Gamma; \Delta \vdash^{\bullet} \mathcal{F}[\textbf{raise}] \parallel \lightning a \parallel a(\epsilon) \leftrightsquigarrow b(\overrightarrow{V})$

as required.

Case E-CloseZap

$$(\nu a)(\nu b)(\mathcal{F}[\textbf{close}\ a] \parallel \lightning b \parallel a(\epsilon) \leftrightsquigarrow b(\epsilon)) \longrightarrow \mathcal{F}[()]$$

Assumption: $\Gamma; \Delta \vdash^{\bullet} (\nu a)(\nu b)(\mathcal{F}[\textbf{close}\ a] \parallel \lightning b \parallel a(\epsilon) \leftrightsquigarrow b(\epsilon))$

By inversion on T-Nu, twice:
- $\Gamma; \Delta, a : S^\sharp, b : T^\sharp \vdash^\bullet \mathcal{F}[\mathbf{close}\ a] \parallel \lightning b \parallel a(\epsilon) \leftrightsquigarrow b(\epsilon)$

By inversion on T-Connect$_1$, twice, and T-Buffer:
- $\Gamma, a : S; \cdot \vdash^\bullet \mathcal{F}[\mathbf{close}\ a]$
- $b : T; \cdot \vdash^\bullet \lightning b$
- $\cdot; a : \overline{S}, b : \overline{T} \vdash^\circ a(\epsilon) \leftrightsquigarrow b(\epsilon)$
- $\Delta = \cdot$

By inversion on T-Main:
- $\exists E.\mathcal{F}[\mathbf{close}\ a] = \bullet E[\mathbf{close}\ a]$
- $\exists C.\Gamma, a : S \vdash E[\mathbf{close}\ a] : C$

By Lemma A.2:
- $\exists \mathbf{D}'$ such that $\mathbf{D}'$ is a subderivation of $\mathbf{D}$ concluding that $a : S \vdash \mathbf{close}\ a : \mathbf{1}$
- The position of $\mathbf{D}'$ in $\mathbf{D}$ corresponds to the position of the hole in E.

By T-Unit, Lemma A.3, and T-Main:
- $\Gamma; \Delta \vdash^\bullet \mathcal{F}[()]$

as required.

Case E-Cancel

$$\mathcal{F}[\mathbf{cancel}\ a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(Q)) \longrightarrow \mathcal{F}[()] \parallel \lightning a \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$$

Assumption: $\Gamma; \Delta \vdash^\bullet \mathcal{F}[\mathbf{cancel}\ a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$
By inversion on T-Connect$_1$:
- $\exists \Gamma_1, \Gamma_2.\Gamma = \Gamma_1, \Gamma_2$
- $\exists \Delta_1, \Delta_2.\Delta = \Delta_1, \Delta_2, a : S^\sharp$
- $\Gamma_1, a : S; \Delta_1 \vdash^\bullet \mathcal{F}[\mathbf{cancel}\ a]$
- $\Gamma_2; \Delta_2, a : \overline{S} \vdash^\circ a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$

By the definition of F:
- $\exists E.F = \bullet(E[\mathbf{cancel}\ a])$

By inversion on T-Main:
- $\Delta_1 = \cdot$
- $\Gamma_1, a : S \vdash E[\mathbf{cancel}\ a] : C$

By Lemma A.2:
- $\exists \Gamma_3, \Gamma_4.\Gamma_1, a : S = \Gamma_3, \Gamma_4, a : S$
- $\exists \mathbf{D}'$ such that $\mathbf{D}'$ is a subderivation of $\mathbf{D}$ concluding $\Gamma_4, a : S \vdash \mathbf{cancel}\ a : B$
- The position of $\mathbf{D}'$ in $\mathbf{D}$ corresponds to the position of the hole in E.

By inversion on T-Cancel:
- $\Gamma_4 = \cdot$
- $B = \mathbf{1}$

By Lemma A.3:
- $\Gamma_3 \vdash E[()] : C$

By T-Main:
- $\Gamma_3; \cdot \vdash^\bullet E[()]$

By T-Zap:
- $a : S; \cdot \vdash \lightning a$

By T-Connect$_1$:

- $\Gamma_2; \Delta_2, a : S^\sharp \vdash^\circ \lightning a \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$

By T-Mix:

- $\Gamma_2, \Gamma_3; \Delta_2, a : S^\sharp \vdash^\bullet E[()] \parallel \lightning a \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W}) \parallel \lightning a$

Since $\Gamma = \Gamma_2, \Gamma_3$ and $\Delta = \Delta_2, a : S^\sharp$, we have that

- $\Gamma; \Delta_2, a : S^\sharp \vdash^\bullet E[()] \parallel \lightning a \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$

as required.

Case E-Zap

$\lightning a \parallel a(U \cdot \overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})) \longrightarrow \lightning a \parallel \lightning c_1 \parallel \cdots \parallel \lightning c_n \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$

Assumptions:

- $\Gamma; \Delta \vdash^\circ \lightning a \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$
- $\mathsf{fn}(U) = \{c_i\}_i$

By inversion (T-Connect$_1$):

- $\Delta = \Delta', a : S^\sharp$
- $a : S \vdash^\circ \lightning a$
- $\Gamma; \Delta, a : \overline{S} \vdash^\circ a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$

By inversion (T-Buffer):

- $\exists \Gamma_1, \Gamma_2 . \Gamma = \Gamma_1, \Gamma_2$
- $\Delta' = b : T$
- $\Gamma_1 \vdash U \cdot \overrightarrow{V} : A \cdot \overrightarrow{A}$
- $\Gamma_2 \vdash \overrightarrow{W} : \overrightarrow{B}$
- $\overline{S} / \overrightarrow{A} = \overline{T / \overrightarrow{B}}$

By definition of slicing:

- $\overline{S} = !A.\overline{S'}$

By duality:

- $S = ?A.S'$

By buffer typing:

- $\exists \Gamma_3, \Gamma_4 . \Gamma_1 = \Gamma_3, \Gamma_4$
- $\Gamma_3 \vdash U : A$
- $\Gamma_4 \vdash \overrightarrow{V} : \overrightarrow{A}$

By assumptions (Γ only contains channel variables; $\mathsf{fn}(U) = \{c_i\}_i$):

- $\Gamma_3 = c_1 : S_{c_1}, \ldots, c_n : S_{c_n}$

By T-Buffer:

- $\Gamma_2; a : \overline{S'}, b : T \vdash^\circ a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$

By T-Zap:

- $a : S'; \cdot \vdash^\circ \lightning a$

By T-Zap (repeated applications):

- $c_i : S_{c_i}; \cdot \vdash^\circ \lightning c_i$ for all c_i

By T-Mix (repeated applications):

- $\Gamma; a : \overline{S'}, b : T \vdash^\circ \lightning c_1 \parallel \cdots \parallel \lightning c_n \parallel a(\lightning) \leftrightsquigarrow b(\overrightarrow{W})$

By T-Connect$_1$:

- $\Gamma; a : S'^\sharp, b : T \vdash^\circ \lightning a \parallel \lightning c_1 \parallel \cdots \parallel \lightning c_n \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$

Recalling that $\Delta' = b : T$, and that $S = ?A.S'$, we have that $?A.S' \longrightarrow S'$, and therefore that $\Gamma; \Delta \longrightarrow \Gamma; a : S'^\sharp, b : T$.

Thus
- $\Gamma; a : S'^{\sharp}, b : T \vdash^{\circ} \lightning a \parallel \lightning c_1 \parallel \cdots \parallel \lightning c_n \parallel a(\overrightarrow{V}) \rightsquigarrow b(\overrightarrow{W})$

as required.

Case E-RaiseMain

$\bullet P[\mathbf{raise}] \longrightarrow \mathbf{halt} \parallel \lightning a_1 \parallel \cdots \parallel \lightning a_n$ where $\mathsf{fn}(E) = \{a_i\}_{i \in 1..n}$

Assumption: $\Gamma; \Delta \vdash^{\bullet} \bullet P[\mathbf{raise}]$

By inversion on T-Main, we have that:
- $\Gamma \vdash P[\mathbf{raise}] : A$
- $\Delta = \cdot$

By Lemma A.2:
- $\exists \Gamma_1, \Gamma_2 . \Gamma = \Gamma_1, \Gamma_2$
- $\exists \mathbf{D}'$ such that $\mathbf{D}'$ is a subderivation of $\mathbf{D}$ concluding that $\Gamma_1 \vdash \mathbf{raise} : B$
- The position of $\mathbf{D}'$ in $\mathbf{D}$ corresponds to the position of the hole in E.

By inversion on T-Raise:
- $\Gamma_1 = \cdot$

So
- $\Gamma = \Gamma_2$

Since Γ contains only runtime names, we have that
- $\Gamma = a_1 : S_1, \ldots, a_n : S_n$

By T-Halt:
- $\cdot; \cdot \vdash^{\bullet} \bullet\mathbf{halt}$

By T-Zap, we have:
- $a_i : S_i; \cdot \vdash^{\circ} \lightning a_i$

for all a_i

By repeated applications of T-Mix, we have:
- $\Gamma; \Delta \vdash^{\bullet} \mathbf{halt} \parallel \lightning a_1 \parallel \cdots \parallel \lightning a_n$

as required.

Case E-RaiseChild

$\circ P[\mathbf{raise}] \longrightarrow \lightning a_1 \parallel \cdots \parallel \lightning a_n$ where $\mathsf{fn}(E) = \{a_i\}_{i \in 1..n}$

Assumption: $\Gamma; \Delta \vdash^{\circ} \circ P[\mathbf{raise}]$

By inversion on T-Child, we have that:
- $\Gamma \vdash P[\mathbf{raise}] : \mathbf{1}$
- $\Delta = \cdot$

By Lemma A.2:
- $\exists \Gamma_1, \Gamma_2 . \Gamma = \Gamma_1, \Gamma_2$
- $\exists \mathbf{D}'$ such that $\mathbf{D}'$ is a subderivation of $\mathbf{D}$ concluding that $\Gamma_1 \vdash \mathbf{raise} : \mathbf{1}$
- The position of $\mathbf{D}'$ in $\mathbf{D}$ corresponds to the position of the hole in H.

By inversion on T-Raise:
- $\Gamma_1 = \cdot$

So
- $\Gamma = \Gamma_2$

Since Γ contains only runtime names, we have that
- $\Gamma = a_1 : S_1, \ldots, a_n : S_n$

By T-Zap, we have:

- $a_i : S_i; \cdot \vdash^{\circ} \lightning a_i$

for all a_i

By repeated applications of T-Mix, we have:

- $\Gamma; \Delta \vdash^{\circ} \lightning a_1 \parallel \cdots \parallel \lightning a_n$

as required.

Case E-Raise

$\mathcal{F}[\textbf{try}\ P[\textbf{raise}]\ \textbf{as}\ x\ \textbf{in}\ M\ \textbf{otherwise}\ N] \longrightarrow$
$\mathcal{F}[N] \parallel \lightning a_1 \parallel \cdots \parallel \lightning a_n$ where $\text{fn}(P[\textbf{raise}]) = \{a_i\}_{i \in 1..n}$

By inversion on T-Main, we have that $\exists E.D = \bullet E[\textbf{try}\ L\ \textbf{as}\ x\ \textbf{in}\ M\ \textbf{otherwise}\ N]$

- $\Gamma \vdash E[\textbf{try}\ L\ \textbf{as}\ x\ \textbf{in}\ M\ \textbf{otherwise}\ N] : A$
- $\Delta = \cdot$

By Lemma A.2:

- $\exists \Gamma_1, \Gamma_2.\Gamma = \Gamma_1, \Gamma_2$
- $\exists \mathbf{D}'$ such that $\mathbf{D}'$ is a subderivation of $\mathbf{D}$ concluding that $\Gamma_1 \vdash \textbf{try}\ L\ \textbf{as}\ x\ \textbf{in}\ M\ \textbf{otherwise}\ N : B$
- The position of $\mathbf{D}'$ in $\mathbf{D}$ corresponds to the position of the hole in H.

By inversion on T-Try:

- $\Gamma_1 \vdash L : B'$
- $x : B' \vdash M : B$
- $\cdot \vdash N : B$

Since Γ contains only runtime names, we have that

- $\Gamma_1 = a_1 : S_1, \ldots, a_n : S_n$

By Lemma A.3:

- $\Gamma_2 \vdash^{\bullet} E[N] : B$

By T-Zap, we have:

- $a_i : S_i; \cdot \vdash^{\circ} \lightning a_i$

for all a_i

By repeated applications of T-Mix, we have:

- $\Gamma; \cdot \vdash^{\bullet} \mathcal{F}[N] \parallel \lightning a_1 \parallel \cdots \parallel \lightning a_n$

Since $\Delta = \cdot$, we have that

- $\Gamma; \Delta \vdash^{\bullet} \mathcal{F}[N] \parallel \lightning a_1 \parallel \cdots \parallel \lightning a_n$

as required.

Case E-Lift

$$\mathcal{G}[C] \longrightarrow \mathcal{G}[C']$$

if $C \longrightarrow C'$.

Assumption:

- $\Gamma; \Delta \vdash^{\phi} \mathcal{G}[C]$

Call this derivation $\mathbf{D}$.

By Lemma A.4:

- $\exists \Gamma', \Delta', \phi'$ such that $\mathbf{D}$ has a subderivation concluding $\Gamma'; \Delta' \vdash^{\phi'} C$
- The position of $\mathbf{D}'$ in $\mathbf{D}$ corresponds to the position of the hole in $\mathcal{G}$.

By the induction hypothesis:

- $\exists \Gamma''; \Delta''.\Gamma''; \Delta'' \vdash^{\phi'} C'$
- $\Gamma'; \Delta' \longrightarrow^{?} \Gamma''; \Delta''$

By Lemma A.5:

- $\exists \Gamma''', \Delta'''$ such that $\Gamma'''; \Delta''' \vdash^{\phi} \mathcal{G}[C']$ and $\Gamma; \Delta \longrightarrow^{?} \Gamma'''; \Delta'''$

as required.

Case E-LiftM

$$\phi M \longrightarrow \phi M'$$

if $M \longrightarrow_{\mathrm{M}} M'$.

Assumption:

- $\Gamma; \Delta \vdash^{\bullet} \bullet M$

By inversion on T-Main:

- $\Delta = \cdot$
- $\Gamma \vdash M : A$

By Lemma 3.1, we have that:

- $\Gamma \vdash M' : A$

By T-Main:

- $\Gamma; \cdot \vdash^{\bullet} \bullet M'$

as required.

□

A.2 Canonical Forms

Theorem 3.11: Canonical Forms *Given C such that $\Gamma; \Delta \vdash^{\bullet} C$, there exists some $C' \equiv C$ such that $\Gamma; \Delta \vdash^{\bullet} C'$ and C' is in canonical form.*

Proof. The proof is by induction on the count of ν-bound variables, following Lindley and Morris [2015]. Without loss of generality, assume that the ν-bound variables of C are distinct. Let $\{a_i \mid 1 \leq i \leq n\}$ be the set of ν-bound variables in C and let $\{\mathcal{D}_j \mid 1 \leq j \leq m\}$ be the set of threads in C.

In the case that $n = 0$, all threads must be composed using T-Mix; we can therefore use commutativity and associativity of parallel composition to derive a well-typed canonical form.

In the case that $n \geq 1$, pick some a_i and $\mathcal{D}_j$ such that a_i is the only ν-bound variable in $\mathsf{fn}(\mathcal{D}_j)$; Lemma 3.7 and a standard counting argument ensure that such a name and configuration exist. By the equivalence rules, there exists $\mathcal{E}$ such that $\Gamma; \Delta \vdash^{\phi} C \equiv (\nu a_i)(\mathcal{D}_j \parallel \mathcal{E})$ (that a_i is the only ν-bound variable in $\mathsf{fn}(\mathcal{D}_j)$ ensures well-typing). Moreover, we have that there exist $\Gamma' \subseteq \Gamma$, $\Delta' \subseteq \Delta$, and S, such that either $\Gamma', a_i : S; \Delta' \vdash^{\phi} \mathcal{E}$ or $\Gamma'; \Delta', a_i : S \vdash^{\phi} \mathcal{E}$. By the induction hypothesis, there exists $\mathcal{E}'$ in canonical form such that either $\Gamma', a_i : S; \Delta' \vdash^{\phi} \mathcal{E} \equiv \mathcal{E}'$ or $\Gamma'; \Delta', a_i : S \vdash^{\phi} \mathcal{E} \equiv \mathcal{E}'$. Let $C' = (\nu a_i)(\mathcal{D}_j \parallel \mathcal{E}')$. By construction it holds that $\Gamma; \Delta \vdash^{\phi} C \equiv C'$ and that C' is in canonical form.

□

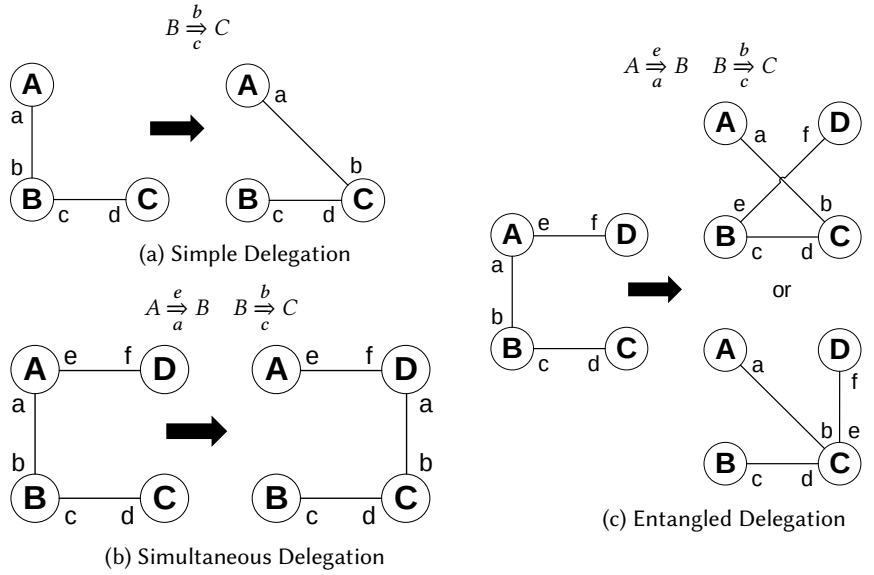


Fig. 12. Cases of Distributed Delegation

B DISTRIBUTED DELEGATION

A key feature of π-calculus is *mobility*, that is, sending channel names as values. In session-based languages and calculi, mobility is realised as *session delegation*, allowing session-typed channel endpoints to be sent over other session-typed channels. We saw an example of session delegation in §6, in the `ChatClient` type:

```
typename ChatClient =!Nickname.
  [&|Join:?(Topic, [Nickname], ClientReceive).ClientSend,
     Nope:End|&];
```

An endpoint of type `ClientReceive` is passed as a message.

B.1 Challenges of Distributed Delegation

Session delegation is a vital abstraction in session-based programming. However, its integration with both asynchrony *and* distribution brings several challenges. The seminal work on distributed delegation is Session Java [Hu et al. 2008].

Fig. 12 shows three scenarios of distributed delegation, as described by Hu et al. [2008]. We write $X \overset{x}{\underset{y}{\Rightarrow}} Y$ to indicate that X wishes to send x to Y over y on the basis that X's last known location of the corresponding endpoint for y is Y. Now suppose $B \overset{b}{\underset{c}{\Rightarrow}} C$. Following Hu et al. [2008], we refer to B as the *session-sender*, C as the *session-receiver*, and A as a *passive party*. There is no happens-before relation between A sending a message to B along a, and B delegating b to C along c. Thus, a message could be sent to A *after* A has given up control of a. Following Hu et al. [2008], we call such messages *lost messages*.

1. $A \to S : \mathsf{Send}(t, v, [b \mapsto \overrightarrow{V}])$
2. $\quad A :$ *start recording lost messages* $\overrightarrow{W}$ *for* b
3. $\quad S : \sigma = \sigma[b \mapsto B]; \delta = \delta \cup \{t\}$
4. $S \to B : \mathsf{Deliver}(t, v, [b \mapsto \overrightarrow{V}])$
5. $S \to A : \mathsf{GetLostMessages}([b])$
6. $\quad A :$ *stop recording lost messages for* b
7. $A \to S : \mathsf{LostMessageResponse}([b \mapsto \overrightarrow{W}])$
8. $S \to B : \mathsf{Commit}(t, [b \mapsto \overrightarrow{W}])$
9. $\quad S : \delta = \delta \backslash \{t\}$
10. $\quad B :$ *buffers*$[b] = \overrightarrow{V} +\!\!+ \overrightarrow{W} +\!\!+ \overrightarrow{U}$
    where $\overrightarrow{U} =$ *messages received for* b *between (3) and (8)*

Fig. 13. Operation of Distributed Delegation Protocol

B.2 Approaches to Distributed Delegation

The simplest safe way to implement distributed delegation is to store all buffers on the server, but this requires a blocking remote call for every receive operation. A second naïve method is *indefinite redirection*, where the session-sender indefinitely forwards all messages to the session-receiver. This retains buffer locality, but requires the session-sender to remain online for the duration of the delegated session.

Hu et al. [2008] describe two more realistic distributed delegation algorithms: a *resending* protocol, which re-sends lost messages *after* a connection for the delegated session is established, and a *forwarding* protocol, which forwards lost messages *before* the delegated session is established. The key idea behind both algorithms is to establish a connection between the passive party and the session-receiver, ensure that the lost messages are received by the session-receiver, and to continue the session only once lost messages are received.

B.3 Delegation in Distributed Session Links

Alas, we cannot directly re-use the resending and forwarding protocols of Hu et al. [2008] because of two fundamental differences in our setting: Links clients do not connect to each other directly, and in Links multiple sessions may be sent at once. Thus, we describe the high-level details of a modified algorithm which addresses these two constraints. We utilise two key ideas:

- Much like the resending protocol, lost messages are retrieved and relayed to the session-receiver once the new session has been established.
- We ensure the session-receiver endpoint is not delegated until the delegation has completed, by queueing messages that include the session-receiver endpoint, and resending them once delegation has completed.

We now consider the case where session-sender and session-receiver are different clients; the case where session-sender is a client and session-receiver the server is similar. Let client A be session-sender and client B be session-receiver.

Example. Suppose client A sends a value v containing a session endpoint d along channel (s, t), recalling that s is the peer endpoint and t is the local endpoint. The initial endpoint location table is:

$$\sigma \triangleq [s \mapsto A, t \mapsto B, b \mapsto A, c \mapsto A]$$

Fig. 13 shows the operation of the delegation protocol on this example. In Step 1, A sends a message to the server S, containing the peer endpoint t, value to send v, and the buffer $\overrightarrow{V}$ for b, before beginning to record lost messages for b. Upon receiving this message, the server updates its internal mapping for the location of b to be B, adds t to the set of delegation carriers δ, and sends a Deliver message containing t, v, and $\overrightarrow{V}$, before sending a GetLostMessages request to A. Upon receiving this message, A will stop recording lost messages for b, and relay the lost messages $\overrightarrow{W}$ for b to S. The server then sends a Commit message containing t and the lost messages for all delegated endpoints, and removes t from the set of delegation carriers.

The final buffer for b is the concatenation of the initial buffer $\overrightarrow{V}$, the lost messages $\overrightarrow{W}$, and all messages $\overrightarrow{U}$ received for b before the Commit message.

B.4 Correctness

We argue correctness of the algorithm in a similar manner to Hu et al. [2008]. Due to co-operative threading, we can treat each sequence of actions happening at a single participant (for example, steps 3–8) as atomic. Since (as per step 3) the endpoint location table is updated prior to the lost message request, we can safely split the buffer of the delegated session into three parts: the initial buffer being delegated ($\overrightarrow{V}$); the lost messages ($\overrightarrow{W}$); and the messages received after the change in the lookup table but before the Commit message is received ($\overrightarrow{U}$) and reassemble them, retaining ordering.

In our setting, since session channels are not associated with sockets, simultaneous delegation (Fig. 12b) can be handled in the same way as simple delegation. In the case of entangled delegation (Fig. 12c, since delegation carriers may not be delegated themselves until the lost messages have been received, we can be sure that the lost message requests are sent to the correct participant. Hence, the case devolves to simple delegation.