Exceptional Asynchronous Session Types

Session Types without Tiers

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

Session types statically guarantee that 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 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.

ACM Reference Format:
Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. 2019. Exceptional Asynchronous Session Types: Session Types without Tiers. *Proc. ACM Program. Lang.* POPL, 1, Article 1 (November 2019), 56 pages. https://doi.org/10.475/123_4

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 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

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

2475-1421/2019/11-ART1
https://doi.org/10.475/123_4

$$\begin{array}{l}\mathsf{TwoFactorServer} \triangleq \\ \quad ?(\mathsf{Username}, \mathsf{Password}).\oplus\{ \\ \qquad \mathsf{Authenticated} : \mathsf{ServerBody}, \\ \qquad \mathsf{Challenge} : !\mathsf{ChallengeKey}.?\mathsf{Response}. \\ \qquad\quad \oplus\{\mathsf{Authenticated} : \mathsf{ServerBody}, \\ \qquad\qquad \mathsf{AccessDenied} : \mathsf{End}\}, \\ \qquad \mathsf{AccessDenied} : \mathsf{End}\}\end{array}$$

(a) Server Session Type

$$\begin{array}{l}\mathsf{TwoFactorClient} \triangleq \\ \quad !(\mathsf{Username}, \mathsf{Password}).\&\{ \\ \qquad \mathsf{Authenticated} : \mathsf{ClientBody}, \\ \qquad \mathsf{Challenge} : ?\mathsf{ChallengeKey}.!\mathsf{Response}. \\ \qquad\quad \&\{\mathsf{Authenticated} : \mathsf{ClientBody}, \\ \qquad\qquad \mathsf{AccessDenied} : \mathsf{End}\}, \\ \qquad \mathsf{AccessDenied} : \mathsf{End}\}\end{array}$$

(b) Client Session Type

Fig. 1. Two-factor Authentication Session Types

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 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.

Duality. 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 $\mathsf{TwoFactorClient} = \overline{\mathsf{TwoFactorServer}}$ and $\mathsf{TwoFactorServer} = \overline{\mathsf{TwoFactorClient}}$.

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

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

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

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

$$
\begin{array}{l}
\mathsf{twoFactorClient} : (\mathsf{Username} \times \mathsf{Password} \times \mathsf{TwoFactorClient}) \multimap \mathbf{1} \\
\mathsf{twoFactorClient}(\mathit{username}, \mathit{password}, s) \triangleq \\
\quad \textbf{let}\ s = \textbf{send}\ (\mathit{username}, \mathit{password})\ s\ \textbf{in} \\
\quad \textbf{offer}\ s\ \{ \begin{array}[t]{@{}l@{\ }l}
\mathsf{Authenticated}(s) & \mapsto \mathsf{clientBody}(s) \\
\mathsf{Challenge}(s) & \mapsto \begin{array}[t]{@{}l}
\textbf{let}\ (\mathit{key}, s) = \textbf{receive}\ s\ \textbf{in} \\
\textbf{let}\ s = \textbf{send}\ (\mathsf{generateResponse}(\mathit{key}))\ s\ \textbf{in} \\
\textbf{offer}\ s\ \{ \begin{array}[t]{@{}l@{\ }l}
\mathsf{Authenticated}(s) & \mapsto \mathsf{clientBody}(s) \\
\mathsf{AccessDenied}(s) & \mapsto \textbf{close}\ s; \mathsf{loginFailed} \}
\end{array}
\end{array} \\
\mathsf{AccessDenied}(s) & \mapsto \textbf{close}\ s; \mathsf{loginFailed} \}
\end{array}
\end{array}
$$

The twoFactorClient function takes the credentials and an endpoint of type TwoFactorClient as its arguments. The credentials are sent along the endpoint, then three choices are offered depending on whether the server authenticates the user, sends a two-factor challenge, or rejects the authentication attempt. If the server authenticates the user, then the program progresses to the main application (clientBody(s)). If the server sends a challenge, then the client receives the challenge key, and sends the response, calculated by generateResponse. Two choices are then offered according to whether the challenge response was successful. The rejection of an authentication attempt is part of the protocol and *not* exceptional behaviour. We can also write a server implementation.

$$
\begin{array}{l}
\mathsf{twoFactorServer} : \mathsf{TwoFactorServer} \multimap \mathbf{1} \\
\mathsf{twoFactorServer}(s) \triangleq \begin{array}[t]{@{}l}
\textbf{let}\ ((\mathit{username}, \mathit{password}), s) = \textbf{receive}\ s\ \textbf{in} \\
\textbf{if}\ \mathsf{checkDetails}(\mathit{username}, \mathit{password})\ \textbf{then} \\
\quad \textbf{let}\ s = \textbf{select}\ \mathsf{Authenticated}\ s\ \textbf{in}\ \mathsf{serverBody}(s) \\
\textbf{else} \\
\quad \textbf{let}\ s = \textbf{select}\ \mathsf{AccessDenied}\ s\ \textbf{in}\ \textbf{close}\ s
\end{array}
\end{array}
$$

The twoFactorServer function takes an endpoint of type TwoFactorServer along which it receives the credentials, which are checked using checkDetails. If the check passes, then the server proceeds to the application body (serverBody(s)); if not, then the server notifies the client by selecting the AccessDenied branch. This particular server implementation opts to never send a challenge request.

Statically checking session types demands 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 insufficient for safely implementing session types. Consider the following client:

$$
\begin{array}{l}
\mathsf{wrongClient} : \mathsf{TwoFactorClient} \multimap \mathbf{1} \\
\mathsf{wrongClient}(s) \triangleq \begin{array}[t]{@{}l}
\textbf{let}\ t = \textbf{send}\ (\texttt{"Alice"}, \texttt{"hunter2"})\ s\ \textbf{in} \\
\textbf{let}\ t = \textbf{send}\ (\texttt{"Bob"}, \texttt{"letmein"})\ s\ \textbf{in}\ \ldots
\end{array}
\end{array}
$$

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, communication over an endpoint matches its session type. To maintain session fidelity and ensure that all communication actions in a session type occur, session type systems typically require that each endpoint is used *linearly*—exactly once.

Exceptions. In practice, linear session types are unrealistic. Thus far, we have assumed checkDetails always succeeds, which may be plausible if checking against an in-memory store, but not if connecting to a remote database. One option would be for checkDetails to return false on failure, but that would lose information. Instead, suppose we have an exception handling construct. As a first attempt, we might try to write:

$$
\begin{array}{l}
\mathsf{exnServer1} : \mathsf{TwoFactorClient} \multimap \mathbf{1} \\
\mathsf{exnServer1}(s) \triangleq \mathbf{let}\ ((\mathit{username}, \mathit{password}), s) = \mathbf{receive}\ s\ \mathbf{in} \\
\qquad \mathbf{try}\ \mathbf{if}\ \mathsf{checkDetails}(\mathit{username}, \mathit{password})\ \mathbf{then} \\
\qquad\qquad \mathbf{let}\ s = \mathbf{select}\ \mathsf{Authenticated}\ s\ \mathbf{in}\ \mathsf{serverBody}(s) \\
\qquad\quad \mathbf{else} \\
\qquad\qquad \mathbf{let}\ s = \mathbf{select}\ \mathsf{AccessDenied}\ s\ \mathbf{in}\ \mathbf{close}\ s \\
\qquad \mathbf{catch}\ \mathsf{log}(\texttt{"Database Error"})
\end{array}
$$

However, the above code does not type-check and is unsafe. Linear endpoint s is not used in the **catch** block and yet is still 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.

$$
\begin{array}{l}
\mathsf{checkDetailsOpt} : (\mathsf{Username} \times \mathsf{Password}) \multimap \mathsf{Option}(\mathsf{Bool}) \\
\mathsf{checkDetailsOpt}(\mathit{username}, \mathit{password}) \triangleq \mathbf{try}\ \mathsf{Some}(\mathsf{checkDetails}(\mathit{username}, \mathit{password})) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{catch}\ \mathsf{None}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{exnServer2} : \mathsf{TwoFactorServer} \multimap \mathbf{1} \\
\mathsf{exnServer2}(s) \triangleq \mathbf{let}\ ((\mathit{username}, \mathit{password}), s) = \mathbf{receive}\ s\ \mathbf{in} \\
\qquad \mathbf{case}\ \mathsf{checkDetailsOpt}(\mathit{username}, \mathit{password})\ \mathbf{of} \\
\qquad\quad \mathsf{Some}(\mathit{res}) \mapsto \mathbf{if}\ \mathit{res}\ \mathbf{then}\ \mathbf{let}\ s = \mathbf{select}\ \mathsf{Authenticated}\ s\ \mathbf{in}\ \mathsf{serverBody}(s) \\
\qquad\qquad\qquad\qquad\quad \mathbf{else}\ \mathbf{let}\ s = \mathbf{select}\ \mathsf{AccessDenied}\ s\ \mathbf{in}\ \mathbf{close}\ s \\
\qquad\quad \mathsf{None} \mapsto \mathsf{log}(\texttt{"Database Error"})
\end{array}
$$

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:

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

We could use **select** InternalError s in the None branch to yield a type-correct program, but doing so would be unsatisfactory as it clutters the protocol and the implementation with failure points.

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 form the basis of the existing Rust implementation of session types [Jespersen et al. 2015] and dynamically checked affine types form the basis of the OCaml FuSe [Padovani 2017] and Scala `lchannels` [Scalas and Yoshida 2016] session type libraries. Affine types present two quandaries arising from endpoints being silently discarded. First, a developer receives no feedback if they *accidentally* forget to finish a protocol implementation. Second, if an exception is raised in an evaluation context that captures an open endpoint then the peer may be left waiting forever.

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. They provide an exception handling construct which attempts a communication action, running an exception handler if the action fails, and show that explicit cancellation is well-behaved: their calculus satisfies preservation and global progress (well-typed processes never get stuck), and is confluent.

Explicit cancellation neatly handles failure while ruling out 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; this is not merely 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*. Our exception handling construct is *compositional*.

We define a core concurrent λ-calculus, *Exceptional GV* (EGV), 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:

$$
\begin{array}{l}
\mathsf{exnServer3} : \mathsf{TwoFactorServer} \multimap \mathbf{1} \\
\mathsf{exnServer3}(s) \triangleq \mathbf{let}\ ((\mathit{username}, \mathit{password}), s) = \mathbf{receive}\ s\ \mathbf{in} \\
\qquad\qquad\qquad\quad \mathbf{try}\ \mathsf{checkDetails}(\mathit{username}, \mathit{password})\ \mathbf{as}\ \mathit{res}\ \mathbf{in} \\
\qquad\qquad\qquad\qquad \mathbf{if}\ \mathit{res}\ \mathbf{then}\ \mathbf{let}\ s = \mathbf{select}\ \mathsf{Authenticated}\ s\ \mathbf{in}\ \mathsf{serverBody}(s) \\
\qquad\qquad\qquad\qquad \mathbf{else}\ \mathbf{let}\ s = \mathbf{select}\ \mathsf{AccessDenied}\ s\ \mathbf{in}\ \mathbf{close}\ s \\
\qquad\qquad\qquad\quad \mathbf{otherwise} \\
\qquad\qquad\qquad\qquad \mathbf{cancel}\ s;\ \mathsf{log}(\texttt{"Database Error"})
\end{array}
$$

Following Benton and Kennedy [2001], an exception handler **try** L **as** x **in** M **otherwise** N takes an explicit success continuation M as well as the usual failure continuation N. If checkDetails fails with an exception, then s is safely discarded 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.

```
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

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 core 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 (§5.4), 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 by invoking `opam install links`.

The rest of the paper is structured as follows: §2 presents Exceptional GV and §3 its metatheory; §4 discusses extensions 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 that has 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.

Due to GV's close correspondence with classical linear logic, EGV has a strong metatheory, enjoying preservation, global progress, the diamond property, and termination. Much like the simply-typed λ-calculus, this well-behaved core must be extended to be expressive enough to write larger applications. Nonetheless, the core calculus alone is expressive enough to support our two-factor authentication example, and to support server applications which gracefully handle disconnection. In §3, we show that cancellation is well-behaved, and does not violate any of the core properties of GV. In §4, following Lindley and Morris [2015, 2017], we extend EGV modularly

Types	$A, B, C ::= \mathbf{1} \mid A \multimap B \mid A + B \mid A \times B \mid S$
Session Types	$S, T ::= {!A.S} \mid {?A.S} \mid \mathsf{End}$
Variables	x, y
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

with standard features of our implementation, some of which provide weaker guarantees. Channel cancellation and exceptions are orthogonal to these features.

2.1 Integrating Sessions with Exceptions, by Example

Integrating session types with failure handling into a higher-order functional language requires care. Fig. 2 illustrates three important cases: 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. Fig. 2a forks a thread which immediately cancels its endpoint. The parent attempts to receive, but the message can never arrive so an exception is raised and the **otherwise** clause is invoked.

Delegation. A central feature of 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 is passed endpoint t on which it blocks receiving. Next, the parent forks a second child, yielding endpoint u. The second child is passed endpoint v, which is immediately discarded using **cancel**. Now the parent thread sends endpoint s along u. Endpoint s will never be received as the peer endpoint v of u has been cancelled. In turn, this renders s irretrievable and an exception is thrown in the first child thread, as it can never receive a value.

Closures. It is crucial that cancellation plays nicely with closures. The code in Fig. 2c defines a function f which sends its argument x along s. The parent thread then raises an exception. As s appears in the closure bound to f, which appears in the continuation and is thus discarded, s must 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 **let** $() = M$ **in** N and **let** $x = M$ **in** N for $(\lambda x.N)\,M$. The standard session typing primitives [Lindley and Morris 2015] are as follows: **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; **send** $M\,N$ sends M along endpoint N; **receive** M receives along endpoint M; and **close** M closes an endpoint when a session is complete.

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

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

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

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

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

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

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

We introduce three new term constructs to support session typing with failure handling: **cancel** M explicitly discards session endpoint M; **raise** raises an exception; and **try** L **as** x **in** M **otherwise** N evaluates L, on success binding the result to x in M and on failure evaluating N.

Explicit success continuations. Benton and Kennedy [2001] argue that:

> From the points of view of programming pragmatics, rewriting and operational semantics, the syntactic construct used for exception handling in ML-like programming languages, and in much theoretical work on exceptions, has subtly undesirable features.

Benton and Kennedy show that explicit success continuations avoid the subtly undesirable features they identify; correspondingly, we adopt their construct. Moreover, explicit success continuations align with the definition of handlers for algebraic effects [Plotkin and Pretnar 2013] that we use in our implementation (§5.4).

Branching and selection. 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].

Typing. Fig. 4 gives the typing rules for EGV. As usual, linearity is enforced by splitting environments when typing subterms, ensuring T-Var takes a singleton environment, and leaf rules T-Unit and T-Raise take an empty environment. We write Γ_1, Γ_2 to mean the disjoint union of Γ_1 and Γ_2. 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.

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

Syntactic Sugar

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

Fig. 5. Runtime Syntax

As exceptions do not return values, the rule T-Raise allows an exception to be given any type A. Rule T-Try embraces explicit success continuations as advocated by Benton and Kennedy [2001], binding 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. We avoid this pitfall by raising an exception when a communication action would wait forever due to cancellation.

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 write $S^\sharp$ for the type of a channel which can be split into two endpoints of types S and $\overline{S}$. Runtime types R are either session types or channel types. 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 π-calculus processes: $(\nu a)\mathcal{C}$ 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* ($\bullet$), which returns a top-level result, or a *child thread* ($\circ$), which

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

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

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

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

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

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

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

$$
\begin{array}{llcl}
\textsc{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} \\
\textsc{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) \\
\textsc{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}) \\
\textsc{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}'[()] \\
\textsc{E-Cancel} & \mathcal{F}[\mathbf{cancel}\ a] & \longrightarrow & \mathcal{F}[()] \parallel \lightning\, a \\
\textsc{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}) \\
\textsc{E-CloseZap} & \mathcal{F}[\mathbf{close}\ a] \parallel \lightning\, b \parallel a(\epsilon) \leftrightsquigarrow b(\epsilon) & \longrightarrow & \mathcal{F}[\mathbf{raise}] \parallel \lightning\, a \parallel \lightning\, b \parallel a(\epsilon) \leftrightsquigarrow b(\epsilon) \\
\textsc{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}) \\
\textsc{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 \\
\textsc{E-RaiseChild} & \circ P[\mathbf{raise}] & \longrightarrow & \lightning\, P \\
\textsc{E-RaiseMain} & \bullet P[\mathbf{raise}] & \longrightarrow & \mathbf{halt} \parallel \lightning\, P \\
\textsc{E-LiftC} & \mathcal{G}[\mathcal{C}] & \longrightarrow & \mathcal{G}[\mathcal{D}], \quad \text{if } \mathcal{C} \longrightarrow \mathcal{D} \\
\textsc{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

does not, and must return the unit value. A configuration has at most one main thread. As well as 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* ($a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$) models asynchrony: $\overrightarrow{V}$ and $\overrightarrow{W}$ are sequences of values ready to be received along endpoints a and b respectively. We 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 object 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.

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, name restriction reordering, and scope extrusion. Formally, equivalence is defined as the smallest congruence relation satisfying the equivalence axioms in Figure 6. We incorporate a further 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 endpoints 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 ($\lightning U$). It is applied repeatedly until the buffer is empty.

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 an endpoint whose buffer is empty and whose peer endpoint has been cancelled. Similarly, E-CloseZap raises an exception if an attempt is made to close a channel where the peer endpoint has been cancelled. 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 §3.4. Not raising exceptions on message sends to dead peers is standard behaviour 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).

2.4 Synchrony

As we are interested in writing distributed applications, we consider asynchronous session types. However, our semantics adapts straightforwardly to the synchronous setting, where a send to a cancelled peer must also raise an exception:

$$\begin{array}{lrcl}
\text{E-SyncComm} & \mathcal{F}[\mathbf{send}\ V\ a] \parallel \mathcal{F}'[\mathbf{receive}\ a] & \longrightarrow & \mathcal{F}[a] \parallel \mathcal{F}'[(V, a)] \\
\text{E-SyncSendZap} & \mathcal{F}[\mathbf{send}\ V\ a] \parallel \lightning a & \longrightarrow & \mathcal{F}[\mathbf{raise}] \parallel \lightning V \parallel \lightning a \parallel \lightning a \\
\text{E-SyncRecvZap} & \mathcal{F}[\mathbf{receive}\ a] \parallel \lightning a & \longrightarrow & \mathcal{F}[\mathbf{raise}] \parallel \lightning a \parallel \lightning a
\end{array}$$

$$(\nu a)(\lightning a \parallel \lightning a) \parallel C \equiv C$$

3 METATHEORY

Even in the presence of channel cancellation and exceptions, EGV retains GV's strong metatheory [Lindley and Morris 2015]. The central property of session-typed systems is session fidelity: all communication follows the prescribed session types. Session fidelity follows as a corollary of preservation of configuration typing under reduction.

Session calculi with roots in linear logic are deadlock-free as interpreting the logical cut rule as a combination of name restriction and parallel composition necessarily ensures acyclicity [Caires and Pfenning 2010]. 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

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, runtime environment Δ, and thread flag ϕ, configuration C is well-typed. We additionally require that $\mathsf{fn}(\Gamma) \cap \mathsf{fn}(\Delta) = \emptyset$. 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. We write $\Gamma; \Delta \vdash^{\bullet} C : A$ whenever the derivation of $\Gamma; \Delta \vdash^{\bullet} C$ contains a subderivation of the form

$$\frac{\Gamma' \vdash M : A}{\Gamma'; \cdot \vdash^{\bullet} \bullet M} \qquad \text{or} \qquad \frac{}{\cdot; \cdot \vdash^{\bullet} \mathbf{halt}}$$

We say that a C is a *ground configuration* if there exists A such that $\cdot; \cdot \vdash^{\bullet} C : A$ and A contains no session types or function types.

The T-Nu rule introduces a channel name; T-Connect$_1$ and T-Connect$_2$ connect two configurations over a channel; and T-Mix composes two configurations that share no channels. The latter three rules use the + operator to combine the flags from 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 runtime 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

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

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

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

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

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

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

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

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

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

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

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

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
$$\frac{S/\overrightarrow{A} = \overline{S'/\overrightarrow{B}} \quad \Gamma_1 \vdash \overrightarrow{V} : \overrightarrow{A} \quad \Gamma_2 \vdash \overrightarrow{W} : \overrightarrow{B}}{\Gamma_1, \Gamma_2; a : S, b : S' \vdash^{\circ} a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}$$

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

$$\bullet + \circ = \bullet \qquad \circ + \bullet = \bullet$$
$$\circ + \circ = \circ \qquad \bullet + \bullet \text{ undefined}$$

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

types of two buffer endpoints are compatible if they are dual up to values contained in the buffer. The partiality of the slicing operator coupled with the duality constraint ensures that at least one queue in a buffer is always empty. Appendix A shows an example configuration typing derivation.

3.2 Preservation

Preservation for the functional fragment of EGV is standard.

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

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

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

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

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

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

Typing and Configuration Equivalence. As is common in logically-inspired session-typed functional languages [Lindley and Morris 2015, 2017], typeability of configurations is *not* preserved by equivalence. Consider $\Gamma; \Delta \vdash^{\phi} (\nu a)(\nu b)(\mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E}))$ with $a \in \mathsf{fn}(\mathcal{C})$, $b \in \mathsf{fn}(\mathcal{D})$, and $a, b \in \mathsf{fn}(\mathcal{E})$. But $\Gamma; \Delta \nvdash^{\phi} (\nu a)(\nu b)((\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E})$. Fortunately this looseness of the equivalence relation is unproblematic: we may always safely re-associate parallel composition (for example, $\Gamma; \Delta \vdash^{\phi} (\nu a)(\nu b)((\mathcal{C} \parallel \mathcal{E}) \parallel \mathcal{D})$; see Appendix C.1), and any reduction sequence which uses ill-typed equivalences may be replaced by one that does not.

Theorem 3.3 (Preservation Modulo Equivalence). *If $\Psi; \Delta \vdash^{\phi} \mathcal{C}$, $\mathcal{C} \equiv \mathcal{D}$, and $\mathcal{D} \longrightarrow \mathcal{D}'$, then:*

(1) There exists some $\mathcal{E} \equiv \mathcal{D}$ and some $\mathcal{E}'$ such that $\Psi; \Delta \vdash^{\phi} \mathcal{E}$ 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. The only non-trivial reductions are those involving 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

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

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 to enable threads inside $\mathcal{C}$ and $\mathcal{D}$ to synchronise. But for synchronisation to be possible there must exist a name a such that $a \in \mathsf{fn}(\mathcal{C})$ and $a \in \mathsf{fn}(\mathcal{D})$. Because the left-hand-side of the equation is well-typed, we know that $\mathcal{C}$ 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 applying the rule from right-to-left is symmetric. In summary, any ill-typed use of equivalence is useless. □

3.3 Progress

To prove that EGV enjoys a strong notion of progress we identify a *canonical form* for configurations. We prove that every well-typed configuration is equivalent to a well-typed configuration in canonical form, and that ground configurations can always either reduce, or are equivalent to either a value or **halt**.

The functional fragment of EGV enjoys progress.

Lemma 3.4 (Progress: Open Terms). *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: **fork** V, **send** V W, **receive** V, **close** V, or **cancel** V.*

Proof. By induction on the derivation of $\Psi \vdash M : A$. □

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 3.5 (Canonical Form). A configuration $\mathcal{C}$ 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:

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

where $a_i \in \mathsf{fn}(\mathcal{A}_i)$ for each $i \in 1..n$.

The following lemma implies that communication topologies are always acyclic.

Lemma 3.6. *If* $\Gamma; \Delta \vdash^{\phi} C$ *and* $C = \mathcal{G}[\mathcal{D} \parallel \mathcal{E}]$, *then* $\mathsf{fn}(\mathcal{D}) \cap \mathsf{fn}(\mathcal{E})$ *is either* $\emptyset$ *or* $\{a\}$ *for some* a.

Proof. By induction on the derivation of $\Gamma; \Delta \vdash^{\phi} C$; the only interesting rules are those for parallel composition. As the environments are well-formed, $\mathsf{fn}(\Gamma) \cap \mathsf{fn}(\Delta) = \emptyset$. Thus, T-Connect$_1$ and T-Connect$_2$ allow exactly one name to be shared, whereas T-Mix forbids sharing of names. □

All well-typed configurations can be written in canonical form.

Theorem 3.7 (Canonical Forms). *Given* C *such that* $\Gamma; \Delta \vdash^{\bullet} C$, *there exists some* $\mathcal{D} \equiv C$ *such that* $\Gamma; \Delta \vdash^{\bullet} \mathcal{D}$ *and* $\mathcal{D}$ *is in canonical form.*

Proof. By induction on the count of ν-bound variables, following Lindley and Morris [2015] and making use of Lemma 3.6. The additional features of EGV do not change the essential argument. The full proof can be found in Appendix C.2. □

Definition 3.8. 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:

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

Using the notion of a ready thread, we may classify a notion of progress for open configurations.

Theorem 3.9 (Progress: Open). *Suppose* $\Psi; \Delta \vdash^{\bullet} C$, *where* C *is in canonical form.*
Let $C = (\nu a_1)(\mathcal{A}_1 \parallel (\nu a_2)(\mathcal{A}_2 \parallel \cdots \parallel (\nu a_n)(\mathcal{A}_n \parallel \mathcal{M}))\ldots))$.
Either there exists some C' *such that* $C \Longrightarrow C'$, *or:*

(1) *For* $1 \leq i \leq n$, *each auxiliary thread* $\mathcal{A}_i$ *is either:*
   (a) *a child thread* $\circ M$ *for which there exists* $a \in \{a_j \mid 1 \leq j \leq i\} \cup \mathsf{fn}(\Psi)$ *such that* $\mathsf{ready}(a, M)$;
   (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* $1 \leq j \leq m$:
   (a) $\mathcal{A}'_j$ *is either:*
      (i) *a child thread* $\circ N$ *with* $N = ()$ *or* $\mathsf{ready}(a, N)$ *for some* $a \in \{a_i \mid 1 \leq i \leq n\} \cup \mathsf{fn}(\Psi) \cup \mathsf{fn}(\Delta)$;
      (ii) *a zapper thread* $\lightning a$ *for some* $a \in \{a_i \mid 1 \leq i \leq n\} \cup \mathsf{fn}(\Psi) \cup \mathsf{fn}(\Delta)$; *or*
      (iii) *a buffer.*
   (b) *Either* $\mathcal{T} = \bullet N$, *where* N *is either a value or* $\mathsf{ready}(a, N)$ *for some* $a \in \{a_i \mid 1 \leq i \leq n\} \cup \mathsf{fn}(\Psi) \cup \mathsf{fn}(\Delta)$; *or* $\mathcal{T} =$ ***halt***.

Proof. The result follows from a more verbose, but finer-grained, property which we prove by induction on the derivation of $\Psi; \Delta \vdash^{\bullet} C$. Full details are in Appendix C.3. □

This theorem tells us that open reduction cannot "go wrong". A progress theorem states that either reduction is possible or the configuration is a value. Conditions 1(a)(b)(c) and 2(a)(b) constitute a suitable generalisation of 'value'.

By restricting attention to closed environments, we obtain a tighter progress property.

Theorem 3.10 (Progress: Closed). *Suppose* $\cdot; \cdot \vdash^{\bullet} C$ *where* C *is in canonical form.*
Let $C = (\nu a_1)(\mathcal{A}_1 \parallel (\nu a_2)(\mathcal{A}_2 \parallel \cdots \parallel (\nu a_n)(\mathcal{A}_n \parallel \mathcal{M}))\ldots))$.
Either there exists some C' *such that* $C \Longrightarrow C'$, *or:*

(1) *For* $1 \leq i \leq n$, *each auxiliary thread* $\mathcal{A}_i$ *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) *Either $\mathcal{M} = \bullet W$ for some value W, or $\mathcal{M} =$ **halt**.*

The above progress results do not specifically mention deadlock. However, Lemma 3.6 ensures deadlock-freedom. Nevertheless, communication can still be blocked if an endpoint appears in the value returned by the main thread. A conservative way of disallowing endpoints in the result is to insist that the return type of the program be free of session types and function types (closures may capture endpoints). All configurations of such a programs are ground configurations.

Theorem 3.11 (Global Progress). *Suppose C is a ground configuration. Either there exists some C' such that $C \Longrightarrow C'$; or $C \equiv \bullet V$; or $C \equiv$ **halt**.*

Proof. As a consequence of Theorem 3.10, either there exists some C' such that $C \Longrightarrow C'$, or $C \not\Longrightarrow$ and each thread $\mathcal{A}_i$ must be a zapper, a buffer, or ready to perform an action. If $C \not\Longrightarrow$, since C is ground, by by Lemma 3.6, we have that no thread can be ready to perform an action. Thus, each $\mathcal{A}_i$ must be either $\circ()$, a zapper, or an empty buffer. The result then follows by the garbage collection congruences of Fig. 6. □

3.4 Confluence

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

Theorem 3.12 (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. Asynchrony and cancellation introduce two critical pairs which may be resolved in a single step; see Appendix C.4 for details. □

Remark. The system becomes non-confluent if we choose to raise an exception when sending to a cancelled buffer. Suppose that instead of the current semantics, we were to replace E-Send with the following two rules:

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

Then, sending and cancelling peer endpoints of a buffer results in a non-convergent critical pair:

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

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 significance, in that it characterises the race between sending a message and propagating a cancellation notification.

3.5 Termination

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

Theorem 3.13 (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 exhaustively applying the garbage collection equivalences 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 handling 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 handling the exception in checkDetails. An exception may arise if the database is corrupt, or if there are too many connections. We might like to handle each case separately:

$$
\begin{array}{l}
\mathsf{exnServer4}(s) \triangleq \\
\quad \mathbf{let}\ ((\mathit{username}, \mathit{password}), s) = \mathbf{receive}\ s\ \mathbf{in} \\
\quad \mathbf{try}\ \mathsf{checkDetails}(\mathit{username}, \mathit{password})\ \mathbf{as}\ \mathit{res}\ \mathbf{in} \\
\qquad \mathbf{if}\ \mathit{res}\ \mathbf{then}\ \mathbf{let}\ s = \mathbf{select}\ \mathsf{Authenticated}\ s\ \mathbf{in}\ \mathsf{serverBody}(s) \\
\qquad \mathbf{else}\ \mathbf{let}\ s = \mathbf{select}\ \mathsf{AccessDenied}\ s\ \mathbf{in}\ \mathbf{close}\ s \\
\quad \mathbf{unless} \\
\qquad \mathsf{DBCorrupt}(y) \qquad\qquad\quad \mapsto \mathbf{cancel}\ s;\ \mathsf{log}(\texttt{"Database Corrupt: " + y}) \\
\qquad \mathsf{TooManyConnections}(y) \mapsto \mathbf{cancel}\ s;\ \mathsf{log}(\texttt{"Too many connections: " + y})
\end{array}
$$

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 extensions to EGV for 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 payload types. We extend **raise** to take a term of type Exn as its argument. Finally, we generalise $\mathbf{try} L \mathbf{as} x \mathbf{in} M \mathbf{otherwise} N$ to $\mathbf{try} L \mathbf{as} x \mathbf{in} M \mathbf{unless} H$, 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 exception's payload matches its expected type. The TP-Raise and TP-Try are the natural extensions of T-Raise and T-Try.

Semantics. Our presentation is similar to operational accounts of effect handlers; 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 handled(E), which defines the exceptions handled in a given evaluation context:

$$
\begin{array}{ll}
\mathsf{handled}(P) = \emptyset & \mathsf{handled}(\mathbf{try}\ E\ \mathbf{as}\ x\ \mathbf{in}\ M\ \mathbf{unless}\ H) = \mathsf{handled}(E) \cup \mathsf{dom}(H) \\
\mathsf{handled}(E) = \mathsf{handled}(E'), & \text{if } E \text{ is not a } \mathbf{try} \text{ and } E' \text{ is the immediate subcontext of } E
\end{array}
$$

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. Due to the use of the handled function we no longer require pure contexts.

Syntax

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

Runtime Syntax

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

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

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

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

Term and Configuration Reduction $\boxed{M \longrightarrow_{\mathsf{M}} N}$ $\boxed{\mathcal{C} \longrightarrow \mathcal{D}}$

$$\begin{array}{llll}
\textsc{EP-Val} & \textbf{try}\ V\ \textbf{as}\ x\ \textbf{in}\ M\ \textbf{unless}\ H \longrightarrow_{\mathsf{M}} & M\{V/x\} & \\
\textsc{EP-Raise} & & & \\
 & \mathcal{F}[\textbf{try}\ E[\textbf{raise}\ X(V)]\ \textbf{as}\ x\ \textbf{in}\ M\ \textbf{unless}\ H] \longrightarrow & \mathcal{F}[N\{V/y\}] \parallel \lightning E & \text{where } X \notin \mathsf{handled}(E) \\
 & & & (X(y) \mapsto N) \in H \\
\textsc{EP-RaiseChild} & \circ\, E[\textbf{raise}\ X(V)] \longrightarrow & \lightning E \parallel \lightning V & \text{where } X \notin \mathsf{handled}(E) \\
\textsc{EP-RaiseMain} & \bullet\, E[\textbf{raise}\ X(V)] \longrightarrow & \textbf{halt} \parallel \lightning E \parallel \lightning V & \text{where } X \notin \mathsf{handled}(E)
\end{array}$$

Fig. 8. User-defined Exceptions with Payloads

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 because some data is naturally multi-use, and access points because they admit cyclic communication topologies supporting racey stateful servers such as chat servers. *Access points* decouple spawning a thread from establishing a session. An access point has the unrestricted type $\mathsf{AP}(S)$; we write $\mathsf{un}(A)$ to mean that A is unrestricted and $\mathsf{un}(\Gamma)$ if $\mathsf{un}(A_i)$ for all $x_i : A_i \in \Gamma$. Figure 9 shows the syntax, typing rules, and reduction rules for EGV extended with access points.

Unrestricted Types. To support unrestricted types, we introduce a splitting judgement ($\Gamma = \Gamma_1 + \Gamma_2$), which allows variables of unrestricted type to be shared across sub-environments, but requires linear variables to be used only in a single sub-environment. We relax rule T-Var to allow the use of unrestricted environments, and adapt all rules containing multiple subterms to use the splitting judgement. We detail T-App in the figure; the adaptations of other rules are similar. While unrestricted types are useful in general, we show the specific case of unrestricted access points.

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

$$\textbf{fork}\ M \triangleq \textbf{let}\ ap = \textbf{new}_S\ \textbf{in}\ \textbf{spawn}\ (M\ (\textbf{accept}\ ap));\ \textbf{request}\ ap$$

Syntax

Types	$A ::= \cdots \mid \mathsf{AP}(S)$
Access Point Names	z
Terms	$M ::= \cdots \mid z \mid \textbf{spawn}\ M \mid \textbf{new}_S \mid \textbf{request}\ M \mid \textbf{accept}\ M$
Configurations	$\mathcal{C} ::= \cdots \mid (\nu z)\mathcal{C} \mid z(\mathcal{X}, \mathcal{Y})$
Runtime typing environments	$\Delta ::= \cdots \mid \Delta, z : S$

Splitting $\boxed{\Gamma = \Gamma_1 + \Gamma_2}$

$$\frac{}{\cdot = \cdot + \cdot} \qquad \frac{\mathsf{un}(A)}{\Gamma, x : A = (\Gamma_1, x : A) + (\Gamma_2, x : A)} \qquad \frac{\Gamma = \Gamma_1 + \Gamma_2}{\Gamma, x : A = (\Gamma_1, x : A) + \Gamma_2} \qquad \frac{\Gamma = \Gamma_1 + \Gamma_2}{\Gamma, x : A = \Gamma_1 + (\Gamma_2, x : A)}$$

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

$$\textsc{T-Var}\ \frac{x : A \in \Gamma \qquad \mathsf{un}(\Gamma)}{\Gamma \vdash x : A} \qquad \textsc{T-App}\ \frac{\Gamma = \Gamma_1 + \Gamma_2 \qquad \Gamma_1 \vdash M : A \multimap B \qquad \Gamma_2 \vdash N : A}{\Gamma \vdash M\, N : B} \qquad \cdots$$

$$\textsc{TA-Spawn}\ \frac{\Gamma \vdash M : \mathbf{1}}{\Gamma \vdash \textbf{spawn}\ M : \mathbf{1}} \qquad \textsc{TA-New}\ \frac{}{\Gamma \vdash \textbf{new}_S : \mathsf{AP}(S)} \qquad \textsc{TA-Request}\ \frac{\Gamma \vdash M : \mathsf{AP}(S)}{\Gamma \vdash \textbf{request}\ M : \overline{S}} \qquad \textsc{TA-Accept}\ \frac{\Gamma \vdash M : \mathsf{AP}(S)}{\Gamma \vdash \textbf{accept}\ M : S}$$

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

E-Spawn	$\mathcal{F}[\textbf{spawn}\ M]$	$\longrightarrow$	$\mathcal{F}[()] \parallel \circ M$	
E-New	$\mathcal{F}[\textbf{new}_S]$	$\longrightarrow$	$(\nu z)(\mathcal{F}[z] \parallel z(\epsilon, \epsilon))$	z is fresh
E-Accept	$\mathcal{F}[\textbf{accept}\ z] \parallel z(\mathcal{X}, \mathcal{Y})$	$\longrightarrow$	$(\nu a)(\mathcal{F}[a] \parallel z(\{a\} \cup \mathcal{X}, \mathcal{Y}))$	a is fresh
E-Request	$\mathcal{F}[\textbf{request}\ z] \parallel z(\mathcal{X}, \mathcal{Y})$	$\longrightarrow$	$(\nu a)(\mathcal{F}[a] \parallel z(\mathcal{X}, \{a\} \cup \mathcal{Y}))$	a is fresh
E-Match	$z(\{a\} \cup \mathcal{X}, \{b\} \cup \mathcal{Y})$	$\longrightarrow$	$z(\mathcal{X}, \mathcal{Y}) \parallel a(\epsilon) \leftrightsquigarrow b(\epsilon)$	

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

$$\textsc{TA-ApName}\ \frac{\Gamma, z : \mathsf{AP}(S); \Delta, z : S \vdash^{\phi} \mathcal{C}}{\Gamma; \Delta \vdash^{\phi} (\nu z)\mathcal{C}} \qquad \textsc{TA-Ap}\ \frac{\mathsf{un}(\Gamma)}{\Gamma, z : \mathsf{AP}(S); \mathcal{X} : \overline{S}, \mathcal{Y} : S, z : S \vdash^{\circ} z(\mathcal{X}, \mathcal{Y})}$$

$$\textsc{TA-ConnectN}\ \frac{\Gamma = \Gamma_1 + \Gamma_2 \qquad \Gamma_1, \overrightarrow{a : S}; \Delta_1, \overrightarrow{b : \overline{T}} \vdash^{\phi_1} \mathcal{C} \qquad \Gamma_2, \overrightarrow{b : T}; \Delta_2, \overrightarrow{a : \overline{S}} \vdash^{\phi_2} \mathcal{D}}{\Gamma; \Delta_1, \Delta_2, \overrightarrow{a : S^{\sharp}}, \overrightarrow{b : T^{\sharp}} \vdash \mathcal{C} \parallel \mathcal{D}}$$

Fig. 9. Access Points

Reduction rules. We let z range over access point names. Configuration $(\nu z)\mathcal{C}$ denotes binding access point name z in $\mathcal{C}$, and $z(\mathcal{X}, \mathcal{Y})$ is an access point with name z and two sets $\mathcal{X}$ and $\mathcal{Y}$ containing endpoints to be matched.

Rule E-Spawn creates a new child thread but, unlike **fork**, returns the unit value instead of creating a channel and returning an endpoint. Rule E-New creates a new access point with fresh name z. Rules E-Accept and E-Request create a fresh name a, returning the newly-created name to the thread, and adding the name to sets $\mathcal{X}$ and $\mathcal{Y}$ respectively. Rule E-Match matches two endpoints a and b contained in $\mathcal{X}$ and $\mathcal{Y}$, and creates an empty buffer $a(\epsilon) \leftrightsquigarrow b(\epsilon)$.

Configuration typing. Configuration typing judgements again have the shape $\Gamma; \Delta \vdash^{\phi} \mathcal{C}$. Whereas Γ may contain unrestricted variables, Δ remains entirely linear.

Read bottom-up, rule TA-ApName adds an unrestricted reference $z : \mathsf{AP}(S)$ to Γ, and a linear entry $z : S$ to Δ. Rule TA-Ap types an access point configuration. We write $\mathcal{X} : S$ for $a_1 : S, \ldots, a_n : S$, where $\mathcal{X} = \{a_1, \ldots, a_n\}$. For an access point $z(\mathcal{X}, \mathcal{Y})$ to be well-typed, Δ must contain $z : S$, along with the names in $\mathcal{X}$ having type $\overline{S}$ and the names in $\mathcal{Y}$ having type S. Rule T-ConnectN generalises T-Connect$_1$ and T-Connect$_2$ to allow any number of channels to communicate across a buffer; this therefore introduces the possibility of deadlock.

Interaction with cancellation. We need no additional reduction rules to account for interaction between access points and channel cancellation. Should an endpoint waiting to be matched be cancelled, it is paired as usual, and interaction with its associated buffer raises an exception:

$$\begin{aligned} \lightning a \parallel \mathcal{F}[\textbf{receive}\ b] \parallel z(\{a\}, \{b\}) &\Longrightarrow \lightning a \parallel \mathcal{F}[\textbf{receive}\ b] \parallel z(\epsilon, \epsilon) \parallel a(\epsilon)\leftrightsquigarrow b(\epsilon) \\ &\Longrightarrow \lightning a \parallel \mathcal{F}[\textbf{raise}] \parallel \lightning b \parallel z(\epsilon, \epsilon) \parallel a(\epsilon)\leftrightsquigarrow b(\epsilon) \end{aligned}$$

Metatheory. By decoupling process and channel creation we lose the guarantee that the communication topology is acyclic, and therefore introduce the possibility of deadlock. Preservation continues to hold—in fact, we gain a stronger preservation result since the use of TA-ConnectN allows typeability to be preserved by equivalences.

Theorem 4.1 (Preservation Modulo Equivalence (Access Points)). *If $\Psi; \Delta \vdash^{\phi} \mathcal{C}$ and $\mathcal{C} \Longrightarrow \mathcal{D}$, then there exist Ψ', Δ' such that $\Psi; \Delta \longrightarrow \Psi'; \Delta'$ and $\Psi'; \Delta' \vdash^{\phi} \mathcal{D}$.*

Proof. By induction on the derivation of $\mathcal{C} \longrightarrow \mathcal{D}$ and preservation by $\equiv$; see Appendix D. □

Alas, the introduction of cyclic topologies and therefore the loss of deadlock-freedom necessarily violates global progress. Nevertheless, a weaker form of progress still holds: if a configuration does not reduce, then it is due to deadlock rather than cancellation.

Theorem 4.2 (Progress (Access Points)). *Suppose $\cdot; \cdot \vdash^{\phi} \mathcal{C}$ and $\mathcal{C} \not\Longrightarrow$. Then each thread in $\mathcal{C}$ is either a value; a buffer; a zapper thread; an access point; requesting or accepting on an access point; or ready to perform a communication action.*

If $\mathcal{C}$ contains a thread ϕM and $\mathsf{ready}(a, M)$ for some name a, then $\mathcal{C}$ contains some buffer $a(\epsilon)\leftrightsquigarrow b(\overrightarrow{W})$, and $\mathcal{C}$ does not contain a zapper thread $\lightning b$.

Proof. We can prove a similar property for open configurations by induction on the derivation of $\Psi; \Delta \vdash^{\phi} \mathcal{C}$; the above result arises as a corollary and by inspection of the reduction rules. □

In the presence of access points confluence and termination no longer hold: access points are nondeterministic and can encode higher-order state and hence fixpoints via Landin's knot.

4.3 Recursive Session Types

Recursive session types support repeating protocols. The extension of EGV with recursive session types is standard [Lindley and Morris 2016, 2017] and orthogonal to the main ideas of this paper, so we do not spell out the details here. The implementation (§5) does provide recursive session types.

5 SESSION TYPES WITHOUT TIERS

In this section we describe our extensions to Links to support exception handling, 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 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 language. Lindley and Morris [2017] extend Links with first-class session types, relying on lightweight linear typing [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 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] invoked a fresh copy of the server per server request and communication between client and server was via RPC calls. 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.

Links now adopts a persistent application server model, incorporating client-server communication 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.

5.2 Concurrency

Links provides typed actor-style concurrency where processes have a single incoming message queue and can send asynchronous messages. Lindley and Morris [2017] extend 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.

5.3 Distributed Communication

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 after the connection is established. We use a JSON protocol to communicate messages such as access point operations, remote session messages, and endpoint cancellation notifications.

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 new connection is established; 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 in §2 and §5.4.

Distributed Delegation. It is possible to send endpoints as part of a message. Session delegation in the presence of distributed communication requires some care to ensure that messages are delivered to the correct participant; our implementation adapts the algorithms of Hu et al. [2008]. Further details can be found in Appendix E.

5.4 Session Typing with Failure Handling

Effect Handlers. Effect handlers [Plotkin and Pretnar 2013] provide a modular approach to programming with user-defined effects. Exception handlers are a special case of effect handlers. 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 has access not only to a payload, but also 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. We translate exception handling as follows.

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

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 $\vec{p}\ r \mapsto N$, specifying how to handle an operation analogously to how exception handler clauses specify how to handle an exception, except that as well as binding payload parameters $\vec{p}$, an operation clause also binds a *resumption* parameter r. The resumption r binds a closure representing 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 raise 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). A formalisation of linear effect handlers for session typing is outside the scope of this paper and left as future work.

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 peer endpoint has been cancelled. Thus, we know statically where any exceptions may be raised. 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 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.

5.5 Distributed Exceptions

Our implementation fully supports the semantics described in §2. The concurrency runtime at each location maintains a set of cancelled endpoints.

Cancellation. Suppose endpoint a is connected to peer endpoint b. If a is cancelled, then all endpoints in the queue for a are also cancelled according to the E-Zap rule. If a and b are at the same location, then a is added to the set of cancelled endpoints. If they are at different locations, then a cancellation notification for a is routed to b's location. Zapper threads are modelled in the implementation by recording sets of cancelled endpoints and propagating cancellation messages.

Failed communications. Again, suppose endpoint a is connected to peer endpoint b. Should a process attempt to read from a when the buffer for a is empty, then the runtime will check to see whether b is in the set of cancelled endpoints. If so, then a is cancelled and an exception is raised in the blocked process; if not, the process is suspended until a message is ready. Should the runtime later add b to the set of cancelled endpoints, then again a is cancelled and an exception raised. These actions implement the E-ReceiveZap rule.

Disconnection. To handle disconnection, the server maintains a map from client IDs to the list of endpoints at the associated client. WebSockets—much like TCP sockets—raise a *closed* event on disconnection. Consequently, when a connection is closed, the runtime looks up the endpoints owned by the terminated client and notifies all other clients containing the peer endpoints.

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. We write the following informal 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.

```
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;
```

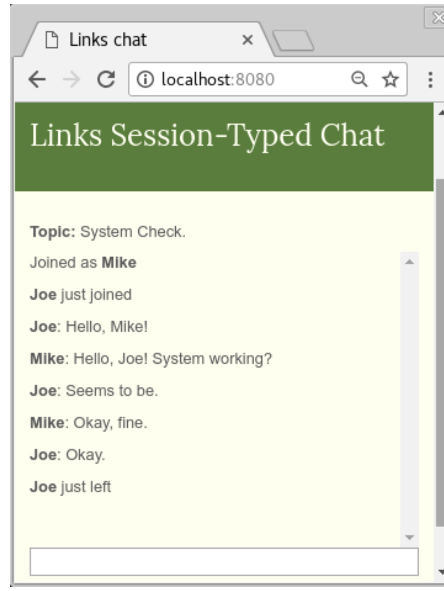


Fig. 10. Chat Application Session Types

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).

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.

How can session types help? The `connect` function (Fig. 11a) is run when a client enters a nickname. First, the client requests a fresh channel of type `ChatClient` from access point `wap` of type `AP(ChatServer)`. Next, the client obtains the content of the DOM input box for the nickname by calling `getInputContents(nameBoxId)`, where `nameBoxId` is the DOM ID for the nickname entry box. Next, the client sends the nickname to the server and waits for a response; in the case of a `Join` message, the client receives the room data and an incoming message channel, and calls the `beginChat` function. In the case of a `Nope` message, an error is printed and the session ends.

Now, suppose the developer forgets to write code to check the server response (Fig. 11b). This implementation is incorrect since there is a *communication mismatch*: the server is expecting to

```
fun connect() {
  var s = request(wap);
  var nick = getInputContents(nameBoxId);
  var s = send(nick, s);
  offer(s) {
    case Join(s) ->
      var ((topic, nicks, incoming), s) =
        receive(s);
      beginChat(topic, nicks, incoming, s)
    case Nope(s) ->
      print("Nickname '" ^^ nick ^^ "' already taken")
  }
}
```

(a) Correct `connect` function

```
fun connect() {
  var s = request(wap);
  var nick = getInputContents(nameBoxId);
  var s = send(nick, s);
  var ((topic, nicks, incoming), s) =
    receive(s);
  beginChat(topic, nicks, incoming, s)
}
```

(b) Incorrect `connect` function

Fig. 11. Implementations of `connect` function

accept or reject the request to join the room, whereas the client is expecting to receive data about the room. However, since `s` has type `ChatClient` but does not follow the protocol, Links catches the communication mismatch statically. Similarly, Links will statically detect an unused endpoint (e.g. the developer forgets to finish a protocol) or an endpoint being used more than once, as in §1.2.

Architecture. Figure 12a depicts the architecture of the chat 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. 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 makes essential use of access points.

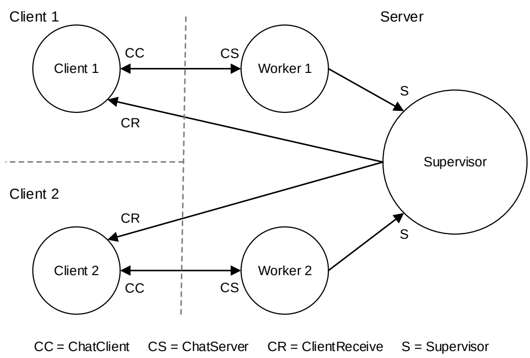
Disconnection. Figure 12b shows the implementation of a worker process which receives messages from a client. The worker takes the nickname of the client, as well as a channel endpoint of type `WorkerReceive` (which is the dual of `ClientSend`). The server offers the client a choice of sending a message (`Chat`), or changing topic (`NewTopic`); in each case, the associated data is received and a message dispatched to the supervisor process by calling `chat` or `newTopic`. 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. To handle disconnection, we wrap the function in an exception handler, which recursively calls `worker` if the interaction is successful, and notifies the supervisor that the user has left via a call to `leave` if an exception is raised.

Additional examples. We have concentrated on the chat server example for exposition, but have also implemented an extended chat server and a multiplayer game. These can be found at http://www.github.com/SimonJF/distributed-links-examples.

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.



(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. 12. Chat Application Architecture and Worker Implementation

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 non-determinism 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, whereas they opt for an approach more closely resembling checked exceptions, aided by a monadic presentation.

The above 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 [2014, 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. 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 detects duplicate endpoint use 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, providing 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, but since affine endpoints can be discarded implicitly, their library does not guarantee progress. Although it is not possible to distinguish between dynamic failure and a developer forgetting to finish an implementation, our semantics can be implemented using Rust's destructor mechanism, enabling a progress property [Kokke 2018].

8 CONCLUSION AND FUTURE WORK

Session types allow protocol conformance to be checked statically. The prevailing consensus has hitherto been to require that endpoints be used linearly to enforce session fidelity and 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 an extension of the Links programming language to support distributed session-based communication for web applications, thus providing the first implementation of asynchronous session types with failure handling in a functional programming language. Our implementation leverages recent work on effect handlers.

Future work. Our implementation combines linearity and effect handlers. Linear effect handlers are new, and a ripe area of study in their own right; we plan to formalise session-typed concurrency and failure handling directly in terms of linear effect handlers. Multiparty session types [Honda et al. 2016] are yet to be included as a first-class construct of a core functional language. A natural starting point is to identify a λ-calculus into which we can translate the MCP calculus of Carbone et al. [2016] and then investigate how our approach adapts to the multiparty setting.

ACKNOWLEDGMENTS

Thanks to James McKinna and the anonymous reviewers for detailed comments and suggestions. This work was supported by EPSRC grants EP/L01503X/1 (EPSRC CDT in Pervasive Parallelism) and EP/K034413/1 (From Data Types to Session Types—A Basis for Concurrency and Distribution), and an LFCS internship.

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.

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

Luís Caires and Frank Pfenning. 2010. Session types as intuitionistic linear propositions. In *CONCUR*, Vol. 10. Springer, 222–236.

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.

Wen Kokke. 2018. `rusty-variation`: a library for deadlock-free session-typed communication in Rust. https://github.com/wenkokke/rusty-variation. (2018).

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. 2014. Multiparty session actors. In *COORDINATION (Lecture Notes in Computer Science)*, Vol. 8459. Springer, 131–146.

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 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.

APPENDIX CONTENTS

A EXAMPLE RUNTIME TYPING DERIVATION

We give an example derivation to illustrate how channels are introduced by name restrictions and then split into endpoints using the T-$\textsc{Connect}_i$ rules. We assume suitable encodings of linear booleans and integers using linear sums and products.

Let us assume we have derivations for:

$$\Gamma_1, a : \mathsf{!Int.End} \vdash^{\circ} E[\mathbf{send}\ 5\ a] : \mathbf{1} \qquad \Gamma_2, b : \mathsf{?Bool.?Int.End} \vdash E'[\mathbf{receive}\ b] : A \qquad \cdot \vdash \mathsf{true} : \mathsf{Bool}$$

We construct a derivation $\mathbf{D}$ of $(\nu a)(\nu b)(\circ E[\mathbf{send}\ 5\ a] \parallel (a(\epsilon) \leftrightsquigarrow b(\mathsf{true}) \parallel \bullet E'[\mathbf{receive}\ b]))$.

First let $\mathbf{D}_1$ be the following subderivation.

$$\frac{\mathsf{?Int.End}/\epsilon = \overline{\mathsf{!Bool.!Int.End}/\mathsf{Bool}} \qquad \overline{\cdot \vdash \epsilon : \epsilon} \qquad \cdot \vdash \mathsf{true} : \mathsf{Bool}}{\cdot; a : \mathsf{?Int.End},\ b : \mathsf{!Bool.!Int.End} \vdash^{\circ} a(\epsilon) \leftrightsquigarrow b(\mathsf{true})}\ \textsc{T-Buffer}$$

Then let $\mathbf{D}_2$ be the following subderiviation.

$$\textsc{T-Connect}_2\ \frac{\mathbf{D}_1 \qquad \dfrac{\Gamma_2,\ b : \mathsf{?Bool.?Int.End} \vdash E'[\mathbf{receive}\ b] : A}{\Gamma_2,\ b : \mathsf{?Bool.?Int.End}; \cdot \vdash^{\bullet} \bullet E'[\mathbf{receive}\ b]}\ \textsc{T-Main}}{\Gamma_2; a : \mathsf{?Int.End},\ b : (\mathsf{?Bool.?Int.End})^{\sharp} \vdash^{\bullet} a(\epsilon) \leftrightsquigarrow b(\mathsf{true}) \parallel \bullet E'[\mathbf{receive}\ b]}$$

The complete derivation $\mathbf{D}$ is as follows.

$$\dfrac{\dfrac{\dfrac{\textsc{T-Thread}\ \dfrac{\Gamma_1,\ a : \mathsf{!Int.End} \vdash^{\circ} E[\mathbf{send}\ 5\ a] : \mathbf{1}}{\Gamma_1,\ a : \mathsf{!Int.End}; \cdot \vdash^{\circ} E[\mathbf{send}\ 5\ a]} \qquad \mathbf{D}_2}{\Gamma_1, \Gamma_2; a : (\mathsf{!Int.End})^{\sharp},\ b : (\mathsf{?Bool.?Int.End})^{\sharp} \vdash^{\circ} \circ E[\mathbf{send}\ 5\ a] \parallel (a(\epsilon) \leftrightsquigarrow b(\mathsf{true}) \parallel \bullet E'[\mathbf{receive}\ b])}\ \textsc{T-Connect}_1}{\Gamma_1, \Gamma_2; a : (\mathsf{!Int.End})^{\sharp} \vdash^{\bullet} (\nu b)(\circ E[\mathbf{send}\ 5\ a] \parallel (a(\epsilon) \leftrightsquigarrow b(\mathsf{true}) \parallel \bullet E'[\mathbf{receive}\ b]))}\ \textsc{T-Nu}}{\Gamma_1, \Gamma_2; \cdot \vdash^{\bullet} \bullet(\nu a)(\nu b)(\circ E[\mathbf{send}\ 5\ a] \parallel (a(\epsilon) \leftrightsquigarrow b(\mathsf{true}) \parallel \bullet E'[\mathbf{receive}\ b]))}\ \textsc{T-Nu}$$

Let us read $\mathbf{D}$ bottom-upwards. The two instances of the T-$\textsc{Nu}$ rule introduce channels a and b into the runtime environment. The T-$\textsc{Connect}_1$ rule splits channel a into dual endpoints: on the left the endpoint a appears in the type environment and the sending thread; on the right the end point a appears in the runtime environment and the buffer. The T-$\textsc{Connect}_2$ rule splits channel b into dual endpoints: on the left the endpoint b appears in the runtime environment and the buffer; on the right the endpoint b appears in the type environment and the receiving thread.

B DEADLOCK-FREEDOM

Here we give a graph-theoretic account of deadlock-freedom in EGV, independent of our notion of progress, following Lindley and Morris [2015].

Due to the asynchronous semantics of EGV, sending on an endpoint and cancelling an endpoint reduce immediately. Deadlocks may therefore only occur when cycles occur receiving or closing an endpoint. We begin by classifying the notion of a *blocked thread*: that is, a thread which is blocked performing an action on some channel endpoint.

Definition B.1. We say that term M is *blocked on name a* if M is about to receive on or close a. Formally:

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

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

Definition B.2. Let C be a configuration 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 blocked on b, or if a depends on some name c which depends on b. Formally:

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

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

Definition B.3. 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 configurations are deadlock-free.

LEMMA B.4. *If* $\mathsf{depends}(a, b, C)$ *then* $a, b \in \mathsf{fn}(C)$.

PROOF. By induction on the definition of $\mathsf{depends}(a, b, C)$. □

THEOREM B.5. *If* $\Gamma; \Delta \vdash C$*, then* $\neg\mathsf{deadlocked}(C)$.

PROOF. By contradiction. Suppose $\mathsf{deadlocked}(C)$, that is:

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

Thus by Lemma B.4, $a, b \in \mathsf{fn}(\mathcal{D})$ and $b, a \in \mathsf{fn}(\mathcal{E})$. Then by Lemma 3.6, C must be ill-typed. □

Remark. We regard blocked threads as deadlocked only if there is a cyclic dependency. It is perfectly possible for a configuration to include blocked threads without there being a deadlock.

- Deadlock-free open terms can block on external communication along a free endpoint.
- Deadlock-free closed terms can block on communication along an endpoint that appears in the return value of a program. This also amounts to being blocked on external communication.

All blocked threads can be ruled out by restricting the type of a program to be free of both session types and function types (the latter is necessary as closures can capture endpoints).

C SUPPLEMENT TO SECTION 3 (METATHEORY OF EGV)

C.1 Preservation

In this section, we present proofs that typeability is preserved by configuration reduction.

C.1.1 Equivalence. We begin by describing the properties of configuration equivalence. As described in §3, typeability of configurations is *not* preserved by equivalence. Nonetheless, Lemma C.1 shows that only the associativity of parallel composition may cause a configuration to be ill-typed.

Lemma C.1. *If $\Gamma; \Delta \vdash^{\phi} \mathcal{C}$ and $\mathcal{C} \equiv \mathcal{D}$, where the derivation of $\mathcal{C} \equiv \mathcal{D}$ does not contain a use of the axiom for associativity, then $\Gamma; \Delta \vdash^{\phi} \mathcal{D}$.*

Proof. By induction on the derivation of $\mathcal{C} \equiv \mathcal{D}$, examining the equivalence in both directions to account for symmetry. We show that a typing derivation of the left-hand side of an equivalence rule implies the existence of the right-hand side, and vice versa.

That reflexivity, transitivity, and symmetry of the equivalence relation respect typing follows immediately because equality of typing derivations is an equivalence relation.

We make implicit use of the induction hypothesis.

Congruence rules

Case Name restriction

$$\frac{\mathcal{C} \equiv \mathcal{D}}{(\nu a)\mathcal{C} \equiv (\nu a)\mathcal{D}}$$

$$\frac{\Gamma; \Delta, a : S^{\sharp} \vdash^{\phi} \mathcal{C}}{\Gamma; \Delta \vdash^{\phi} (\nu a)\mathcal{C}} \iff \frac{\Gamma; \Delta, a : S^{\sharp} \vdash^{\phi} \mathcal{D}}{\Gamma; \Delta \vdash^{\phi} (\nu a)\mathcal{D}}$$

Case Parallel Composition

$$\frac{\mathcal{C} \equiv \mathcal{D}}{\mathcal{C} \parallel \mathcal{E} \equiv \mathcal{D} \parallel \mathcal{E}}$$

There are three subcases, based on whether the parallel composition arises from T-Connect$_1$, T-Connect$_2$, or T-Mix.

Subcase T-Mix

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

Subcase T-Connect$_1$

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

Subcase T-Connect$_2$

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

Equivalence Axioms

Case $C \parallel \mathcal{D} \equiv \mathcal{D} \parallel C$

There are three subcases, based on which rule is used for parallel composition.

Subcase T-Mix

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

Subcase T-Connect$_1$

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

Subcase T-Connect$_2$

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

Case $C \parallel (\nu a)\mathcal{D} \equiv (\nu a)(C \parallel \mathcal{D})$ if $a \notin \mathsf{fn}(C)$

There are again three subcases based on which parallel composition rule is used. The exact rule does not affect the discussion, so without loss of generality we assume this is T-Mix.

$$\frac{\Gamma_1; \Delta_1 \vdash^{\phi_1} C \qquad \dfrac{\Gamma_2; \Delta_2, a : S^\sharp \vdash^{\phi_2} \mathcal{D}}{\Gamma_2; \Delta_2 \vdash^{\phi_2} (\nu a)\mathcal{D}}}{\Gamma_1, \Gamma_2; \Delta_1, \Delta_2 \vdash^{\phi_1+\phi_2} C \parallel (\nu a)\mathcal{D}} \iff \frac{\dfrac{\Gamma_1; \Delta_1 \vdash^{\phi_1} C \qquad \Gamma_2; \Delta_2, a : S^\sharp \vdash^{\phi_2} \mathcal{D}}{\Gamma_1, \Gamma_2; \Delta_1, \Delta_2, a : S^\sharp \vdash^{\phi_1+\phi_2} C \parallel \mathcal{D}}}{\Gamma_1, \Gamma_2; \Delta_1, \Delta_2 \vdash^{\phi_1+\phi_2} (\nu a)(C \parallel \mathcal{D})}$$

In the left-to-right direction, that $\Gamma_1, \Gamma_2; \Delta_1, \Delta_2, a : S^\sharp$ is well-defined follows because $a \notin \mathsf{fn}(C)$.

Case $(\nu a)(\nu b)C \equiv (\nu b)(\nu a)C$

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

Case $a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W}) \equiv b(\overrightarrow{W}) \leftrightsquigarrow a(\overrightarrow{V})$

$$\frac{S/\overrightarrow{A} = \overline{T/\overrightarrow{B}} \qquad \Gamma_1 \vdash \overrightarrow{V} : \overrightarrow{A} \qquad \Gamma_2 \vdash \overrightarrow{W} : \overrightarrow{B}}{\Gamma_1, \Gamma_2; a : S, b : T \vdash^{\circ} a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})} \iff \frac{T/\overrightarrow{B} = \overline{S/\overrightarrow{A}} \qquad \Gamma_2 \vdash \overrightarrow{W} : \overrightarrow{B} \qquad \Gamma_1 \vdash \overrightarrow{V} : \overrightarrow{A}}{\Gamma_1, \Gamma_2; a : S, b : T \vdash^{\circ} b(\overrightarrow{W}) \leftrightsquigarrow a(\overrightarrow{V})}$$

The above holds because $S/\overrightarrow{A} = \overline{T/\overrightarrow{B}} \iff T/\overrightarrow{B} = \overline{S/\overrightarrow{A}}$:

$$
\begin{array}{l}
S/\overrightarrow{A} = \overline{T/\overrightarrow{B}} \\
\quad\iff \text{(duality)} \\
\overline{S/\overrightarrow{A}} = \overline{\overline{T/\overrightarrow{B}}} \\
\quad\iff \text{(duality is involutive)} \\
\overline{S/\overrightarrow{A}} = T/\overrightarrow{B} \\
\quad\iff \text{(equality is symmetric)} \\
T/\overrightarrow{B} = \overline{S/\overrightarrow{A}}
\end{array}
$$

Case $\circ() \parallel C \equiv C$

$$
\frac{\dfrac{\dfrac{}{\cdot \vdash () : \mathbf{1}}}{\cdot; \cdot \vdash^{\circ} \circ()} \qquad \Gamma; \Delta \vdash^{\phi} C}{\Gamma; \Delta \vdash^{\phi} \circ() \parallel C} \qquad \iff \qquad \Gamma; \Delta \vdash^{\phi} C
$$

Case $(\nu a)(\nu b)(\lightning\, a \parallel \lightning\, b \parallel a(\epsilon)\leftrightsquigarrow b(\epsilon)) \parallel C \equiv C$

$$
\frac{\dfrac{\dfrac{\dfrac{\dfrac{}{a : S; \cdot \vdash^{\circ} \lightning\, a} \quad \dfrac{\dfrac{}{b : T; \cdot \vdash^{\circ} \lightning\, b} \quad \dfrac{\overline{S}/\epsilon = \overline{\overline{T}/\epsilon} \quad \dfrac{}{\cdot \vdash \epsilon : \epsilon} \quad \dfrac{}{\cdot \vdash \epsilon : \epsilon}}{\cdot; a : \overline{S}, b : \overline{T} \vdash^{\circ} a(\epsilon)\leftrightsquigarrow b(\epsilon)}}{\cdot; a : \overline{S}, b : T^{\sharp} \vdash^{\circ} \lightning\, b \parallel a(\epsilon)\leftrightsquigarrow b(\epsilon)}}{\cdot; a : S^{\sharp}, b : T^{\sharp} \vdash^{\circ} \lightning\, a \parallel \lightning\, b \parallel a(\epsilon)\leftrightsquigarrow b(\epsilon)}}{\cdot; a : S^{\sharp} \vdash^{\circ} (\nu b)(\lightning\, a \parallel \lightning\, b \parallel a(\epsilon)\leftrightsquigarrow b(\epsilon))}}{\cdot; \cdot \vdash^{\circ} (\nu a)(\nu b)(\lightning\, a \parallel \lightning\, b \parallel a(\epsilon)\leftrightsquigarrow b(\epsilon))} \quad \Gamma; \Delta \vdash^{\phi} C}{\Gamma; \Delta \vdash^{\phi} (\nu a)(\nu b)(\lightning\, a \parallel \lightning\, b \parallel a(\epsilon)\leftrightsquigarrow b(\epsilon)) \parallel C} \qquad \iff \qquad \Gamma; \Delta \vdash^{\phi} C
$$

□

While it is true that re-associating parallel composition may cause a configuration to be ill-typed, Lemma C.2 shows that it is always possible to re-associate parallel composition either directly, or by first commuting two subconfigurations.

Lemma C.2 (Associativity).

- *If* $\Gamma; \Delta \vdash^{\phi} \mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E})$, *then either* $\Gamma; \Delta \vdash^{\phi} (\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E}$ *or* $\Gamma; \Delta \vdash^{\phi} (\mathcal{C} \parallel \mathcal{E}) \parallel \mathcal{D}$.
- *If* $\Gamma; \Delta \vdash^{\phi} (\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E}$, *then either* $\Gamma; \Delta \vdash^{\phi} \mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E})$ *or* $\Gamma; \Delta \vdash^{\phi} \mathcal{D} \parallel (\mathcal{C} \parallel \mathcal{E})$.

Proof. The cases where either parallel composition arises by T-Mix are unproblematic and can be re-associated without jeopardising typeability. Therefore, we concentrate on the cases where both compositions arise via T-Connect$_i$.

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

By the assumption that $\Gamma; \Delta \vdash^{\phi} C \parallel (\mathcal{D} \parallel \mathcal{E})$ we have that $\Gamma = \Gamma_1, \Gamma_2, \Gamma_3$, and $\Delta = \Delta_1, \Delta_2, \Delta_3, a : S^{\sharp}, b : T^{\sharp}$, and $\phi = \phi_1 + \phi_2 + \phi_3$. There are 8 cases, based on whether $a, b \in \mathsf{fn}(C)$ or $a, b \in \mathsf{fn}(\mathcal{D})$ (it cannot be the case that $a, b \in \mathsf{fn}(\mathcal{E})$, as $\mathcal{E}$ only occurs under a single parallel composition), and the exact dualisation (i.e., whether composition happens via T-Connect$_1$ or T-Connect$_2$).

Of these, we are only interested in the cases where the sharing of the names differs, as opposed to the dualisation. Thus, we consider the following two cases, where both compositions occur using T-Connect$_1$:

(1) $\Gamma_1, a : S; \Delta_1 \vdash^{\phi_1} C$, and $\Gamma_2, b : T; \Delta_2, a : \overline{S} \vdash^{\phi_2} \mathcal{D}$, and $\Gamma_3; \Delta_3, b : \overline{T} \vdash^{\phi_3} \mathcal{E}$
(2) $\Gamma_1, a : S; \Delta_1 \vdash^{\phi_1} C$, and $\Gamma_2, b : T; \Delta_2 \vdash^{\phi_2} \mathcal{D}$, and $\Gamma_3; \Delta_3, a : \overline{S}, b : \overline{T} \vdash^{\phi_3} \mathcal{E}$

Subcase $a \in \mathsf{fn}(C), a, b \in \mathcal{D}, b \in \mathcal{E}$

$$\dfrac{\Gamma_1, a : S; \Delta_1 \vdash^{\phi_1} C \qquad \dfrac{\Gamma_2, b : T; \Delta_2, a : \overline{S} \vdash^{\phi_2} \mathcal{D} \qquad \Gamma_3; \Delta_3, b : \overline{T} \vdash^{\phi_3} \mathcal{E}}{\Gamma_2, \Gamma_3; \Delta_2, \Delta_3, a : \overline{S}, b : T^{\sharp} \vdash^{\phi_2 + \phi_3} \mathcal{D} \parallel \mathcal{E}}}{\Gamma_1, \Gamma_2, \Gamma_3; \Delta_1, \Delta_2, \Delta_3, a : S^{\sharp}, b : T^{\sharp} \vdash^{\phi_1 + \phi_2 + \phi_3} C \parallel (\mathcal{D} \parallel \mathcal{E})}$$

As $\mathcal{D}$ contains both a and b, associativity does not alter the sharing of names and may be applied safely.

$$\dfrac{\dfrac{\Gamma_1, a : S; \Delta_1 \vdash^{\phi_1} C \qquad \Gamma_2, b : T; \Delta_2, a : \overline{S} \vdash^{\phi_2} \mathcal{D}}{\Gamma_1, \Gamma_2, b : T; \Delta_1, \Delta_2, a : S^{\sharp} \vdash^{\phi_1 + \phi_2} C \parallel \mathcal{D}} \qquad \Gamma_3; \Delta_3, b : \overline{T} \vdash^{\phi_3} \mathcal{E}}{\Gamma_1, \Gamma_2, \Gamma_3; \Delta_1, \Delta_2, \Delta_3, a : S^{\sharp}, b : T^{\sharp} \vdash^{\phi_1 + \phi_2 + \phi_3} (C \parallel \mathcal{D}) \parallel \mathcal{E}}$$

Subcase $a \in \mathsf{fn}(C); b \in \mathcal{D}; a, b \in \mathcal{E}$

$$\dfrac{\Gamma_1, a : S; \Delta_1 \vdash^{\phi_1} C \qquad \dfrac{\Gamma_2, b : T; \Delta_2 \vdash^{\phi_2} \mathcal{D} \qquad \Gamma_3; \Delta_3, a : \overline{S}, b : \overline{T} \vdash^{\phi_3} \mathcal{E}}{\Gamma_2, \Gamma_3; \Delta_2, \Delta_3, a : \overline{S}, b : T^{\sharp} \vdash^{\phi_2 + \phi_3} \mathcal{D} \parallel \mathcal{E}}}{\Gamma_1, \Gamma_2, \Gamma_3; \Delta_1, \Delta_2, \Delta_3, a : S^{\sharp}, b : T^{\sharp} \vdash^{\phi_1 + \phi_2 + \phi_3} C \parallel (\mathcal{D} \parallel \mathcal{E})}$$

Here, we may not apply associativity directly. But, we may first commute $\mathcal{D}$ and $\mathcal{E}$:

$$\dfrac{\Gamma_1, a : S; \Delta_1 \vdash^{\phi_1} C \qquad \dfrac{\Gamma_3; \Delta_3, a : \overline{S}, b : \overline{T} \vdash^{\phi_3} \mathcal{E} \qquad \Gamma_2, b : T; \Delta_2 \vdash^{\phi_2} \mathcal{D}}{\Gamma_2, \Gamma_3; \Delta_2, \Delta_3, a : \overline{S}, b : T^{\sharp} \vdash^{\phi_2 + \phi_3} \mathcal{E} \parallel \mathcal{D}}}{\Gamma_1, \Gamma_2, \Gamma_3; \Delta_1, \Delta_2, \Delta_3, a : S^{\sharp}, b : T^{\sharp} \vdash^{\phi_1 + \phi_2 + \phi_3} C \parallel (\mathcal{E} \parallel \mathcal{D})}$$

and from here we may safely re-associate to the left:

$$\dfrac{\dfrac{\Gamma_2, a : S; \Delta_2 \vdash^{\phi_1} C \qquad \Gamma_3; \Delta_3, a : \overline{S}, b : \overline{T} \vdash^{\phi_3} \mathcal{E}}{\Gamma_2, \Gamma_3; \Delta_2, \Delta_3, a : S^{\sharp}, b : \overline{T} \vdash^{\phi_1 + \phi_2} \mathcal{D} \parallel \mathcal{E}} \qquad \Gamma_3, b : T; \Delta_3 \vdash^{\phi_3} \mathcal{D}}{\Gamma_1, \Gamma_2, \Gamma_3; \Delta_1, \Delta_2, \Delta_3, a : S^{\sharp}, b : T^{\sharp} \vdash^{\phi_1 + \phi_2 + \phi_3} (C \parallel \mathcal{E}) \parallel \mathcal{D}}$$

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

(1) $\Gamma_1, a : S; \Delta_1 \vdash^{\phi_1} C$, and $\Gamma_2, b : T; \Delta_2, a : \overline{S} \vdash^{\phi_2} \mathcal{D}$, and $\Gamma_3; \Delta_3, b : \overline{T} \vdash^{\phi_3} \mathcal{E}$
(2) $\Gamma_1, a : S, b : T; \Delta_1 \vdash^{\phi_1} C$, and $\Gamma_2; \Delta_2, b : \overline{T} \vdash^{\phi_2} \mathcal{D}$, and $\Gamma_3; \Delta_3, a : \overline{S} \vdash^{\phi_3} \mathcal{E}$

Subcase $a \in C; a, b \in \mathcal{D}; b \in \mathcal{E}$

Assumption:

$$\dfrac{\dfrac{\Gamma_1, a : S; \Delta_1 \vdash^{\phi_1} \mathcal{C} \qquad \Gamma_2, b : T; \Delta_2, a : \overline{S} \vdash^{\phi_2} \mathcal{D}}{\Gamma_1, \Gamma_2, b : T; \Delta_1, \Delta_2, a : S^\sharp \vdash^{\phi_1+\phi_2} \mathcal{C} \parallel \mathcal{D}} \qquad \Gamma_3; \Delta_3, b : \overline{T} \vdash^{\phi_3} \mathcal{E}}{\Gamma_1, \Gamma_2, \Gamma_3; \Delta_1, \Delta_2, \Delta_3, a : S^\sharp, b : T^\sharp \vdash^{\phi_1+\phi_2+\phi_3} (\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E}}$$

Applying associativity here does not make the configuration ill-typed, as $\mathcal{D}$ contains both names:

$$\dfrac{\Gamma_1, a : S; \Delta_1 \vdash^{\phi_1} \mathcal{C} \qquad \dfrac{\Gamma_2, b : T; \Delta_2, a : \overline{S} \vdash^{\phi_2} \mathcal{D} \qquad \Gamma_3; \Delta_3, b : \overline{T} \vdash^{\phi_3} \mathcal{E}}{\Gamma_2, \Gamma_3; \Delta_2, \Delta_3, a : \overline{S}, b : T^\sharp \vdash^{\phi_2+\phi_3} \mathcal{D} \parallel \mathcal{E}}}{\Gamma_1, \Gamma_2, \Gamma_3; \Delta_1, \Delta_2, \Delta_3, a : S^\sharp, b : T^\sharp \vdash^{\phi_1+\phi_2+\phi_3} \mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E})}$$

Subcase $a, b \in \mathcal{C}; a \in \mathcal{D}; b \in \mathcal{E}$

Assumption:

$$\dfrac{\dfrac{\Gamma_1, a : S, b : T; \Delta_1 \vdash^{\phi_1} \mathcal{C} \qquad \Gamma_2; \Delta_2, a : \overline{S} \vdash^{\phi_2} \mathcal{D}}{\Gamma_2, \Gamma_3, b : T; \Delta_2, \Delta_3, a : S^\sharp \vdash^{\phi_2+\phi_3} \mathcal{C} \parallel \mathcal{D}} \qquad \Gamma_3; \Delta_3, b : \overline{T} \vdash^{\phi_3} \mathcal{E}}{\Gamma_1, \Gamma_2, \Gamma_3; \Delta_1, \Delta_2, \Delta_3, a : S^\sharp, b : T^\sharp \vdash^{\phi_1+\phi_2+\phi_3} (\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E}}$$

By commutativity:

$$\dfrac{\dfrac{\Gamma_2; \Delta_2, a : \overline{S} \vdash^{\phi_2} \mathcal{D} \qquad \Gamma_1, a : S, b : T; \Delta_1 \vdash^{\phi_1} \mathcal{C}}{\Gamma_2, \Gamma_3, b : T; \Delta_2, \Delta_3, a : S^\sharp \vdash^{\phi_2+\phi_1} \mathcal{D} \parallel \mathcal{C}} \qquad \Gamma_3; \Delta_3, b : \overline{T} \vdash^{\phi_3} \mathcal{E}}{\Gamma_1, \Gamma_2, \Gamma_3; \Delta_1, \Delta_2, \Delta_3, a : S^\sharp, b : T^\sharp \vdash^{\phi_1+\phi_2+\phi_3} (\mathcal{D} \parallel \mathcal{C}) \parallel \mathcal{E}}$$

By associativity:

$$\dfrac{\Gamma_2; \Delta_2, a : \overline{S} \vdash^{\phi_2} \mathcal{D} \qquad \dfrac{\Gamma_1, a : S, b : T; \Delta_1 \vdash^{\phi_1} \mathcal{C} \qquad \Gamma_3; \Delta_3, b : \overline{T} \vdash^{\phi_3} \mathcal{E}}{\Gamma_1, \Gamma_3, a : S; \Delta_1, \Delta_3, b : T^\sharp \vdash^{\phi_1+\phi_3} \mathcal{C} \parallel \mathcal{E}}}{\Gamma_1, \Gamma_2, \Gamma_3; \Delta_1, \Delta_2, \Delta_3, a : S^\sharp, b : T^\sharp \vdash^{\phi_1+\phi_2+\phi_3} \mathcal{D} \parallel (\mathcal{C} \parallel \mathcal{E})}$$

as required.

□

C.1.2 Configuration Reduction. We may now show that configuration reduction preserves typeability of configurations. We begin by stating some auxiliary results about substitution and evaluation contexts.

Typing of terms is preserved by substitution.

Lemma C.3 (Substitution). *If:*

(1) $\Gamma_1 \vdash M : B$
(2) $\Gamma_2, x : B \vdash N : A$
(3) Γ_1, Γ_2 *is well-defined*

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 C.4 shows that a subterm of a well-typed evaluation context E (and therefore also a pure evaluation context P) is typeable with a subset of the type environment. Lemma C.5 states that the subterm of a well-typed evaluation context can be replaced. Both follow the formulation of Gay and Vasconcelos [2010].

Lemma C.4 (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 C.5 (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
- $\Gamma_3 \vdash N : B$
- Γ_1, Γ_3 *is well-defined*

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

Proof. By induction on the structure of E. □

To prove preservation on configurations, we must first establish some auxiliary results on configuration contexts. Lemma C.6 states how we may type subconfigurations.

Lemma C.6 (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 C.7 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 C.7 (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* $C \longrightarrow \mathcal{D}$*, then there exist* Γ', Δ' *such that* $\Gamma; \Delta \longrightarrow^{?} \Gamma'; \Delta'$ *and* $\Gamma'; \Delta' \vdash^{\phi} \mathcal{D}$*.*

Proof. By induction on the derivation of $C \longrightarrow \mathcal{D}$. Where there is a choice of value for ϕ, we consider the case where $\phi = \bullet$; the cases where $\phi = \circ$ are similar.

Case E-Fork

Assumption:

$$\frac{\Gamma_1, \Gamma_2 \vdash \bullet E[\mathbf{fork}\, \lambda x.M] : A}{\Gamma_1, \Gamma_2; \cdot \vdash^{\bullet} \bullet E[\mathbf{fork}\, \lambda x.M]}$$

By Lemma C.4:

$$\dfrac{\dfrac{\Gamma_2, x : S \vdash M : \mathbf{1}}{\Gamma_2 \vdash \lambda x.M : S \multimap \mathbf{1}}}{\Gamma_2 \vdash \mathbf{fork}\ \lambda x.M : \overline{S}}$$

By Lemma C.3, $\Gamma_2, b : S \vdash M\{b/x\} : \mathbf{1}$, and by Lemma C.5, $\Gamma_1, a : \overline{S} \vdash E[a] : A$. As duality is involutive, $\overline{\overline{S}} = S$.

Reconstructing:

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

Case E-Send

Assumption:

$$\dfrac{\dfrac{\Gamma_1, \Gamma_2 \vdash E[\mathbf{send}\ U\ a] : C}{\Gamma_1, \Gamma_2, a : S; \cdot \vdash^{\bullet} \bullet E[\mathbf{send}\ U\ a]} \qquad \dfrac{\overline{S}/\overrightarrow{A} = \overline{T/\overrightarrow{B}} \qquad \Gamma_3 \vdash \overrightarrow{V} : \overrightarrow{A} \qquad \Gamma_4 \vdash \overrightarrow{W} : \overrightarrow{B}}{\Gamma_3, \Gamma_4; a : \overline{S}, b : T \vdash^{\circ} a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}}{\Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4; a : S^{\sharp}, b : T \vdash^{\bullet} \bullet E[\mathbf{send}\ U\ a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}$$

By Lemma C.4:

$$\dfrac{\Gamma_2 \vdash U : A \qquad a : {!A.S'} \vdash a : {!A.S'}}{\Gamma_2, a : {!A.S'} \vdash \mathbf{send}\ U\ a : S'}$$

Thus, $S = {!A.S'}$, and $\overline{S} = ?A.\overline{S'}$. We may therefore refine our original derivation:

$$\dfrac{\dfrac{\Gamma_1, \Gamma_2, a : {!A.S'} \vdash E[\mathbf{send}\ U\ a] : C}{\Gamma_1, \Gamma_2, a : {!A.S'}; \cdot \vdash^{\bullet} \bullet E[\mathbf{send}\ U\ a]} \qquad \dfrac{?A.\overline{S'}/\overrightarrow{A} = \overline{T/\overrightarrow{B}} \qquad \Gamma_3 \vdash \overrightarrow{V} : \overrightarrow{A} \qquad \Gamma_4 \vdash \overrightarrow{W} : \overrightarrow{B}}{\Gamma_3, \Gamma_4; a : ?A.\overline{S'}, b : T \vdash^{\circ} a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}}{\Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4; a : {!A.S'}^{\sharp}, b : T \vdash^{\bullet} \bullet E[\mathbf{send}\ U\ a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}$$

Since $?A.\overline{S'}/\overrightarrow{A} = \overline{T/\overrightarrow{B}}$ is well-defined, we have that $\overrightarrow{A} = \epsilon$. By the definition of slicing, we have that $\overline{T} = \overline{!B_1.\cdots.!B_n.!A.S'}$, where $\overrightarrow{B} = B_1, \ldots, B_n$. It follows that $\overline{S'}/\overrightarrow{A} = \overline{T/\overrightarrow{B} \cdot A}$.

By Lemma C.5, we have $\Gamma_1, \Gamma_2, a : S' \vdash E[a] : C$.

Reconstructing:

$$\dfrac{\dfrac{\Gamma_1, a : S' \vdash E[a] : C}{\Gamma_1, a : S'; \cdot \vdash^{\bullet} \bullet E[a]} \qquad \dfrac{\overline{S'}/\overrightarrow{A} = \overline{T/\overrightarrow{B} \cdot A} \qquad \Gamma_3 \vdash \overrightarrow{V} : \overrightarrow{A} \qquad \Gamma_2, \Gamma_4 \vdash \overrightarrow{W} \cdot U : \overrightarrow{B} \cdot A}{\Gamma_2, \Gamma_3, \Gamma_4; a : \overline{S'}, b : T \vdash^{\circ} a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W} \cdot U)}}{\Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4; a : S'^{\sharp}, b : T \vdash^{\bullet} \bullet E[a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W} \cdot U)}$$

Finally, we must show environment reduction:

$$\frac{!A.S' \longrightarrow S'}{\Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4; a : (!A.S')^\sharp, b : T \longrightarrow \Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4; a : S'^\sharp, b : T}$$

as required.

Case E-Receive

Assumption:

$$\dfrac{\dfrac{\Gamma_1, a : S \vdash E[\textbf{receive}\ a] : C}{\Gamma_1, a : S; \cdot \vdash^\bullet E[\textbf{receive}\ a]} \qquad \dfrac{\overline{S}/\overrightarrow{A} = \overline{T/\overrightarrow{B}} \qquad \Gamma_2, \Gamma_3 \vdash U \cdot \overrightarrow{V} : \overrightarrow{A} \qquad \Gamma_4 \vdash \overrightarrow{W} : \overrightarrow{B}}{\Gamma_2, \Gamma_3, \Gamma_4; a : \overline{S}, b : T \vdash^\circ a(U \cdot \overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}}{\Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4; a : S^\sharp, b : T \vdash^\bullet \bullet E[\textbf{receive}\ a] \parallel a(U \cdot \overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}$$

By Theorem C.4:

$$\frac{a : ?A.S' \vdash a : ?A.S'}{a : ?A.S' \vdash \textbf{receive}\ a : (A \times S')}$$

Thus, we have that $S = ?A.S'$ and $\overline{S} = !A.\overline{S'}$, and we may therefore refine the original typing derivation:

$$\dfrac{\dfrac{\Gamma_1, a : ?A.S' \vdash E[\textbf{receive}\ a] : C}{\Gamma_1, a : ?A.S'; \cdot \vdash^\bullet E[\textbf{receive}\ a]} \qquad \dfrac{!A.\overline{S'}/A \cdot \overrightarrow{A'} = \overline{T/\overrightarrow{B}} \qquad \dfrac{\Gamma_1 \vdash U : A \qquad \Gamma_3 \vdash \overrightarrow{V} : \overrightarrow{A'}}{\Gamma_2, \Gamma_3 \vdash U \cdot \overrightarrow{V} : A \cdot \overrightarrow{A'}} \qquad \Gamma_4 \vdash \overrightarrow{W} : \overrightarrow{B}}{\Gamma_2, \Gamma_3, \Gamma_4; a : !A.\overline{S'}, b : T \vdash^\circ a(U \cdot \overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}}{\Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4; a : (?A.S')^\sharp, b : T \vdash^\bullet \bullet E[\textbf{receive}\ a] \parallel a(U \cdot \overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}$$

By Lemma C.5, we have $\Gamma_1, \Gamma_2, a : S' \vdash E[(U, a)] : C$ (that Γ_1, Γ_2 is defined follows from the fact that Γ_1 and Γ_2 are sub-environments of the original typing environment and are therefore necessarily disjoint).

By the definition of slicing, $!A.\overline{S'}/A \cdot \overrightarrow{A'} \iff \overline{S'}/\overrightarrow{A'}$.

Thus, recomposing:

$$\dfrac{\dfrac{\Gamma_1, \Gamma_2, a : S' \vdash E[(U, a)] : C}{\Gamma_1, \Gamma_2, a : S'; \cdot \vdash^\bullet E[(U, a)]} \qquad \dfrac{\overline{S'}/\overrightarrow{A'} = \overline{T/\overrightarrow{B}} \qquad \Gamma_3 \vdash \overrightarrow{V} : \overrightarrow{A'} \qquad \Gamma_4 \vdash \overrightarrow{W} : \overrightarrow{B}}{\Gamma_3, \Gamma_4; a : \overline{S'}, b : T \vdash^\circ a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}}{\Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4; a : S'^\sharp, b : T \vdash^\bullet \bullet E[(U, a)] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}$$

Finally, we must show environment reduction:

$$\frac{?A.S' \longrightarrow S'}{\Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4; a : (?A.S'^\sharp); b : T \longrightarrow \Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4; a : S'^\sharp, b : T}$$

as required.

Case E-Close

Assumption:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma_1, a : S \vdash E[\mathbf{close}\ a] : C}{\Gamma_1, a : S; \cdot \vdash^{\bullet} \bullet E[\mathbf{close}\ a]} \quad \dfrac{\dfrac{\Gamma_2, b : T \vdash E'[\mathbf{close}\ b] : \mathbf{1}}{\Gamma_2, b : T; \cdot \vdash^{\circ} \circ E'[\mathbf{close}\ b]} \quad \dfrac{\overline{S}/\epsilon = \overline{T}/\epsilon \quad \dfrac{}{\cdot \vdash \epsilon : \epsilon} \quad \dfrac{}{\cdot \vdash \epsilon : \epsilon}}{\cdot; a : \overline{S}, b : \overline{T} \vdash^{\circ} a(\epsilon)\leftrightsquigarrow b(\epsilon)}}{\Gamma_2; a : \overline{S}, b : T^{\sharp} \vdash^{\circ} E'[\mathbf{close}\ b] \parallel a(\epsilon)\leftrightsquigarrow b(\epsilon)}}{\Gamma_1, \Gamma_2; a : S^{\sharp}, b : T^{\sharp} \vdash^{\bullet} \bullet E[\mathbf{close}\ a] \parallel \circ E'[\mathbf{close}\ b] \parallel a(\epsilon)\leftrightsquigarrow b(\epsilon)}}{\Gamma_1, \Gamma_2; a : S^{\sharp} \vdash^{\bullet} (\nu b)(\bullet E[\mathbf{close}\ a] \parallel \circ E'[\mathbf{close}\ b] \parallel a(\epsilon)\leftrightsquigarrow b(\epsilon))}}{\Gamma_1, \Gamma_2; \cdot \vdash^{\bullet} (\nu a)(\nu b)(\bullet E[\mathbf{close}\ a] \parallel \circ E'[\mathbf{close}\ b] \parallel a(\epsilon)\leftrightsquigarrow b(\epsilon))}$$

By Lemma C.4:

$$\dfrac{a : \mathsf{End} \vdash a : \mathsf{End}}{a : \mathsf{End} \vdash \mathbf{close}\ a : \mathbf{1}} \qquad \dfrac{b : \mathsf{End} \vdash b : \mathsf{End}}{b : \mathsf{End} \vdash \mathbf{close}\ b : \mathbf{1}}$$

By Lemma C.5, we have that $\Gamma_1 \vdash E[()] : C$ and that $\Gamma_2 \vdash E'[()] : \mathbf{1}$. Thus by T-Mix, we may show:

$$\dfrac{\dfrac{\Gamma_1 \vdash E[()] : C}{\Gamma_1; \cdot \vdash^{\bullet} \bullet E[()]} \qquad \dfrac{\Gamma_2 \vdash E[()] : \mathbf{1}}{\Gamma_2; \cdot \vdash^{\circ} \circ E[()]}}{\Gamma_1, \Gamma_2; \cdot \vdash^{\bullet} \bullet E[()] \parallel \circ E[()]}$$

as required.

Case E-Cancel

$$\mathcal{F}[\mathbf{cancel}\ a] \longrightarrow \mathcal{F}[()] \parallel \lightning a$$

Assumption:

$$\dfrac{\Gamma \vdash E[\mathbf{cancel}\ a] : C}{\Gamma; \cdot \vdash^{\bullet} \bullet E[\mathbf{cancel}\ a]}$$

By Lemma C.4, $\Gamma = \Gamma_1, \Gamma_2$, where

$$\dfrac{\Gamma_2 \vdash a : S}{\Gamma_2 \vdash \mathbf{cancel}\ a : \mathbf{1}}$$

Thus $\Gamma_2 = a : S$. By Lemma C.5, $\Gamma_1 \vdash E[()] : C$. By T-Zap, we have that $a : S \vdash^{\circ} \lightning a$. Thus, recomposing:

$$\dfrac{\dfrac{\Gamma \vdash E[()] : C}{\Gamma_1; \cdot \vdash^{\bullet} \bullet E[()]} \qquad \dfrac{}{a : S; \cdot \vdash^{\circ} \lightning a}}{\Gamma_1, a : S; \cdot \vdash^{\bullet} \bullet E[()] \parallel \lightning a}$$

as required.

Case E-Zap

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

where $\mathsf{fn}(U) = \{c_i\}_i$.

Assumption:

$$\dfrac{\dfrac{}{a : S; \cdot \vdash^\circ \lightning a} \qquad \dfrac{\overline{S}/\overrightarrow{A} = \overline{T/\overrightarrow{B}} \qquad \Gamma_1, \Gamma_2 \vdash U \cdot \overrightarrow{V} : \overrightarrow{A} \qquad \Gamma_3 \vdash \overrightarrow{W} : \overrightarrow{B}}{\Gamma_1, \Gamma_2, \Gamma_3; a : \overline{S}, b : T \vdash^\circ a(U \cdot \overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}}{\Gamma_1, \Gamma_2, \Gamma_3; a : S^\sharp, b : T \vdash^\circ \lightning a \parallel a(U \cdot \overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}$$

By the definition of slicing, we have that there exist some A and S' such that $\overline{S} = !A.\overline{S'}$. Thus, we may refine our judgement:

$$\dfrac{\dfrac{}{a : ?A.S'; \cdot \vdash^\circ \lightning a} \qquad \dfrac{!A.\overline{S'}/A \cdot \overrightarrow{A'} = \overline{T/\overrightarrow{B}} \qquad \Gamma_1, \Gamma_2 \vdash U \cdot \overrightarrow{V} : A \cdot \overrightarrow{A'} \qquad \Gamma_3 \vdash \overrightarrow{W} : \overrightarrow{B}}{\Gamma_1, \Gamma_2, \Gamma_3; a : \overline{S}, b : T \vdash^\circ a(U \cdot \overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}}{\Gamma_1, \Gamma_2, \Gamma_3; a : S^\sharp, b : T \vdash^\circ \lightning a \parallel a(U \cdot \overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}$$

By the definition of buffer typing, we have that $\Gamma_1 \vdash U : A$. By the definition of the reduction rule, $\mathsf{fn}(U) = \{c_i\}_i$, and by assumption, Γ contains only runtime names. Thus, we may conclude that U is closed and therefore that $\Gamma_1 = c_1 : S_1, \ldots c_n : S_n$ for some session types $S_1, \ldots S_n$.

By the definition of slicing, we have that $!A.\overline{S'}/A \cdot \overrightarrow{A'} \iff \overline{S'}/\overrightarrow{A'}$. Correspondingly, by T-Buffer, we may show

$$\dfrac{\overline{S'}/\overrightarrow{A'} = \overline{T/\overrightarrow{B}} \qquad \Gamma_2 \vdash \overrightarrow{V} : \overrightarrow{A'} \qquad \Gamma_3 \vdash \overrightarrow{W} : \overrightarrow{B}}{\Gamma_2, \Gamma_3; a : \overline{S'}, b : T \vdash^\circ a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}$$

By repeated applications of T-Zap and T-Mix, we have that

$$\Gamma_2, \Gamma_3, c_1 : S_1, \ldots, c_n : S_n; a : \overline{S'}, b : T \vdash^\circ \lightning c_1 \parallel \cdots \parallel \lightning c_n \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$$

Recomposing:

$$\dfrac{\dfrac{}{a : S'; \cdot \vdash^\circ \lightning a} \qquad \dfrac{\dfrac{}{c_1 : S_1; \cdot \vdash^\circ \lightning c_1} \qquad \dfrac{\dfrac{}{c_n : S_n; \cdot \vdash^\circ \lightning c_n} \qquad \dfrac{\overline{S'}/\overrightarrow{A'} = \overline{T/\overrightarrow{B}} \qquad \Gamma_2 \vdash \overrightarrow{V} : \overrightarrow{A'} \qquad \Gamma_3 \vdash \overrightarrow{W} : \overrightarrow{B}}{\Gamma_2, \Gamma_3; a : \overline{S'}, b : T \vdash^\circ a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}}{\vdots}}{\Gamma_2, \Gamma_3, c_1 : S_1, \ldots, c_n : S_n; a : \overline{S'}, b : T \vdash^\circ \lightning c_1 \parallel \ldots \parallel \lightning c_n \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}}{\Gamma_2, \Gamma_3, c_1 : S_1, \ldots, c_n : S_n; a : S'^\sharp, b : T \vdash^\circ \lightning a \parallel \lightning c_1 \parallel \ldots \parallel \lightning c_n \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}$$

Finally, we must show environment reduction:

$$\dfrac{?A.S' \longrightarrow S'}{\Gamma_2, \Gamma_3, c_1 : S_1, \ldots, c_n : S_n; a : (?A.S'^\sharp), b : T \longrightarrow \Gamma_2, \Gamma_3, c_1 : S_1, \ldots, c_n : S_n; a : S'^\sharp, b : T}$$

as required.

Case E-CloseZap

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

Assumption:

$$\dfrac{\dfrac{\Gamma, a : S \vdash E[\mathbf{close}\ a] : C}{\Gamma, a : S; \cdot \vdash^{\bullet} \bullet E[\mathbf{close}\ a]} \quad \dfrac{\dfrac{}{b : T; \cdot \vdash^{\circ} \lightning b} \quad \dfrac{\overline{S} = \overline{\overline{T}} \quad \dfrac{}{\cdot \vdash \epsilon : \epsilon} \quad \dfrac{}{\cdot \vdash \epsilon : \epsilon}}{\cdot; a : \overline{S}, b : \overline{T} \vdash^{\circ} a(\epsilon)\leftrightsquigarrow b(\epsilon)}}{\cdot; a : \overline{S}, b : T^{\sharp} \vdash^{\circ} \lightning b \parallel a(\epsilon)\leftrightsquigarrow b(\epsilon)}}{\Gamma; a : S^{\sharp}, b : T^{\sharp} \vdash^{\bullet} \bullet E[\mathbf{close}\ a] \parallel \lightning b \parallel a(\epsilon)\leftrightsquigarrow b(\epsilon)}$$

By Lemma C.4:

$$\dfrac{a : \mathsf{End} \vdash a : \mathsf{End}}{a : S \vdash \mathbf{close}\ a : \mathbf{1}}$$

We may therefore refine our original derivation:

$$\dfrac{\dfrac{\Gamma, a : \mathsf{End} \vdash E[\mathbf{close}\ a] : C}{\Gamma, a : \mathsf{End}; \cdot \vdash^{\bullet} \bullet E[\mathbf{close}\ a]} \quad \dfrac{\dfrac{}{b : \mathsf{End}; \cdot \vdash^{\circ} \lightning b} \quad \dfrac{\mathsf{End} = \mathsf{End} \quad \dfrac{}{\cdot \vdash \epsilon : \epsilon} \quad \dfrac{}{\cdot \vdash \epsilon : \epsilon}}{\cdot; a : \mathsf{End}, b : \mathsf{End} \vdash^{\circ} a(\epsilon)\leftrightsquigarrow b(\epsilon)}}{\cdot; a : \mathsf{End}, b : \mathsf{End}^{\sharp} \vdash^{\circ} \lightning b \parallel a(\epsilon)\leftrightsquigarrow b(\epsilon)}}{\Gamma; a : \mathsf{End}^{\sharp}, b : \mathsf{End}^{\sharp} \vdash^{\bullet} \bullet E[\mathbf{close}\ a] \parallel \lightning b \parallel a(\epsilon)\leftrightsquigarrow b(\epsilon)}$$

By Lemma C.5, $\Gamma \vdash E[\mathbf{raise}] : C$.
Thus, recomposing:

$$\dfrac{\dfrac{\Gamma \vdash E[\mathbf{raise}] : C}{\Gamma \vdash^{\bullet} \bullet E[\mathbf{raise}]} \quad \dfrac{\dfrac{}{a : \mathsf{End}; \cdot \vdash^{\circ} \lightning a} \quad \dfrac{\dfrac{}{b : \mathsf{End}; \cdot \vdash^{\circ} \lightning b} \quad \dfrac{\mathsf{End} = \mathsf{End} \quad \dfrac{}{\cdot \vdash \epsilon : \epsilon} \quad \dfrac{}{\cdot \vdash \epsilon : \epsilon}}{\cdot; a : \mathsf{End}, b : \mathsf{End} \vdash^{\circ} a(\epsilon)\leftrightsquigarrow b(\epsilon)}}{\cdot; a : \mathsf{End}, b : \mathsf{End}^{\sharp} \vdash^{\circ} \lightning b \parallel a(\epsilon)\leftrightsquigarrow b(\epsilon)}}{\cdot; a : \mathsf{End}^{\sharp}, b : \mathsf{End}^{\sharp} \vdash^{\circ} \lightning a \parallel \lightning b \parallel a(\epsilon)\leftrightsquigarrow b(\epsilon)}}{\Gamma; a : \mathsf{End}^{\sharp}, b : \mathsf{End}^{\sharp} \vdash^{\bullet} \bullet E[\mathbf{close}\ a] \parallel \lightning b \parallel a(\epsilon)\leftrightsquigarrow b(\epsilon)}$$

as required.

Case E-ReceiveZap

$$\bullet E[\mathbf{receive}\ a] \parallel \lightning b \parallel a(\epsilon)\leftrightsquigarrow b(\overrightarrow{W}) \longrightarrow \bullet E[\mathbf{raise}] \parallel \lightning a \parallel \lightning b \parallel a(\epsilon)\leftrightsquigarrow b(\overrightarrow{W})$$

Assumption:

$$\dfrac{\dfrac{\Gamma_1, a : S \vdash E[\mathbf{receive}\ a] : C}{\Gamma_1, a : S; \cdot \vdash^{\bullet} \bullet E[\mathbf{receive}\ a]} \quad \dfrac{\dfrac{}{b : T; \cdot \vdash^{\circ} \lightning b} \quad \dfrac{\overline{S}/\epsilon = \overline{\overline{T}/\overrightarrow{B}} \quad \dfrac{}{\cdot \vdash \epsilon : \epsilon} \quad \Gamma_2 \vdash \overrightarrow{W} : \overrightarrow{B}}{\Gamma_2; a : \overline{S}, b : \overline{T} \vdash^{\circ} a(\epsilon)\leftrightsquigarrow b(\overrightarrow{W})}}{\Gamma_2; a : \overline{S}, b : T^{\sharp} \vdash^{\circ} \lightning b \parallel a(\epsilon)\leftrightsquigarrow b(\overrightarrow{W})}}{\Gamma_1, \Gamma_2; a : S^{\sharp}, b : T^{\sharp} \vdash^{\bullet} \bullet E[\mathbf{receive}\ a] \parallel \lightning b \parallel a(\epsilon)\leftrightsquigarrow b(\overrightarrow{W})}$$

By Lemma C.4:

$$\dfrac{a : ?A.S' \vdash a : ?A.S'}{a : ?A.S' \vdash \mathbf{receive}\ a : (A \times S')}$$

By Lemma C.5, $\Gamma_1 \vdash E[\mathbf{raise}] : S'$. Thus, recomposing:

$$\dfrac{\dfrac{\Gamma_1 \vdash E[\mathbf{raise}] : C}{\Gamma_1; \cdot \vdash^\bullet \bullet E[\mathbf{raise}]} \quad \dfrac{\overline{a : S; \cdot \vdash \lightning a} \quad \dfrac{\overline{b : T; \cdot \vdash^\circ \lightning b} \quad \dfrac{\overline{\overline{S}/\epsilon = \overline{T}/\overrightarrow{B}} \quad \overline{\cdot \vdash \epsilon : \epsilon} \quad \Gamma_2 \vdash \overrightarrow{W} : \overrightarrow{B}}{\Gamma_2; a : \overline{S}, b : \overline{T} \vdash^\circ a(\epsilon) \leftrightsquigarrow b(\overrightarrow{W})}}{\Gamma_2; a : \overline{S}, b : T^\sharp \vdash^\circ \lightning b \parallel a(\epsilon) \leftrightsquigarrow b(\overrightarrow{W})}}{\Gamma_2; a : S^\sharp, b : T^\sharp \vdash^\circ \lightning a \parallel \lightning b \parallel a(\epsilon) \leftrightsquigarrow b(\overrightarrow{W})}}{\Gamma_1, \Gamma_2; a : S^\sharp, b : T^\sharp \vdash^\bullet \bullet E[\mathbf{raise}] \parallel \lightning a \parallel \lightning b \parallel a(\epsilon) \leftrightsquigarrow b(\overrightarrow{W})}$$

as required.

Case E-Raise

$$\bullet E[\mathbf{try}\ P[\mathbf{raise}]\ \mathbf{as}\ x\ \mathbf{in}\ M\ \mathbf{otherwise}\ N] \longrightarrow E[N] \parallel \lightning c_1 \parallel \cdots \parallel \lightning c_n$$

and $\mathsf{fn}(P) = \{c_i\}_i$.

Assumption:

$$\dfrac{\Gamma \vdash E[\mathbf{try}\ P[\mathbf{raise}]\ \mathbf{as}\ x\ \mathbf{in}\ M\ \mathbf{otherwise}\ N] : A'}{\Gamma; \cdot \vdash^\bullet \bullet E[\mathbf{try}\ P[\mathbf{raise}]\ \mathbf{as}\ x\ \mathbf{in}\ M\ \mathbf{otherwise}\ N]}$$

By Lemma C.4, there exist $\Gamma_1, \Gamma_2, A, B, C$ such that $\Gamma = \Gamma_1, \Gamma_2, \Gamma_3$ and

$$\dfrac{\Gamma_2 \vdash P[\mathbf{raise}] : A \qquad \Gamma_3, x : B \vdash M : C \qquad \Gamma_3 \vdash N : C}{\Gamma_2, \Gamma_3 \vdash \mathbf{try}\ P[\mathbf{raise}]\ \mathbf{as}\ x\ \mathbf{in}\ M\ \mathbf{otherwise}\ N : C}$$

Since Γ contains only runtime names and $\mathsf{fn}(P) = \{c_i\}_i$, we know that $\Gamma_2 = c_1 : S_1, \ldots, c_n : S_n$ for some S_i.

By Lemma C.5, we have that:

$$\overline{\Gamma_1, \Gamma_3 \vdash E[N] : A'}$$

By repeated applications of T-Zap and T-Mix, we have that $\Gamma_2 \vdash \lightning c_1 \parallel \cdots \parallel \lightning c_n$.

Therefore, recomposing:

$$\dfrac{\dfrac{\Gamma_1, \Gamma_3 \vdash E[N] : C}{\Gamma_1, \Gamma_3; \cdot \vdash^\bullet \bullet E[N]} \quad \dfrac{\overline{c_1 : S_1; \cdot \vdash^\circ \lightning c_1} \quad \dfrac{\overline{c_{n-1} : S_{n-1}; \cdot \vdash^\circ \lightning c_{n-1}} \quad \overline{c_n : S_n; \cdot \vdash^\circ \lightning c_n}}{\vdots}}{c_1 : S_1, \ldots, c_n : S_n; \cdot \vdash^\circ \lightning c_1 \parallel \cdots \parallel \lightning c_n}}{\Gamma_1, \Gamma_3, c_1 : S_1, \ldots, c_n : S_n; \cdot \vdash^\bullet \bullet E[N] \parallel \lightning c_1 \parallel \cdots \parallel \lightning c_n}$$

as required.

Case E-RaiseChild

$$\circ P[\mathbf{raise}] \longrightarrow \lightning c_1 \parallel \cdots \parallel \lightning c_n$$

Assumption:

$$\dfrac{\Gamma \vdash P[\mathbf{raise}] : \mathbf{1}}{\Gamma; \cdot \vdash^\circ \circ P[\mathbf{raise}]}$$

By Lemma C.4, the knowledge that Γ contains only runtime names, the knowledge that $\mathsf{fn}(P) = c_1, \ldots, c_n$, and the typing rule T-Raise, we have that $\Gamma = c_1 : S_1, \ldots, c_n : S_n$ for some session types $\{S_i\}_i$.

Thus, by repeated applications of T-Zap and T-Mix, we may deduce that

$$\Gamma; \cdot \vdash^{\circ} \lightning c_1 \parallel \cdots \parallel \lightning c_n$$

as required.

Case E-RaiseMain

$$\bullet P[\mathbf{raise}] \longrightarrow \mathbf{halt} \parallel \lightning c_1 \parallel \cdots \parallel \lightning c_n$$

where $\mathsf{fn}(P) = \{c_i\}_i$.

Assumption:

$$\frac{\Gamma \vdash P[\mathbf{raise}] : C}{\Gamma; \cdot \vdash^{\bullet} \bullet P[\mathbf{raise}]}$$

By Lemma C.4, the knowledge that Γ contains only runtime names, the knowledge that $\mathsf{fn}(P) = c_1, \ldots, c_n$, and the typing rule T-Raise, we have that $\Gamma = c_1 : S_1, \ldots, c_n : S_n$ for some session types $\{S_i\}_i$.

By repeated applications of T-Zap and T-Mix, we may deduce that

$$\Gamma; \cdot \vdash^{\circ} \lightning c_1 \parallel \cdots \parallel \lightning c_n$$

By T-Halt, we have that $\cdot; \cdot \vdash^{\bullet} \mathbf{halt}$. Thus, recomposing, we arrive at

$$\frac{\dfrac{}{\cdot; \cdot \vdash^{\bullet} \mathbf{halt}} \quad \dfrac{\dfrac{}{c_1 : S_1; \cdot \vdash^{\circ} \lightning c_1} \quad \dfrac{\dfrac{}{c_{n-1} : S_{n-1}; \cdot \vdash^{\circ} \lightning c_{n-1}} \quad \dfrac{}{c_n : S_n; \cdot \vdash^{\circ} \lightning c_n}}{\vdots}}{c_1 : S_1, \ldots, c_n : S_n; \cdot \vdash^{\circ} \lightning c_1 \parallel \cdots \parallel \lightning c_n}}{\Gamma_1, \Gamma_3, c_1 : S_1, \ldots, c_n : S_n; \cdot \vdash^{\bullet} \mathbf{halt} \parallel \lightning c_1 \parallel \cdots \parallel \lightning c_n}$$

as required.

Case LiftC

Assumptions:

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

Let $\mathbf{D}$ be a derivation of $\Gamma; \Delta \vdash^{\phi} \mathcal{G}[C]$. By Lemma C.6, we have that there exists some $\mathbf{D}'$ such that $\mathbf{D}'$ is a subderivation of $\mathbf{D}$ concluding $\Gamma'; \Delta' \vdash^{\phi'} C$, where the position of $\mathbf{D}'$ in $\mathbf{D}$ corresponds to that of the hole in $\mathcal{G}$.

By the IH, we have that there exists some $\Gamma''; \Delta''$ such that $\Gamma; \Delta \longrightarrow^{?} \Gamma''; \Delta''$ and $\Gamma''; \Delta'' \vdash^{\phi} \mathcal{D}$.

By Lemma C.7, we have that there exist some $\Gamma'''; \Delta'''$ such that $\Gamma; \Delta \longrightarrow^{?} \Gamma'''; \Delta'''$ and $\Gamma'''; \Delta''' \vdash^{\phi} \mathcal{G}[\mathcal{D}]$, as required.

Case E-LiftM

Assumptions:

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

and $M \longrightarrow_{\mathsf{M}} N$. By Lemma 3.1, we have that $\Gamma \vdash N : A$. Recomposing:

$$\frac{\Gamma \vdash N : A}{\Gamma; \cdot \vdash^{\bullet} \bullet N}$$

as required. □

C.2 Canonical Forms

Theorem 3.7: 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 $n = 0$, by Lemma C.1 we can safely commute the main thread such that it is the rightmost configuration, and associate parallel composition to the right using Lemma C.2 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 3.6 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. □

C.3 Progress

To prove Theorem 3.9, we prove a similar property in which canonical configurations are decomposed step-by-step rather than in one go.

Definition C.8 (Open Progress). Suppose $\Psi; \Delta \vdash^{\bullet} C$, where C is in canonical form and $C \not\Longrightarrow$. We say that C *satisfies open progress* if:

(1) $C = (\nu a)(\mathcal{A} \parallel \mathcal{D})$, where $\Psi = \Psi_1, \Psi_2$ and $\Delta = \Delta_1, \Delta_2$ such that either:
  (a) $\Psi_1, a : S; \Delta_1 \vdash^{\circ} \mathcal{A}$ and $\Psi_2; \Delta_2, a : \overline{S} \vdash^{\bullet} \mathcal{D}$ where $\mathcal{D}$ satisfies open progress, and $\mathcal{A}$ is either:
    (i) A thread $\circ M$ where $\mathsf{ready}(b, M)$ for some $b \in \mathsf{fn}(\Psi_1, a : S)$; or
    (ii) A zapper thread $⚡a$; or
    (iii) A buffer $b(\overrightarrow{V}) \leftrightsquigarrow c(\overrightarrow{W})$ where $b, c \neq a$ and either $a \in \overrightarrow{V}$ or $a \in \overrightarrow{W}$
  (b) $\Psi_1; \Delta_1, a : \overline{S} \vdash^{\circ} \mathcal{A}$ and $\Psi_2, a : S; \Delta_2 \vdash^{\bullet} \mathcal{D}$, where $\mathcal{D}$ satisfies open progress, and $\mathcal{A}$ is either $a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$ or $b(\overrightarrow{V}) \leftrightsquigarrow a(\overrightarrow{W})$ for some $b \in \mathsf{fn}(\Delta_1)$
(2) $C = \mathcal{A} \parallel \mathcal{M}$, where $\Psi = \Psi_1, \Psi_2$ and either:
  (a) $\Delta = \Delta_1, \Delta_2, a : S^{\sharp}$, where $\Psi_1, a : S; \Delta_1 \vdash^{\circ} \mathcal{A}$ and $\Psi_2; \Delta_2, a : \overline{S} \vdash^{\bullet} \mathcal{M}$, where $\mathcal{M}$ satisfies open progress, and $\mathcal{A}$ is either:
    (i) A thread $\circ M$ where $\mathsf{ready}(b, M)$ for some $b \in \mathsf{fn}(\Psi_1, a : S)$; or
    (ii) A zapper thread $⚡a$; or
    (iii) A buffer $b(\overrightarrow{V}) \leftrightsquigarrow c(\overrightarrow{W})$ where $b, c \neq a$ and either $a \in \mathsf{fn}(\overrightarrow{V})$ or $a \in \mathsf{fn}(\overrightarrow{W})$
  (b) $\Delta = \Delta_1, \Delta_2, a : S^{\sharp}$, where $\Psi_1; \Delta_1, a : \overline{S} \vdash^{\circ} \mathcal{A}$ and $\Psi_2, a : S; \Delta_2 \vdash^{\bullet} \mathcal{M}$, where $\mathcal{M}$ satisfies open progress, and $\mathcal{A}$ is either $a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$ or $b(\overrightarrow{V}) \leftrightsquigarrow a(\overrightarrow{W})$ for some $b \in \mathsf{fn}(\Delta_1)$

(c) $\Delta = \Delta_1, \Delta_2$, where $\Psi_1; \Delta_1 \vdash^{\circ} \mathcal{A}$ and $\Psi_2; \Delta_2 \vdash^{\bullet} \mathcal{M}$, where $\mathcal{M}$ satisfies open progress, and $\mathcal{A}$ is either:
   (i) A thread $\circ M$ where either $M = ()$, or $\mathsf{ready}(a, M)$ for some $a \in \mathsf{fn}(\Psi_1)$; or
   (ii) A zapper thread $\lightning a$ for some $a \in \mathsf{fn}(\Psi_1)$; or
   (iii) A buffer $a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$ for some $a, b \in \mathsf{fn}(\Delta_1)$

(3) $C = \mathcal{T}$, where either:
   (a) $\mathcal{T} = \bullet N$, where N is either a value or $\mathsf{ready}(b, N)$ for some $b \in \mathsf{fn}(\Psi)$
   (b) $\mathcal{T} = \mathbf{halt}$

Lemma C.9. *Suppose $\Psi; \Delta \vdash^{\bullet} C$, where C is in canonical form and $C \not\Longrightarrow$. Then C satisfies open progress.*

Proof. By induction on the derivation of $\Psi; \Delta \vdash^{\bullet} C$. We have three cases, based on the structure of the given canonical form.

Case $C = (\nu a)(\mathcal{A} \parallel \mathcal{D})$, with $a \in \mathsf{fn}(\mathcal{A})$, and where $\mathcal{D}$ is in canonical form

By assumption, we know that $\Psi; \Delta \vdash^{\phi} (\nu a)(\mathcal{A} \parallel \mathcal{D})$.

This configuration is typeable by T-Nu, followed by either T-Connect$_1$ or T-Connect$_2$. As the definition of canonical forms requires that $a \in \mathsf{fn}(\mathcal{A})$, it cannot be the case that the parallel composition arises as a result of T-Mix.

We consider these two subcases to show that $\mathcal{A}$ satisfies the properties required by open progress.

Subcase T-Connect$_1$

$$\dfrac{\dfrac{\Psi_1, a : S; \Delta_1 \vdash^{\phi_1} \mathcal{A} \qquad \Psi_2; \Delta_2, a : \overline{S} \vdash^{\phi_2} \mathcal{D}}{\Psi_1, \Psi_2; \Delta_1, \Delta_2, a : S^{\sharp} \vdash^{\phi_1 + \phi_2} \mathcal{A} \parallel \mathcal{D}}}{\Psi_1, \Psi_2; \Delta_1, \Delta_2 \vdash^{\phi_1 + \phi_2} (\nu a)(\mathcal{A} \parallel \mathcal{D})}$$

By the definition of auxiliary threads and inversion on the typing relation, we know that $\mathcal{A}$ is of the following forms:

- $\circ M$, where $a \in \mathsf{fn}(M)$, and $\Psi_1, a : S \vdash M : \mathbf{1}$
- $\lightning a$
- $b(\overrightarrow{V}) \leftrightsquigarrow c(\overrightarrow{W})$, where $b, c \in \mathsf{fn}(\Delta_1)$ and $a \in \mathsf{fn}(V)$
- $b(\overrightarrow{V}) \leftrightsquigarrow c(\overrightarrow{W})$, where $b, c \in \mathsf{fn}(\Delta_1)$ and $a \in \mathsf{fn}(W)$

(since $a \notin \Delta_1$, it cannot be the case that a appears as a buffer endpoint).

Lemma 3.4 tells us that either there exists some M' such that $M \longrightarrow_{\mathsf{M}} M'$; that M is a value; or that M is a communication and concurrency construct. Since $C \not\Longrightarrow$, we have that M is unable to reduce (as otherwise C could reduce by E-LiftM). Since $a \in \mathsf{fn}(M)$ and a does not have type $\mathbf{1}$, it cannot be the case that M is a value.

Therefore, we have that M has the form $E[N]$, where N is a communication / concurrency construct. This cannot be **fork**, since **fork** may always reduce by E-Fork, so there must exist some $b \in \mathsf{fn}(\Psi, a : S)$ such that $\mathsf{ready}(b, M)$.

Subcase T-Connect$_2$

$$\dfrac{\dfrac{\Psi_1; \Delta_1, a : \overline{S} \vdash^{\circ} \mathcal{A} \qquad \Psi_2, a : S; \Delta_2 \vdash^{\bullet} \mathcal{D}}{\Psi_1, \Psi_2; \Delta_1, \Delta_2, a : S^{\sharp} \vdash^{\bullet} \mathcal{A} \parallel \mathcal{D}}}{\Psi_1, \Psi_2; \Delta_1, \Delta_2 \vdash^{\bullet} (\nu a)(\mathcal{A} \parallel \mathcal{D})}$$

By the definition of auxiliary threads and inversion on the typing relation, we know that $\mathcal{A}$ is of the following forms:

- $a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$, where $b \in \Delta_1$
- $b(\overrightarrow{V}) \leftrightsquigarrow a(\overrightarrow{W})$, where $b \in \Delta_1$

(as $a \in \mathsf{fn}(\mathcal{A})$ and $a \in \Delta_1$, it cannot be the case that $\mathcal{A}$ is a child thread or a zapper thread, as these require empty runtime typing environments).

By the induction hypothesis, we know that $\mathcal{D}$ satisfies open progress; hence $(\nu a)(\mathcal{A} \parallel \mathcal{D})$ satisfies open progress.

Case $C = \mathcal{A} \parallel \mathcal{M}$

There are three subcases, based on whether the parallel composition arises as a result of T-Connect$_1$, T-Connect$_2$, or T-Mix.

Subcase T-Connect$_1$

$$\frac{\Psi_1, a : S; \Delta_1 \vdash^{\circ} \mathcal{A} \qquad \Psi_2; \Delta_2, a : \overline{S} \vdash^{\bullet} \mathcal{M}}{\Psi_1, \Psi_2; \Delta_1, \Delta_2, a : S^{\sharp} \vdash^{\bullet} \mathcal{A} \parallel \mathcal{M}}$$

By inversion on the typing rules, we have that $\mathcal{A}$ may be:

- A child thread $\circ M$, where $a \in \mathsf{fn}(M)$
- A zapper thread $\lightning a$
- A buffer $b(\overrightarrow{V}) \leftrightsquigarrow c(\overrightarrow{W})$, where $b, c \neq a$ and either $a \in \mathsf{fn}(\overrightarrow{V})$ or $a \in \mathsf{fn}(\overrightarrow{W})$

In the case of (1), by Lemma 3.4, we have that either M is a value; there exists N such that $M \longrightarrow_{\mathsf{M}} N$; or $M = E[N]$ for some E, N, where N is a communication / concurrency construct.

By T-Child, $\Psi_1, a : S \vdash M : \mathbf{1}$. Since $a \in \mathsf{fn}(M)$ and the only value with type $\mathbf{1}$ is the unit value () it therefore cannot be the case that M is a value. Since $C \not\Longrightarrow$, it cannot be the case that $M \longrightarrow_{\mathsf{M}} N$, since otherwise C could reduce. Thus, it must be the case that $M = E[N]$ where N is a communication and concurrency construct; by similar reasoning as above cases, it therefore must be the case that $\mathsf{ready}(b, M)$ for some $b \in \mathsf{fn}(\Psi_1, a : S)$.

(2) and (3) satisfy the required conditions by definition.

Subcase T-Connect$_2$

$$\frac{\Psi_1; \Delta_1, a : \overline{S} \vdash^{\circ} \mathcal{A}; \Psi_2, a : S; \Delta_2 \vdash^{\bullet} \mathcal{M}}{\Psi_1, \Psi_2; \Delta_1, \Delta_2, a : S^{\sharp} \vdash^{\bullet} \mathcal{A} \parallel \mathcal{M}}$$

Since the runtime typing environment $\Delta_1, a : \overline{S}$ is non-empty, it cannot be the case that $\mathcal{A}$ is a child thread or zapper thread. Thus, $\mathcal{A}$ must either be of the form:

(1) $a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$, where $a, b \in \Delta_1$; or
(2) $b(\overrightarrow{V}) \leftrightsquigarrow a(\overrightarrow{W})$, where $a, b \in \Delta_1$

which satisfy the required conditions by definition.

Subcase T-Mix

$$\frac{\Psi_1; \Delta_1 \vdash^{\circ} \mathcal{A} \qquad \Psi_2; \Delta_2 \vdash^{\bullet} \mathcal{M}}{\Psi_1, \Psi_2; \Delta_1, \Delta_2 \vdash^{\bullet} \mathcal{A} \parallel \mathcal{M}}$$

By inversion on the typing rules, we have that $\mathcal{A}$ may either be:

(1) A child thread $\circ M$
(2) A zapper thread $\lightning a$ for some $a \in \mathsf{fn}(\Psi_1)$

(3) A buffer thread $a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$ for some $a, b \in \mathsf{fn}(\Delta_1)$

By Lemma 3.4, we have that M is either a value V; there exists some N such that $M \longrightarrow_{\mathsf{M}} N$; or $M = E[N]$ for some E, N such that N is a communication and concurrency primitive. It cannot be the case that $M \longrightarrow_{\mathsf{M}} N$ since otherwise the configuration could reduce.

By T-Child, it must be the case that $\Psi_1; \Delta_1 \vdash M : \mathbf{1}$; if M is a value then by inversion on the term typing rules, it must be the case that $M = ()$.

Following the same reasoning as previous cases, if $M = E[N]$ for some communication / concurrency primitive N, it must be that $\mathsf{ready}(a, M)$ for some $a \in \Psi_1$.

By the induction hypothesis, we know that $\mathcal{M}$ satisfies open progress; hence $\mathcal{A} \parallel \mathcal{M}$ satisfies open progress.

Case $C = \mathcal{T}$

Assumption: $\Psi; \Delta \vdash^{\bullet} \mathcal{T}$. By the definition of $\mathcal{T}$, we have two subcases:

Subcase $\mathcal{T} = \bullet M$

$$\frac{\Psi \vdash M : A}{\Psi; \cdot \vdash^{\bullet} \bullet M}$$

By Lemma 3.4, we have that either M is a value; that there exists some N such that $M \longrightarrow_{\mathsf{M}} N$; or that there exist some E, N such that $M = E[N]$ where N is a communication / concurrency primitive.

Again, as $C \not\Longrightarrow$, it cannot be the case that $M \longrightarrow_{\mathsf{M}} N$, since otherwise C could reduce. If M is a value, then $\mathcal{T}$ satisfies open progress.

Finally, if $M = E[N]$ where N is a communication / concurrency primitive, it cannot be the case that $N = \mathbf{fork}\, M'$ since it could reduce by T-Fork, and so it must be the case that $\mathsf{ready}(a, M)$ for some $a \in \Psi$, satisfying open progress, as required.

Subcase $\mathcal{T} = \mathbf{halt}$

Immediate by the definition of open progress.

□

Theorem 3.9 provides a more global and concise view of the properties exhibited by a non-reducing process in canonical form, and arises as an immediate corollary.

Theorem 3.9 *Suppose $\Psi; \Delta \vdash^{\bullet} C$ where C is in canonical form and $C \not\Longrightarrow$.*

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

Either there exists some C' such that $C \Longrightarrow C'$, or:

(1) *For $1 \leq i \leq n$, each thread in $\mathcal{A}_i$ is either:*
   (a) *a child thread $\circ M$ for which there exists $a \in \{a_j \mid 1 \leq j \leq i\} \cup \mathsf{fn}(\Psi)$ such that $\mathsf{ready}(a, M)$;*
   (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 $1 \leq j \leq m$:*
   (a) *$\mathcal{A}'_j$ is either:*
      (i) *a child thread $\circ N$ such that $N = ()$ or $\mathsf{ready}(a, N)$ for some $a \in \{a_i \mid 1 \leq i \leq n\} \cup \mathsf{fn}(\Psi) \cup \mathsf{fn}(\Delta)$;*
      (ii) *a zapper thread $\lightning a$ for some $a \in \{a_i \mid 1 \leq i \leq n\} \cup \mathsf{fn}(\Psi) \cup \mathsf{fn}(\Delta)$; or*
      (iii) *a buffer.*
   (b) *Either $\mathcal{T} = \bullet N$, where N is either a value or $\mathsf{ready}(a, N)$ for some $a \in \{a_i \mid 1 \leq i \leq n\} \cup \mathsf{fn}(\Psi) \cup \mathsf{fn}(\Delta)$; or $\mathcal{T} = \mathbf{halt}$.*

C.4 Confluence

Theorem 3.12 (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. As noted in Section 3.4, $\longrightarrow_{\mathsf{M}}$ is deterministic and hence confluent due to the setup of term evaluation contexts, and linearity ensures that endpoints to a buffer may not be shared. Consequently, 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.

$$
\begin{tikzcd}[column sep=-3em]
& \mathcal{F}[\mathbf{send}\;U\;a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(V \cdot \overrightarrow{W}) \parallel \mathcal{F}'[\mathbf{receive}\;b] \arrow[dl, Rightarrow] \arrow[dr, Rightarrow] & \\
\mathcal{F}[a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(V \cdot \overrightarrow{W} \cdot U) \parallel \mathcal{F}'[\mathbf{receive}\;b] \arrow[dr, Rightarrow] & & \mathcal{F}[\mathbf{send}\;U\;a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W}) \parallel \mathcal{F}'[(V, b)] \arrow[dl, Rightarrow] \\
& \mathcal{F}[a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W} \cdot U) \parallel \mathcal{F}'[(V, b)] &
\end{tikzcd}
$$

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 as to 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.

$$
\begin{tikzcd}[column sep=-3em]
& \mathcal{F}[\mathbf{send}\;U\;a] \parallel \lightning b \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(V \cdot \overrightarrow{W}) \arrow[dl, Rightarrow] \arrow[dr, Rightarrow] & \\
\mathcal{F}[a] \parallel \lightning b \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(V \cdot \overrightarrow{W} \cdot U) \arrow[dr, Rightarrow] & & \mathcal{F}[\mathbf{send}\;U\;a] \parallel \lightning b \parallel \lightning V \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W}) \arrow[dl, Rightarrow] \\
& \mathcal{F}[a] \parallel \lightning b \parallel \lightning V \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W} \cdot U) &
\end{tikzcd}
$$

□

D SUPPLEMENT TO SECTION 4.1 (METATHEORY OF EGV WITH ACCESS POINTS)

In this section, we prove that the extension of EGV with access points satisfies preservation.

Lemma D.1 (Preservation, Access Points (Equivalence)). *If $\Gamma; \Delta \vdash^{\phi} \mathcal{C}$ and $\mathcal{C} \equiv \mathcal{D}$, then $\Gamma; \Delta \vdash^{\phi} \mathcal{D}$*

Proof. By induction on the derivation of $\mathcal{C} \equiv \mathcal{D}$. Rule T-ConnectN subsumes T-Connect$_1$ and T-Connect$_2$, so the majority of cases are similar to those we have proven in Lemma C.1. We consider the case for associativity in detail.

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

$$\dfrac{\begin{array}{c}\Gamma = \Gamma_1 + \Gamma' \\ \Gamma_1, \overrightarrow{a : S}; \Delta_1, \overrightarrow{b : \overline{T}} \vdash^{\phi_1} \mathcal{C} \qquad \dfrac{\begin{array}{c}\Gamma' = \Gamma_2 + \Gamma_3 \\ \Gamma_2, b_1 : T_1, \ldots, b_{m'} : T_{m'}, \overrightarrow{c : S'}; \Delta_2, a_1 : \overline{S_1}, \ldots, a_m : \overline{S_m}, \overrightarrow{d : \overline{T'}} \vdash^{\phi_2} \mathcal{D} \\ \Gamma_3, b_{m'+1}, \ldots, b_{n'} : T_{n'}, \overrightarrow{d : T'}; \Delta_3, a_{m+1} : \overline{S_{m+1}}, \ldots, a_n : \overline{S_n}, \overrightarrow{c : \overline{S'}} \vdash^{\phi_3} \mathcal{E}\end{array}}{\Gamma', \overrightarrow{b : T}; \Delta_2, \Delta_3, \overrightarrow{a : \overline{S}}, \overrightarrow{c : S'^{\sharp}}, \overrightarrow{d : T'^{\sharp}} \vdash^{\phi_2 + \phi_3} \mathcal{D} \parallel \mathcal{E}}\end{array}}{\Gamma; \Delta_1, \Delta_2, \Delta_3, \overrightarrow{a : S^{\sharp}}, \overrightarrow{b : T^{\sharp}}, \overrightarrow{c : S'^{\sharp}}, \overrightarrow{d : T'^{\sharp}} \vdash^{\phi_1 + \phi_2 + \phi_3} \mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E})}$$

where

$$\begin{aligned}\overrightarrow{a : \overline{S}} &= a_1 : \overline{S_1}, \ldots, a_m : \overline{S_m}, \ldots, a_n : \overline{S_n} \\ \overrightarrow{b : \overline{T}} &= b_1 : T_1, \ldots, b_{m'} : T_{m'}, \ldots, b_{n'} : T_{n'}\end{aligned}$$

$$\Longleftrightarrow$$

$$\dfrac{\begin{array}{c}\Gamma = \Gamma'' + \Gamma_3 \\ \dfrac{\begin{array}{c}\Gamma'' = \Gamma_1 + \Gamma_2 \\ \Gamma_1, \overrightarrow{a : S}; \Delta_1, \overrightarrow{b : \overline{T}} \vdash^{\phi_1} \mathcal{C} \\ \Gamma_2, b_1 : T_1, \ldots, b_m : T_m, \overrightarrow{c : S'}; \Delta_2, \overrightarrow{d : \overline{T'}} \vdash^{\phi_2} \mathcal{D}\end{array}}{\begin{array}{c}\Gamma'', a_{m+1} : S_{m+1}, \ldots, a_n : S_n \\ \Delta_1, \Delta_2, a_1 : S_1^{\sharp}, \ldots, a_m : S_m^{\sharp}, \\ b_1 : T_1^{\sharp}, \ldots, b_{m'} : T_{m'}^{\sharp}; \\ b_{m'+1} : \overline{T_{m'+1}}, \ldots b_{n'} : \overline{T_{n'}}, \vdash^{\phi_1 + \phi_2} \mathcal{C} \parallel \mathcal{D}\end{array}} \qquad \begin{array}{c}\Gamma_3, b_{m'+1}, \ldots, b_n : T_n, \overrightarrow{d : T'}; \\ \Delta_3, a_{m+1} : \overline{S_{m+1}}, \ldots, a_n : \overline{S_n}, \overrightarrow{c : \overline{S'}} \vdash^{\phi_3} \mathcal{E}\end{array}\end{array}}{\Gamma; \Delta_1, \Delta_2, \Delta_3, \overrightarrow{a : S^{\sharp}}, \overrightarrow{b : T^{\sharp}}, \overrightarrow{c : S'^{\sharp}}, \overrightarrow{d : T'^{\sharp}} \vdash^{\phi_1 + \phi_2 + \phi_3} \mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E})}$$

□

The lemmas for subterm typeability and replacement are slightly different as we must consider unrestricted environments.

Lemma D.2 (Typeability of subterms (Access Points)). *If **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 **D** has a subderivation **D′** that concludes $\Gamma_2 \vdash M : B$, and the position of **D′** in **D** corresponds to the position of the hole in E.*

Proof. By induction on the structure of E. □

Lemma D.3 (Replacement (Access Points)). *If:*

- ***D** is a derivation of $\Gamma \vdash E[M] : A$, such that $\Gamma = \Gamma_1 + \Gamma_2$*
- ***D′** is a subderivation of **D** concluding $\Gamma_2 \vdash M : B$*

- *The position of $\mathbf{D}'$ in $\mathbf{D}$ corresponds to that of the hole in E*
- $\Gamma_3 \vdash N : B$
- $\Gamma' = \Gamma_1 + \Gamma_3$ *is defined*

then $\Gamma' \vdash E[N] : A$.

Proof. By induction on the structure of E. □

Theorem D.4 (Preservation, Access Points). *If* $\Gamma; \Delta \vdash^{\phi} C$ *and* $C \Longrightarrow \mathcal{D}$, *then* $\Gamma; \Delta \vdash^{\phi} \mathcal{D}$.

Proof. Recall that $\Longrightarrow$ is defined as $\equiv\longrightarrow\equiv$. Therefore, the result arises by induction on the derivation of $C \longrightarrow \mathcal{D}$ and as a corollary of Lemma D.1.

Again, since T-ConnectN subsumes T-Connect$_1$ and T-Connect$_2$, it suffices only to prove the new cases required for access point reduction.

Case E-Spawn

Assumption:

$$\frac{\Gamma \vdash E[\textbf{spawn}\ M] : C}{\Gamma; \cdot \vdash^{\bullet} \bullet E[\textbf{spawn}\ M]}$$

By Lemma D.2, we have that $\Gamma = \Gamma_1 + \Gamma_2$, and

$$\frac{\Gamma_2 \vdash M : \mathbf{1}}{\Gamma_2 \vdash \textbf{spawn}\ M : \mathbf{1}}$$

By Lemma D.3, we have that $\Gamma_1 \vdash E[()] : C$.
Recomposing:

$$\frac{\Gamma = \Gamma_1 + \Gamma_2 \qquad \Gamma_1; \cdot \vdash^{\bullet} E[()] \qquad \Gamma_2; \cdot \vdash^{\circ} \circ M}{\Gamma; \cdot \vdash^{\bullet} E[()] \parallel \circ M}$$

as required.

Case E-New

Assumption:

$$\frac{\Gamma \vdash E[\textbf{new}_S] : C}{\Gamma; \cdot \vdash^{\bullet} \bullet E[\textbf{new}_S]}$$

By Lemma D.2 and TA-New, we have that $\cdot \vdash \textbf{new}_S : \mathsf{AP}(S)$.
By Lemma D.3, we have that $\Gamma, z : \mathsf{AP}(S) \vdash E[z] : C$.
Thus, we can show:

$$\frac{\dfrac{\dfrac{\Gamma, z : \mathsf{AP}(S) \vdash E[z] : C}{\Gamma, z : \mathsf{AP}(S); \cdot \vdash^{\bullet} \bullet E[z]} \qquad \cdot; z : \mathsf{AP}(S) \vdash^{\circ} z(\epsilon, \epsilon)}{\Gamma, z : \mathsf{AP}(S); z : S \vdash^{\bullet} \bullet E[z] \parallel z(\epsilon, \epsilon)}}{\Gamma; \cdot \vdash^{\bullet} (\nu z)(\bullet E[z] \parallel z(\epsilon, \epsilon))}$$

as required.

Case E-Accept

Assumption:

$$\dfrac{\dfrac{\Gamma \vdash E[\mathbf{accept}\ z] : C}{\Gamma; \cdot \vdash^{\bullet} \bullet E[\mathbf{accept}\ z]} \qquad \cdot; z : S, \mathcal{X} : \overline{S}, \mathcal{Y} : S \vdash^{\circ} z(\mathcal{X}, \mathcal{Y})}{\Gamma; z : S, \mathcal{X} : \overline{S}, \mathcal{Y} : S \vdash^{\bullet} \bullet E[\mathbf{accept}\ z] \parallel z(\mathcal{X}, \mathcal{Y})}$$

By Lemma D.2, we have that $\Gamma = \Gamma_1 + \Gamma_2$ and that $\Gamma_2 \vdash \mathbf{accept}\ z : S$. Thus by TA-Accept we have that $z : \mathsf{AP}(S) \in \Gamma$.

By Lemma D.3, we have that $\Gamma, a : S \vdash E[a] : C$.

Recomposing, we have that:

$$\dfrac{\dfrac{\dfrac{\Gamma, a : S \vdash E[a] : C}{\Gamma, a : S; \cdot \vdash^{\bullet} \bullet E[a]} \qquad \cdot; z : S, \mathcal{X} : \overline{S}, a : \overline{S}, \mathcal{Y} : S \vdash^{\circ} z(\{a\} \cup \mathcal{X}, \mathcal{Y})}{\Gamma; z : S, \mathcal{X} : \overline{S}, \mathcal{Y} : S, a : S^{\sharp} \vdash^{\bullet} \bullet E[a] \parallel z(\{a\} \cup \mathcal{X}, \mathcal{Y})}}{\Gamma; z : S, \mathcal{X} : \overline{S}, \mathcal{Y} : S \vdash^{\bullet} (\nu a)(\bullet E[a] \parallel z(\{a\} \cup \mathcal{X}, \mathcal{Y}))}$$

Case E-Request

Assumption:

$$\dfrac{\dfrac{\Gamma \vdash E[\mathbf{request}\ z] : C}{\Gamma; \cdot \vdash^{\bullet} \bullet E[\mathbf{request}\ z]} \qquad \cdot; z : S, \mathcal{X} : \overline{S}, \mathcal{Y} : S \vdash^{\circ} z(\mathcal{X}, \mathcal{Y})}{\Gamma; z : S, \mathcal{X} : \overline{S}, \mathcal{Y} : S \vdash^{\bullet} \bullet E[\mathbf{accept}\ z] \parallel z(\mathcal{X}, \mathcal{Y})}$$

By Lemma D.2, we have that $\Gamma = \Gamma_1 + \Gamma_2$ and that $\Gamma_2 \vdash \mathbf{request}\ z : \overline{S}$. Thus by TA-Request we have that $z : \mathsf{AP}(S) \in \Gamma$.

By Lemma D.3, we have that $\Gamma, a : \overline{S} \vdash E[a] : C$. As duality is involutive, we have that $\overline{\overline{S}} = S$.

Recomposing, we have that:

$$\dfrac{\dfrac{\dfrac{\Gamma, a : \overline{S} \vdash E[a] : C}{\Gamma, a : \overline{S}; \cdot \vdash^{\bullet} \bullet E[a]} \qquad \cdot; z : S, \mathcal{X} : \overline{S}, \mathcal{Y} : S, a : S \vdash^{\circ} z(\mathcal{X}, \{a\} \cup \mathcal{Y})}{\Gamma; z : S, \mathcal{X} : \overline{S}, \mathcal{Y} : S, a : \overline{S}^{\sharp} \vdash^{\bullet} \bullet E[a] \parallel z(\mathcal{X}, \{a\} \cup \mathcal{Y})}}{\Gamma; z : S, \mathcal{X} : \overline{S}, \mathcal{Y} : S \vdash^{\bullet} (\nu a)(\bullet E[a] \parallel z(\mathcal{X}, \{a\} \cup \mathcal{Y}))}$$

as required.

Case E-Match

Assumption:

$$\dfrac{}{\cdot; z : S, a : \overline{S}, \mathcal{X} : \overline{S}, b : S, \mathcal{Y} : S \vdash^{\circ} z(\{a\} \cup \mathcal{X}, \{b\} \cup \mathcal{Y})}$$

Recomposing:

$$\dfrac{\cdot; z : S, \mathcal{X} : \overline{S}, \mathcal{Y} : S \vdash^{\circ} z(\mathcal{X}, \mathcal{Y}) \qquad \dfrac{\overline{S}/\epsilon = \overline{S/\epsilon} \qquad \cdot \vdash \epsilon : \epsilon \qquad \cdot \vdash \epsilon : \epsilon}{\cdot; a : \overline{S}, b : S \vdash^{\circ} a(\epsilon) \leftrightsquigarrow b(\epsilon)}}{\cdot; z : S, a : \overline{S}, \mathcal{X} : \overline{S}, b : S, \mathcal{Y} : S \vdash^{\circ} z(\mathcal{X}, \mathcal{Y}) \parallel a(\epsilon) \leftrightsquigarrow b(\epsilon)}$$

□

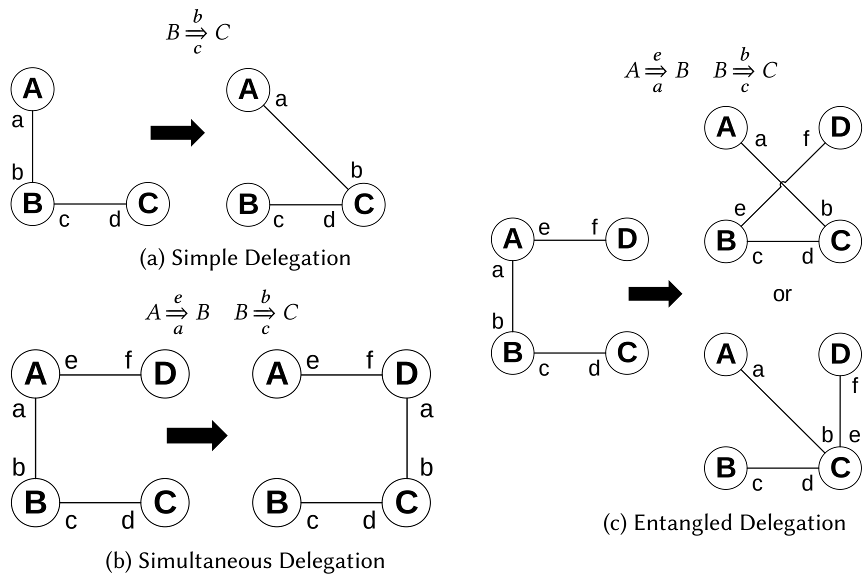


Fig. 13. Cases of Distributed Delegation

E 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.

E.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. 13 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*.

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

Fig. 14. Operation of Distributed Delegation Protocol

E.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.

E.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. 14 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.

E.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. 13b) can be handled in the same way as simple delegation. In the case of entangled delegation (Fig. 13c), 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.