Session Types without Tiers

Simon Fowler
University of Edinburgh
simon.fowler@ed.ac.uk

Sam Lindley
University of Edinburgh
sam.lindley@ed.ac.uk

J. Garrett Morris
University of Kansas
garrett@ittc.ku.edu

Sára Decova
University of Edinburgh
sara.decova@gmail.com

Abstract

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

We give the first formal account of session typing with failure handling in a functional programming language. We define a core calculus which satisfies preservation and progress properties, is deadlock free, data-race free, and terminating.

We provide the first implementation of session typing with failure handling for a fully fledged functional programming language, by extending the Links web programming language. Our implementation draws on existing work on algebraic effects and effect handlers. We illustrate our implementation through a chat server application for the web, in which all communication occurs over session typed channels and disconnections are handled gracefully.

1 Introduction

With the growth of the internet and mobile devices, as well as the failure of Moore's law, concurrency and distribution have become central to many applications. Writing correct concurrent and distributed code requires effective 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 [16, 17] are types for protocols. They describe both the shape and order of messages. If a program typechecks 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 [29] on affine session types, we present an account of session types that smoothly handles both distribution and failure. We present both a core calculus enjoying strong metatheoretical correctness properties and a practical implementation as an extension of the Links web programming language [8].

1.1 Session Types

We illustrate session types with a simple example of two-factor authentication; we present a larger example in §2. Two-factor authentication is often used when logging into banking applications. A user inputs their initial credentials, and if the login attempt is from a known device, then they are authenticated and may proceed to perform privileged actions. If the logon 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 correct response code is now sent to the server, then the user is authenticated.

$$
\begin{array}{l}
\mathbf{TwoFactorServer} \triangleq \\
\quad ?(\mathsf{Username}, \mathsf{Password}).\oplus\{ \\
\quad\quad \mathsf{Authenticated} : \mathit{Main}, \\
\quad\quad \mathsf{TwoFactorChallenge} : !\mathsf{Challenge}.?\mathsf{Response}. \\
\quad\quad\quad \oplus\{\mathsf{Authenticated} : \mathit{Main}, \mathsf{AccessDenied} : \mathsf{End}\}, \\
\quad\quad \mathsf{AccessDenied} : \mathsf{End}\}
\end{array}
$$

Here, we define the session type for a server which first receives (?) a pair of a username and password from a client. Next, the server chooses (⊕) whether to authenticate the client, perform a challenge, or reject the credentials. If the server decides to do a challenge, then it sends the challenge string (!), awaits the response, and finally either authenticates or rejects the client.

The client implements the *dual* session type:

$$
\begin{array}{l}
\mathbf{TwoFactorClient} \triangleq \\
\quad !(\mathsf{Username}, \mathsf{Password}).\&\{ \\
\quad\quad \mathsf{Authenticated} : \mathit{Main}, \\
\quad\quad \mathsf{TwoFactorChallenge} : ?\mathsf{Challenge}.!\mathsf{Response}. \\
\quad\quad \&\{\mathsf{Authenticated} : \mathit{Main}, \mathsf{AccessDenied} : \mathsf{End}\}, \\
\quad\quad \mathsf{AccessDenied} : \mathsf{End}\}
\end{array}
$$

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.

A session type specifies the behaviour of an endpoint of a channel. In our example there is one end point for the server and another one for the client.

1.2 Linearity

Let us suppose we have functions for sending and receiving along a session typed channel.

$$\textbf{send} : (A \times !A.S) \rightarrow () \\ \textbf{receive} : ?A.S \rightarrow A$$

Now, suppose we wish to implement the client. Assume a session channel s of type **`TwoFactorClient`**. We might attempt to write:

$$\textbf{send}((\texttt{"Alice"}, \texttt{"hunter2"}), s); \\ \textbf{send}((\texttt{"Bob"}, \texttt{"letmein"}), s)$$

Since we are re-using the endpoint s, we send a username and password pair twice along the same channel, thus losing all guarantees! In order to prevent this and to ensure all communication actions in a program are taken, we require a notion of *linearity*—that each endpoint is used exactly once. Accordingly, we must also thread the session type through the communication constructs.

$$\textbf{send} : (A \times !A.S) \rightarrow S \\ \textbf{receive} : ?A.S \rightarrow (A \times S)$$

1.3 Implementing Session Types

Linearity remains a major stumbling block for implementations of session types, however ingenious embeddings make use of advanced language features to ensure linearity, for example in Haskell [24, 30, 34, 37].

Another approach is to begin with a linear type system built into the language, and add session-typed constructs as primitives. Gay and Vasconcelos [12] describe a linear λ-calculus with asynchronous session types. Inspired by this, Wadler [41] defines a linear λ-calculus GV with strong connections to linear logic [23].

Lindley and Morris [26] integrate GV into the Links web programming language [8]. Links is a statically-typed functional programming language with polymorphism, Rémy-style row typing [38], and Hindley-Milner type inference. Links provides a *tierless* language which from a single source language compiles to JavaScript on the client, is interpreted on the server, and compiles to SQL for evaluating database queries. In this paper we extend the work of Lindley and Morris [26] to account for distribution and failure.

1.4 Distributed Session Types

Bringing session types to tierless web applications provides several challenges:

Client-to-client Communication It is natural to expect location transparency, meaning that one client may send a message along a channel where the other endpoint is held by another client. How do we implement this?

Distributed Delegation An important feature of session types is *delegation*, allowing channel endpoints to be sent over other session channels. Distributed delegation is difficult [20]—how can we perform distributed delegation when peers do not connect to one another directly?

Affine Sessions and Exception Handling A language providing distributed session types must provide failure handling—a user may simply close their browser window mid-session, after all! How can exceptions and sessions exist harmoniously, preserving correctness guarantees such as progress?

We address all of these challenges. In doing so, we provide the first formal treatment of session typing with failure handling in a functional programming language and the first implementation of session typing with failure handling for a fully fledged functional programming language.

1.5 Contributions

This paper makes four main contributions:

1. The design and implementation of an extension of the Links web programming language with support for session types with failure handling. As a primary use case, we implement multi-user, distributed web applications (§2).
2. A core lambda calculus, *Exceptional GV* (§3), a linear lambda calculus extended with asynchronous session typing with failure handling. We prove that the calculus enjoys progress.
3. An algorithm for distributed delegation (§4.4) in the more restrictive setting of web applications, where peers do not connect to each other directly.
4. Client and server backends for Links implementing session typing with failure handling (§4.5), drawing on connections with handlers for algebraic effects [36].

We have implemented several web applications with our implementation including a chat application, a distributed calculator, and an online multiplayer game. Full implementation and examples: (removed to preserve anonymity)

The rest of the paper is structured as follows: §2 illustrates our system with an example chat application; §3 presents the formalism; §4 describes the implementation; §5 outlines related work; and §6 concludes.

2 A Chat Application

We introduce distributed session types in Links by way of a session-typed chat application. We discuss the session types and the chat server implementation; the client implementation is dual, and is included in the supplemental material.

2.1 Chat Protocol

Informally, our chat protocol is as follows.

- Client initialisation:
  - connect to the chat server; then
  - send a nickname; then

```
typename Nickname = String;
typename Message = String;
typename Topic = String;

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|+];
```

Figure 1. Chat Client Session Types

  – 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.
- All participants should be notified whenever a participant joins or leaves the room.

Figure 1 gives the formal encoding of this protocol. (The syntax of Links uses `[+|...|+]` and `[&|...|&]` for $\oplus\{\dots\}$ and $\&\{\dots\}$.) Figure 1 gives session types for a chat client. A chat client endpoint (of type `ChatClient`) allows the client to proceed as follows. First, it sends a nickname to the server. Then it 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`). 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 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.

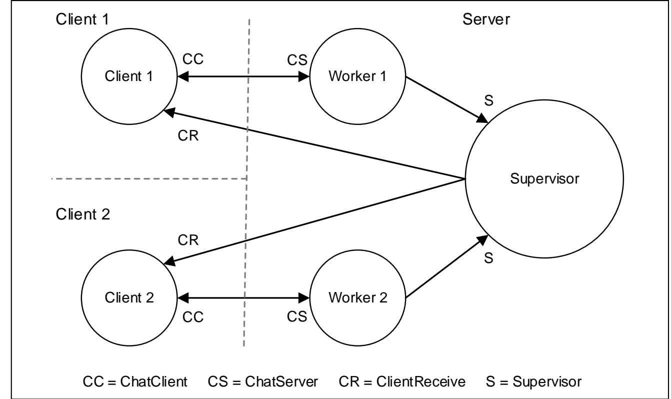

Figure 2. Chat Server Architecture

The chat server communicates with the client along endpoints with dual types. In Links we write `~S` for the dual of session type `S`. For example, the type of a chat server, written `~ChatClient`, is equivalent to:

```
?Nickname.
  [+|Join:!(Topic, [Nickname], ClientReceive).WorkerSend,
     Nope:End|+]
```

2.2 Chat Server Implementation

Figure 2 shows the architecture of the chat server application. Each client has a process which sends messages over a distributed session channel of type `ClientSend` to its own worker process on the server, which in turn sends internal messages to a supervisor process containing the state of the chat room, which then triggers the supervisor process to broadcast a message to all chat clients over a channel of type `~ClientReceive`.

The entry point of the application is the `main` function.

```
fun main() {
  spawn {supervisor((Topic("Hello, world!"), []),
                    LinNil)};
  spawn {acceptor()};
  addRoute("/", fun(_) {ChatClient.mainPage()});
  serveWebsockets();
  servePages()
}
```

(Links uses a JavaScript-like syntax. Functions are defined with `fun`. The `var` keyword is used for let-binding immutable variables. The arrow ~> denotes the function space of effectful functions.) We spawn a process which manages the state of the chat room. The `supervisor` function takes a server state consisting of a pair of the current topic and the current list of nicknames (initialised to `[]`), and a *linear* list of client broadcast channels (initialised to the empty linear list `LinNil`). We then spawn a process to accept new clients.

The last three lines of `main` set up the infrastructure for serving a web page. The `addRoute` function specifies how a request to a URL—in this case, the website's root URL—is

handled, specifying a function which generates a web page. The calls to `serveWebsockets` and `servePages` functions start the web server.

Access Points. Session typed channels are created using *access points* [12], which provide a matchmaking service for processes. Their interface is given by:

```
new : () ~> AP(S)
accept : (AP(S)) ~> S
request : (AP(S)) ~> ~S
```

We can create a *new* access point, *accept* on an access point to yield an endpoint of type `S` or *request* from the access point to yield a dual endpoint of type `~S`. Accepted and requested endpoints generated from a given access point are nondeterministically (but greedily) paired. Accepting and requesting are synchronous operations; that is, `accept` blocks until it matches a `request`, and vice-versa.

Supervisor. The supervisor (Figure 3) receives messages from the workers, using a top-level access point `sap`, and maintains the chat room state (topic, nicknames, and endpoints for communicating with clients). It accepts four types of messages: clients connecting to the chat room, sending messages, changing the topic, or disconnecting. (The `offer` construct offers a choice of messages. The pattern matching binds the continuation of the session.) The most involved case occurs when a client connects. The payload of the `Join` message is a `ChatServer` endpoint. If the supplied nickname is already in use then the supervisor process notifies the client (`select Nope c`) and terminates. Otherwise it forks a worker process. The `fork` library function, defined in terms of the access point API above, spawns a new process while connecting the parent process to the child process via a session-typed channel. (The function passed to `fork` must be linear, as it captures a linear endpoint.) Then, the map function for linear lists, `linMap`, is used to broadcast the new connection notification to all of the clients.

Workers (Figure 4) receive messages from the client and forward them to the supervisor (using the global `sap` access point). However, that is not the whole story. We also need to be able to handle the case where a client closes the browser window or is otherwise disconnected from the network. Handling disconnection requires us to make our first substantial addition to the theory of session types: integrating session types and exceptions. We extend the approach of Mostrous and Vasconcelos [29] from the setting of a synchronous, first-order process calculus to the setting of an asynchronous, higher-order functional language.

If communication succeeds then the worker remains active. Otherwise it sends a leave message to the supervisor. This is achieve using a `try` block around the communication actions which receive from a client. If the communication action completes successfully, `c` is bound in the `as` clause, and the function recurses as normal. If either the `offer` or a `receive`

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

sig sap : AP(Supervisor)
var sap = new();

sig supervisor
  : (Topic, [Nickname], LinList(WorkerSend)) ~> ()
fun supervisor(topic, nicks, cs) {
  var s = accept(sap);
  offer(s) {
    case Join(s) ->
      var (d, _) = receive(s);
      var (name, d) = receive(d);
      switch (filter(fun (nick) {nick == name}, nicks)) {
        case [] ->
          var workerSend = fork(
            linfun (clientReceive) {
              worker(name,
                     send((topic, nicks, clientReceive),
                          select Join d))
            });
          var cs = linMap(fun(c) {
              send (name, select Join c)
            }, cs);
          supervisor(topic, name :: nicks,
                     LinCons(workerSend, cs))
        case _ ->
          ignore(select Nope d);
          supervisor(topic, nicks, cs)
      }
    case Chat(s) ->
      var ((nick, msg), _) = receive(s);
      var cs = linMap(fun(c) {
          send ((nick, msg), select Chat c)
        }, cs);
      supervisor(topic, nicks, cs)
    case NewTopic(s) ->
      var (newTopic, _) = receive(s);
      var cs = linMap(fun(c) {
          send (newTopic, select NewTopic c)
        }, cs);
      supervisor(newTopic, nicks, cs)
    case Leave(s) ->
      var (nick, _) = receive(s);
      var (nicks, cs) = disconnect(nick, nicks, cs);
      var cs = linMap(fun(c) {
          send(nick, select Leave c)
        }, cs);
      supervisor(topic, nicks, cs)
  }
}
```

Figure 3. The `supervisor` function

```
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)
  }
}
```

Figure 4. The `worker` function

```
sig disconnect :
  (Nickname, [Nickname], LinList(WorkerSend)) ~>
  ([Nickname], LinList(WorkerSend))
fun disconnect(nick, ns, cs) {
  switch ((ns, cs)) {
    case ([], LinNil) -> ([], LinNil)
    case ((n :: ns), LinCons(c, cs)) ->
      if (n == nick) {
        cancel(c);
        (ns, cs)
      } else {
        var (ns, cs) = disconnect(nick, ns, cs);
        (n::ns, LinCons(c, cs))
      }
  }
}
```

Figure 5. The `disconnect` function

fails—that is, the peer endpoint disconnects before data is received—then an exception is thrown. Note that the channel is *not* bound in the `otherwise` clause, meaning that it is has been safely *cancelled*. We describe the channel cancellation and exception handling mechanisms in detail in Sections 3 and 4.5.

The `Chat` and `NewTopic` messages result in a straightforward update to the chat room state followed by a broadcast to all of the clients. In response to the `Leave` message, which is issued by a worker process when communication with the client fails (for instance, due to a browser window closing), the supervisor cancels the send channel for that client (Figure 5). The `cancel` function discards channels, introducing explicit *affinity* for session-typed channels.

3 Exceptional GV

In this section, we describe a core calculus for session types with failure handling, *Exceptional GV*, formalised as an extension to an asynchronous variant of the GV session-typed linear λ-calculus of Lindley and Morris [23].

3.1 Affine Sessions by Example

Safely integrating session types with failure handling in a functional language requires careful thought.

Fig. 6a illustrates the simple case where a channel is created using `fork`, and the child process cancels its endpoint. The parent attempts to receive on its endpoint, but no value can ever be sent! Thus, an exception is raised at line 5.

Fig. 6b illustrates the combination of exceptions, delegation, and asynchrony. First, a process is forked, with endpoint `s` passed to the parent process and `t` to the child. Next, the processes synchronise on access point `ap`; the parent is passed `carriedPeer` and the child `carried`. The child sends the `carried` endpoint along `t`, and then the processes synchronise on `sync`. The parent then cancels `s`, meaning that there is no way to retrieve `carried`. Finally, the parent attempts to receive from `carriedPeer`, but as `carried` is inaccessible an exception is raised at line 11.

Fig. 6c illustrates the case where a child process constructs a closure `clos` containing a channel name `a`. Before the closure can be sent, however, an exception is raised using `raise`. The parent process then attempts to receive from `b`, which is the peer endpoint of `a`. However, since the closure was never sent, it is impossible for the send operation to proceed, so an exception is raised on line 10.

3.2 Syntax and Typing Rules for Terms

Fig. 7 gives the syntax and typing rules of Exceptional GV types and terms. Types include unit **1**; linear functions $A \multimap B$; linear sums $A + B$; linear products $A \times B$; and session types S. Additionally, we include runtime types $S^\sharp$ and $S^\flat$ which are used for typing processes, but do not appear within terms (see the supplementary material for details).

The **fork** $(\lambda x.M)$ construct forks a new process, substituting a fresh channel endpoint of type S for x in M, and returns an endpoint with the dual type $\overline{S}$; **send** $M\ N$ sends M along endpoint N; and **receive** M receives along endpoint M.

We introduce three new term constructs to support session typing with failure handling: **cancel** M allows us to discard a session endpoint M, providing us with the ability to introduce explicit affinity; **raise** raises an exception; and **try** L **as** x **in** M **otherwise** N is used for exception handling, and is inspired by Benton & Kennedy's try – in – unless construct [1]. The ability to distinguish between a collection of possibly-failing operations and an explicit success continuation is convenient both for typing and for implementation, as we will see in §4.5.

```
1   try {
2     var s = fork (fun (t) {
3       cancel(t)
4     });
5     var (res, _) = receive(s);
6     res
7   } as (x) in {
8     print("Result: " ^^ x)
9   } otherwise {
10    print("Exception")
11  }
```

(a) Cancelled Endpoint

```
1   var ap = new(); var sync = new();
2   try {
3     var s = fork(fun(t) {
4       var carried = request(ap);
5       var _ = send(carried, t);
6       ignore(request(sync))
7     });
8     var carried = accept(ap);
9     ignore(accept(sync));
10    cancel(s);
11    var (res, _) = receive(carried); res
12  } as (x) in { print("Success: " ^^ x)
13  } otherwise { print("Exception") }
```

(b) Carried Endpoint

```
1   var ap = new();
2   try {
3     var s = fork (fun(t) {
4       var a = accept(ap);
5       var clos = linfun() { send(5, a) };
6       raise;
7       ignore(send(clos, t))
8     });
9     var b = request(ap);
10    ignore(receive(b));
11    cancel(s)
12  } as (x) in { print("Success")
13  } otherwise { print("Exception") }
```

(c) Closure

Figure 6. Examples of Channel Cancellation

Although Links supports branching and selection, we omit them in the core calculus (following [23, 26]) as they can be encoded using sums and delegation [9, 22].

Typing of Terms. We say that a context Γ is unrestricted, written $\mathsf{un}(\Gamma)$, when it only contains entries of the form a : End. As usual, linearity is enforced by the splitting of contexts when typing subterms, and the T-Var rule which allows a variable to be typed only in an unrestricted context.

The majority of the rules are standard for a linear λ-calculus. Session types are related by *duality*. The T-Fork rule makes use of duality to fork a process connected by dual endpoints of a channel. The rules T-Send and T-Recv capture session-typed communication.

As exceptions do not return values, the rule T-Raise allows an exception to be given any type A. The rule T-Try is similar to that of Mostrous and Vasconcelos [29]:

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

Their construct allows exception handling over a *single* communication action ρ, given that the subject of the communication action is some name a. If the communication action fails, then control moves to the exception handling process P which is typeable *without* a.

In order to allow exception handling over multiple communication actions, we take a different approach. Embracing the idea of having an explicit success continuation as advocated by Benton and Kennedy [1], instead of *subtracting* a linear name from a context upon failure, we *add* a result if the communication actions in L are successful.

The rule T-Cancel allows us to discard a session channel of type S, and is the main source of affinity in the calculus. Naïvely implemented, cancellation makes it impossible to prove progress: a process could discard an endpoint, leaving the peer waiting forever. However, as we will see in §3.4, our integration of channel cancellation and exceptions means that we retain strong progress guarantees.

3.3 Operational Semantics

We now give a small-step operational semantics that enjoys progress even in the presence of channel cancellation.

Runtime Syntax. Fig. 8 shows the runtime syntax of Exceptional GV. The semantics makes use of *configurations*, which are reminiscent of processes in the π-calculus: $(\nu a)\mathcal{C}$ binds name a in configuration $\mathcal{C}$; and parallel composition $\mathcal{C} \parallel \mathcal{D}$ is the parallel composition of configurations $\mathcal{C}$ and $\mathcal{D}$. Threads take the form ϕM, where ϕ is a flag describing whether the term is the *main thread* ($\bullet$), meaning that it may return a result, or a *child thread* ($\circ$), which may not. A configuration may have at most one main thread. Asynchrony is modelled by queue processes $a(Q) \leftrightsquigarrow b(R)$. Each buffer state Q and R may be a sequence of values ($\overrightarrow{V}$) or cancelled ($\lightning$). A cancelled endpoint ($\lightning\, a$) is no longer available to be used.

Evaluation Contexts. Handle contexts H allow an arbitrary nesting of exception handlers, ending with a pure context E. Pure contexts do not allow reduction under exception handlers. Configuration contexts $\mathcal{G}$ allow us to describe reduction under ν-binders and as part of a parallel composition. Pure flag contexts F and handle flag contexts D abstract over the values of thread flags.

Reduction Rules. Following prior work on linear functional languages with session types [12, 23, 25, 26], we describe the semantics of Exceptional GV as a deterministic reduction relation on terms, and a nondeterministic reduction relation on configurations. Fig. 9 presents the reduction rules for terms and configurations, which are defined up to the given configuration equivalences. Reduction on terms is largely standard, with the addition of a rule for substituting the result of a fully-evaluated try clause into the success continuation, and lifting rules for both pure and exception contexts.

Syntax of Types and Terms

Types	$A, B ::= \mathbf{1} \mid A \multimap B \mid A + B \mid A \times B \mid S \mid S^\sharp \mid S^\flat$
Session Types	$S ::= !A.S \mid ?A.S \mid \mathsf{End}$
Names	$\alpha ::= x \mid a$
Terms	$L, M, N ::= \alpha \mid \lambda x.M \mid M\,N$
	$\mid \mathbf{inl}\, M \mid \mathbf{inr}\, M$
	$\mid \mathbf{case}\, L\, \mathbf{of}\, \{\mathbf{inl}\, x \mapsto M; \mathbf{inr}\, x \mapsto N\}$
	$\mid () \mid (M, N) \mid \mathbf{let}\, (x, y) = M\, \mathbf{in}\, N$
	$\mid \mathbf{fork}\, M \mid \mathbf{send}\, M\, N \mid \mathbf{receive}\, M$
	$\mid \mathbf{cancel}\, M \mid \mathbf{raise}$
	$\mid \mathbf{try}\, L\, \mathbf{as}\, x\, \mathbf{in}\, M\, \mathbf{otherwise}\, N$

Typing Rules for Terms $\boxed{\Gamma \vdash M : A}$

T-Var
$$\frac{\mathsf{un}(\Gamma)}{\Gamma, \alpha : A \vdash \alpha : A}$$

T-Abs
$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \multimap B}$$

T-App
$$\frac{\Gamma_1 \vdash M : A \multimap B \quad \Gamma_2 \vdash N : A}{\Gamma_1, \Gamma_2 \vdash M\,N : B}$$

T-Unit
$$\frac{}{\cdot \vdash () : \mathbf{1}}$$

T-Inl
$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \mathbf{inl}\, M : A + B}$$

T-Inr
$$\frac{\Gamma \vdash M : B}{\Gamma \vdash \mathbf{inr}\, M : A + B}$$

T-Case
$$\frac{\Gamma_1 \vdash L : A + A' \quad \Gamma_2, x : A \vdash M : B \quad \Gamma_2, x : A' \vdash N : B}{\Gamma_1, \Gamma_2 \vdash \mathbf{case}\, L\, \mathbf{of}\, \{\mathbf{inl}\, x \mapsto M; \mathbf{inr}\, x \mapsto N\} : B}$$

T-Pair
$$\frac{\Gamma_1 \vdash M : A \quad \Gamma_2 \vdash N : B}{\Gamma_1, \Gamma_2 \vdash (M, N) : A \times B}$$

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

T-Fork
$$\frac{\Gamma \vdash M : S \multimap \mathbf{1}}{\Gamma \vdash \mathbf{fork}\, M : \overline{S}}$$

T-Send
$$\frac{\Gamma_1 \vdash M : A \quad \Gamma_2 \vdash N : !A.S}{\Gamma_1, \Gamma_2 \vdash \mathbf{send}\, M\, N : S}$$

T-Recv
$$\frac{\Gamma \vdash M : ?A.S}{\Gamma \vdash \mathbf{receive}\, M : (A \times S)}$$

T-Cancel
$$\frac{\Gamma \vdash M : S}{\Gamma \vdash \mathbf{cancel}\, M : \mathbf{1}}$$

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

T-Raise
$$\frac{}{\cdot \vdash \mathbf{raise} : A}$$

Duality on Session Types $\boxed{\overline{S}}$

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

Figure 7. Syntax and Typing Rules for Exceptional GV

Values	$U, V, W ::= x \mid a \mid \lambda x.M \mid ()$
	$\mid \mathbf{inl}\, V \mid \mathbf{inr}\, V \mid (V, W)$
Pure Contexts	$E ::= [\,] \mid E\,M \mid V\,E \mid \mathbf{inl}\, E \mid \mathbf{inr}\, E$
	$\mid \mathbf{case}\, E\, \mathbf{of}\, \{\mathbf{inl}\, x \mapsto M; \mathbf{inr}\, x \mapsto N\}$
	$\mid \mathbf{let}\, (x, y) = E\, \mathbf{in}\, M$
	$\mid \mathbf{send}\, E\, M \mid \mathbf{send}\, V\, E \mid \mathbf{receive}\, E$
Handle Contexts	$H ::= E \mid \mathbf{try}\, H\, \mathbf{as}\, x\, \mathbf{in}\, M\, \mathbf{otherwise}\, N$
Thread Flags	$\phi ::= \bullet \mid \circ$
Pure Flag Contexts	$D ::= \phi E$
Handle Flag Contexts	$F ::= \phi H$
Configurations	$\mathcal{C}, \mathcal{D}, \mathcal{E} ::= (\nu a)\mathcal{C} \mid \mathcal{C} \parallel \mathcal{D}$
	$\mid a(Q) \leftrightsquigarrow b(R) \mid \phi M \mid \mathbf{halt}$
Buffer States	$Q, R ::= \overrightarrow{V} \mid \lightning$
Config. Contexts	$\mathcal{G} ::= [\,] \mid (\nu a)\mathcal{G} \mid \mathcal{G} \parallel \mathcal{C}$

Figure 8. Runtime Syntax

Communication and Concurrency. Rule E-Fork creates two fresh names for each end 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. Rule E-Send-Ok and E-Recv-Ok describe sending to and receiving from a session channel respectively.

Channel Cancellation. Rule E-Cancel performs explicit cancellation of endpoint a: $F[\mathbf{cancel}\, a]$ reduces to $F[()] \parallel \lightning a$, removing the name from the context, and generating a cancellation process. Rule E-Chan-Cancel ensures that when an endpoint is cancelled, *all other endpoints contained in the channel* must also be cancelled. Thus, E-Chan-Cancel can be repeatedly applied until all affected buffers are cancelled.

Raising Exceptions. Following Mostrous and Vasconcelos [29], we opt to raise an exception when it would be otherwise impossible for a communication action to succeed. We can perform three communication actions: sending to, receiving from, and cancelling a channel endpoint. If a buffer contains values, it makes sense to be able to retrieve them. However, if a buffer is empty *and* the peer endpoint has been cancelled, then the receive operation would be blocked forever, so we raise an exception. More precisely, rule E-Recv-Bad ensures that if a thread is of the form $F[\mathbf{receive}\, a]$, the buffer associated with a is empty, and the peer endpoint b is cancelled, then an exception is raised, and endpoint a is cancelled.

Rule E-Send-Bad handles the case where a process tries to send a value where the peer endpoint has been cancelled. We make the decision not to raise an exception in this case since to do so would break confluence. Not raising exceptions on message sends is common in languages such as Erlang. We do, however, need to cancel all free channel variables contained in the sent message V.

Handling Exceptions. Rule E-Raise-H states that if **raise** occurs within a **try** block, then all channels contained within the pure context are cancelled, and the **otherwise** clause is evaluated. The structure of handling contexts ensures both that nesting of exceptions is possible, and that exceptions are handled by the *innermost* handler. Should an unhandled exception occur, then all names free in the top-level context are cancelled, and the process reduces to **halt**.

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

$$
\begin{array}{rll}
(\lambda x.M)\, V & \longrightarrow_{\mathsf{M}} M\{V/x\} & \\
\mathbf{let}\,(x,y) = (V,W)\ \mathbf{in}\ M & \longrightarrow_{\mathsf{M}} M\{V/x, W/y\} & \\
\mathbf{case\ inl}\ V\ \mathbf{of}\ \{\mathbf{inl}\ x \mapsto M; \mathbf{inr}\ x \mapsto N\} & \longrightarrow_{\mathsf{M}} M\{V/x\} & \\
\mathbf{case\ inr}\ V\ \mathbf{of}\ \{\mathbf{inl}\ x \mapsto M; \mathbf{inr}\ x \mapsto N\} & \longrightarrow_{\mathsf{M}} N\{V/x\} & \\
\mathbf{try}\ V\ \mathbf{as}\ x\ \mathbf{in}\ M\ \mathbf{otherwise}\ N & \longrightarrow_{\mathsf{M}} M\{V/x\} & \\
E[M] & \longrightarrow_{\mathsf{M}} E[M'] & (\text{if } M \longrightarrow_{\mathsf{M}} M') \\
H[M] & \longrightarrow_{\mathsf{M}} H[M'] & (\text{if } M \longrightarrow_{\mathsf{M}} M')
\end{array}
$$

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

$$
\begin{array}{l}
\mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E}) \equiv (\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E} \qquad \mathcal{C} \parallel (\nu a)\mathcal{D} \equiv (\nu a)(\mathcal{C} \parallel \mathcal{D}) \ \text{ if } a \notin \mathsf{fv}(\mathcal{D}) \qquad a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W}) \equiv b(\overrightarrow{W}) \leftrightsquigarrow a(\overrightarrow{V}) \\
\mathcal{C} \parallel \mathcal{D} \equiv \mathcal{D} \parallel \mathcal{C} \qquad \mathcal{G}[\mathcal{C}] \equiv \mathcal{G}[\mathcal{D}] \ \text{ if } \mathcal{C} \equiv \mathcal{D} \qquad (\nu a)(\nu b)(\lightning a \parallel \lightning b \parallel a(\lightning) \leftrightsquigarrow b(\lightning)) \parallel \mathcal{C} \equiv \mathcal{C} \\
\circ() \parallel \mathcal{C} \equiv \mathcal{C} \qquad (\nu a)\mathcal{C} \equiv \mathcal{C} \ \text{ if } a \notin \mathsf{fv}(\mathcal{C}) \qquad (\nu a)(\nu b)(\mathcal{C} \parallel \lightning b \parallel a(\epsilon) \leftrightsquigarrow b(\lightning)) \equiv \mathcal{C} \ \text{ if } a \notin \mathsf{fv}(\mathcal{C}) \\
\mathbf{halt} \parallel \mathcal{C} \equiv \mathcal{C} \qquad (\nu a)(\nu b)(\mathcal{C} \parallel \mathcal{D} \parallel a(\epsilon) \leftrightsquigarrow b(\epsilon)) \equiv \mathcal{C} \parallel \mathcal{D} \ \text{ if } a, b \notin \mathsf{fv}(\mathcal{C}) \cup \mathsf{fv}(\mathcal{D})
\end{array}
$$

Configuration Reduction $\boxed{\mathcal{C}_1 \longrightarrow_{\mathsf{C}} \mathcal{C}_2}$

$$
\begin{array}{lrll}
\text{E-Fork} & \mathcal{F}[\mathbf{fork}\,(\lambda x.M)] & \longrightarrow_{\mathsf{C}} (\nu a)(\nu b)(\mathcal{F}[a] \parallel \circ M\{b/x\} \parallel a(\epsilon) \leftrightsquigarrow b(\epsilon)) & \\
\text{E-Send} & \mathcal{F}[\mathbf{send}\ U\ a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W}) & \longrightarrow_{\mathsf{C}} \mathcal{F}[a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W} \cdot U) & \\
\text{E-Receive} & \mathcal{F}[\mathbf{receive}\ a] \parallel a(U \cdot \overrightarrow{V}) \leftrightsquigarrow b(Q) & \longrightarrow_{\mathsf{C}} \mathcal{F}[(U, a)] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(Q) & \\
\text{E-Cancel} & \mathcal{F}[\mathbf{cancel}\ a] & \longrightarrow_{\mathsf{C}} \mathcal{F}[()] \parallel \lightning a & \\
\text{E-Chan-Cancel} & \lightning a \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(Q) & \longrightarrow_{\mathsf{C}} \lightning a \parallel \lightning c_1 \parallel \ldots \parallel \lightning c_n \parallel a(\lightning) \leftrightsquigarrow b(Q) & \text{where } \mathsf{fcvs}(\overrightarrow{V}) = \{c_i\}_{i \in 1..n} \\
\text{E-Send-Bad} & \mathcal{F}[\mathbf{send}\ W\ a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\lightning) & \longrightarrow_{\mathsf{C}} \mathcal{F}[a] \parallel \lightning c_1 \parallel \cdots \parallel \lightning c_n \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\lightning) & \text{where } \mathsf{fcvs}(W) = \{c_i\}_{i \in 1..n} \\
\text{E-Recv-Bad} & \mathcal{F}[\mathbf{receive}\ a] \parallel a(\epsilon) \leftrightsquigarrow b(\lightning) & \longrightarrow_{\mathsf{C}} \mathcal{F}[\mathbf{raise}] \parallel \lightning a \parallel a(\epsilon) \leftrightsquigarrow b(\lightning) & \\
\text{E-Raise-H} & \mathcal{F}[\mathbf{try}\ E[\mathbf{raise}]\ \mathbf{as}\ x\ \mathbf{in}\ M\ \mathbf{otherwise}\ N] & \longrightarrow_{\mathsf{C}} \mathcal{F}[N] \parallel \lightning a_1 \parallel \cdots \parallel \lightning a_n & \text{where } \mathsf{fcvs}(E[\mathbf{raise}]) = \{a_i\}_{i \in 1..n} \\
\text{E-Raise-UnH} & \mathcal{D}[\mathbf{raise}] & \longrightarrow_{\mathsf{C}} \mathbf{halt} \parallel \lightning a_1 \parallel \cdots \parallel \lightning a_n & \text{where } \mathsf{fcvs}(D[\mathbf{raise}]) = \{a_i\}_{i \in 1..n} \\
\text{E-Lift} & \mathcal{G}[\mathcal{C}] & \longrightarrow_{\mathsf{C}} \mathcal{G}[\mathcal{D}] & \text{if } \mathcal{C} \longrightarrow_{\mathsf{C}} \mathcal{D} \\
\text{E-LiftM} & \phi M & \longrightarrow_{\mathsf{C}} \phi M' & \text{if } M \longrightarrow_{\mathsf{M}} M'
\end{array}
$$

Figure 9. Reduction Rules and Equivalences for Terms and Configurations

3.4 Metatheory

In the supplementary material, we describe a type system for Exceptional GV configurations which rules out deadlocked configurations, and ensures that buffers contain values corresponding to the session types of their endpoints. Subject reduction holds for Exceptional GV configurations; in turn, it follows that communication along a session channel follows its given session type. Exceptional GV retains global progress, even in the presence of channel cancellation. Moreover, if a well-typed closed configuration cannot reduce, and its main thread does not contain free channel names, then it is equivalent either to a value or to **halt**.

Due to lack of space, full details and proofs of the technical development may be found in the supplementary material.

3.5 Unrestricted Types and Access Points

The core of Exceptional GV includes neither unrestricted types (other than End) nor access points. These features are orthogonal and can be readily added following Lindley and Morris [26], as indeed they are in the implementation.

A notable advantage of avoiding access points where possible (programming using **fork** instead) is that the resulting correctness properties are quite strong. In particular, deadlock freedom and data race freedom are guaranteed. On the flipside, **fork** is too weak to support many interesting forms of concurrency and distribution, which necessarily depend on allowing some form of data race.

4 Implementation

In this section we describe our extensions to the Links concurrency runtimes to support distribution; distributed delegation; and how we map handling of communication failure onto handlers for algebraic effects.

4.1 The Links Model

Links provides a uniform language for web applications, with client code compiled to JavaScript; server code run on the server; and database queries compiled to SQL. Each client has its own concurrency runtime, providing lightweight processes which communicate using message passing.

Earlier versions of Links [8] (prior to version 0.7) invoke a fresh copy of the server per server request, and communication between client and server is via RPC calls which invoke a fresh copy of the server, relying on serialisation of continuations in order to maintain server state.

In order to better support multi-user applications such as chat, Links now adopts an application server model, in which the server persists. On top of this we have implemented fully-bidirectional communication between clients and the server, as well as the abstraction of direct client-to-client connection.

4.2 Concurrency

Links provides lightweight, typed, actor-style concurrency, where processes have a single incoming message queue and can send asynchronous messages. Lindley and Morris [26] extended Links with session-typed channels, making use of Links' process-based model but replacing actor mailboxes with session-typed channels. Our implementation extends theirs with support for 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 by the CPS compiler 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 [7] discusses the Links client concurrency model in depth. The server implements concurrency on top of the OCaml `lwt` library [40], 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 and include the location of the endpoint. If a channel (a, b) exists at a given location, then that location should contain a buffer for b.

4.3 Distributed Communication

In order to support bidirectional communication between client and server we use WebSockets [10]. A WebSocket connection is established by a client. When a request is made and a web page is generated, each client is assigned a unique identifier, which it uses to establish a WebSocket connection. Any messages the server attempts to send prior to a WebSocket connection being established are buffered and delivered once the connection is established. Once the WebSocket connection is established, we use a JSON protocol to communicate messages such as access point operations, remote session messages, and channel cancellation notifications.

Due to access points and delegation, it is possible that one client will have one end of a channel, and another client will have the other end of the channel. 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

`Endpoint ID` $\mapsto$ `Location`

where `Location` is either `Server` or `Client(ID)`, where `ID` identifies a particular client. The map is updated if: a connection is established using `fork` or an access point; an endpoint is sent as part of a message (§4.4); or a client disconnects.

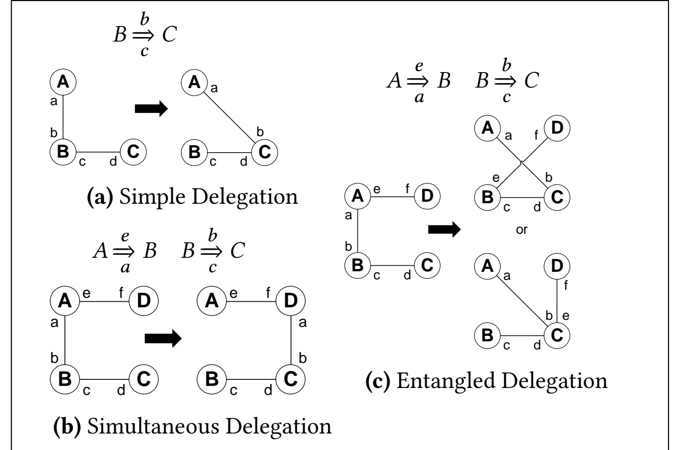

Figure 10. Cases of Distributed Delegation

The server also maintains a map

`Client ID` $\mapsto$ `[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, we cancel all channels associated with the client, and invoke exceptions as per the exception handling mechanism described further in §3 and §4.5.

4.4 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 §2, in the `ChatClient` type:

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

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

4.4.1 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 [20].

Fig. 10 shows three scenarios of distributed delegation, as described by Hu et al. [20]. 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. [20], 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. [20], we call such messages *lost messages*.

4.4.2 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. [20] 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.

4.4.3 Delegation in Distributed Session Links

Alas, we cannot directly re-use the resending and forwarding protocols of Hu et al. [20] 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. 11 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

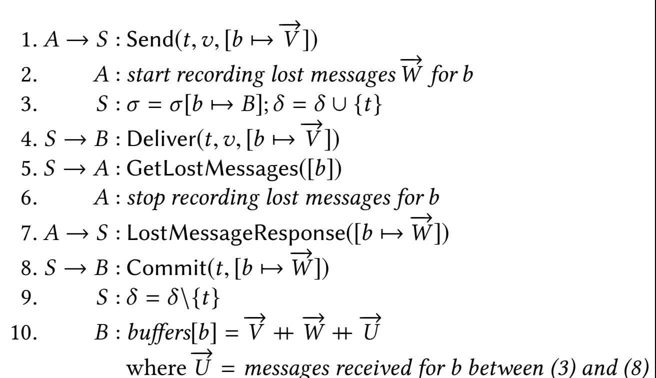

Figure 11. Operation of Distributed Delegation Protocol

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.

4.4.4 Correctness

We argue correctness of the algorithm in a similar manner to Hu et al. [20]. 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. 10b) can be handled in the same way as simple delegation. In the case of entangled delegation (Fig. 10c, 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.

4.5 Session Typing with Failure Handling

4.5.1 Handlers for Algebraic Effects

Algebraic effects [35] and their handlers [36] are a modular, compositional abstraction for programming with user-defined effects. Exception handlers are in fact a special case of

```
Cancel(Channel(peer, local)) ≜
  if local ∉ cancelled then
    let affected = {c | v ∈ buffers[local], c ∈ Channels(v)} in
    cancelled := cancelled ∪ {local}
    Delete(buffers[local])
    for (c ∈ affected) {Cancel(c)}
    Notify(peer)
```

Figure 12. Channel Cancellation

handlers for algebraic effects. Consequently, we leverage the existing implementation of effect handlers in Links [13, 14].

Adopting the syntax of Hillerström and Lindley's $\lambda^{\rho}_{\text{eff}}$ calculus [13], we translate exception handling as follows:

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

At a high-level, **handle** M **with** H installs a handler H for effects occurring in the effectful computation M. A handler H is a collection of *operation* clauses $\{\ell_i\ x_i\ k_i \mapsto M_i\}_i$, and a distinguished return clause $\{\textbf{return} x \mapsto M\}$. Each operation clause describes how to handle operation ℓ_i, binding the *parameter* x_i and *resumption* (a first-class manifestation of the delimited continuation of the computation up to the nearest enclosing handler) of the computation k_i in the clause body M_i. The result value of M is bound to x in the **return** clause. Should an effectful operation be invoked in M, which is handled in a clause in H, then control will pass to the associated handler clause. The *resumption* of a handler is reified as a function, allowing control to pass back to the program after handling the effect.

In our setting, we have a single operation, **fail** with no parameter. Should **fail** be invoked, then control passes to the **fail** clause in the nearest handler. At this point, assuming an extension of the **cancel** operator to arbitrary values, we cancel all free endpoint variables captured by the resumption (it being the manifestation of the current evaluation context).

As a preprocessing step, before translating to effect handlers, we insert a dummy exception handler around each spawned process.

$$\mathcal{T}(\!|\textbf{spawn } M|\!) = \textbf{spawn}(\textbf{try } \mathcal{T}(\!|M|\!) \textbf{ as } x \textbf{ in } () \textbf{ otherwise } ())$$

The translation $\mathcal{T}(\!|-|\!)$ ensures that unhandled exceptions are trapped and all channels in the context are cancelled should a communication action raise an exception.

4.5.2 Channel Cancellation

The ability to cancel all channels associated with an arbitrary value crucially depends on being able to inspect closures at runtime. Fortunately, Links performs closure conversion so this is relatively straightforward to achieve.

Fig. 12 shows the implementation of channel cancellation. We write `Channels` for the function that returns the set of channels contained in a value. We assume a global variable *cancelled*, bound to a set of cancelled channel endpoints. If an endpoint has already been cancelled, then the algorithm returns immediately; if not, then the *local* endpoint is added to the set of cancelled endpoints. Next, any channels appearing in the buffer associated with *local* are cancelled recursively. Finally, a cancel notification is sent to the peer.

4.5.3 Raising Exceptions

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

In order to support cancellation of closures on the client, we adorn function closures with an explicit environment field that can be directly inspected. Currently, Links does not closure convert continuations on the client, so we use a workaround in order to simulate cancelling a resumption (as required by the translation of exception handlers of §4.5.1) When compiling client code, for each occurrence of **do fail**, we compile a function which inspects all affected variables and cancels any affected channels in the continuation. For each occurrence of **receive**, we compile a continuation to cancel affected channels, which can be invoked by the runtime system should the receive operation fail.

4.5.4 Distributed Exceptions

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

5 Related Work

5.1 Session Types with Failure Handling

Structured Interactional Exceptions. Carbone et al. [3] provide the first theoretical basis for exceptions in a session-typed process calculus. Our approach allows significant simplifications: channel cancellation processes provide us with a simpler semantics and remove the need for queue levels, their meta-reduction relation, and their liveness protocol.

Affine Sessions. Our work draws on that of Mostrous and Vasconcelos [29], who introduced channel cancellation processes $\lightning a$. We make three further formal contributions to the theory: adapting it to a higher-order setting; working

with asynchronous channels; and allowing exception handling over multiple operations. Moreover, we have a full implementation extending Links.

Multiparty Session Types and Let-it-fail. Fowler [11] describes an Erlang implementation of the Multiparty Session Actor framework proposed by Neykova and Yoshida [32] with a limited form of failure recovery; Neykova and Yoshida [31] present a much more fully-fledged approach, based on refining existing Erlang supervision strategies. Chen et al. [5] introduce a formalism based on multiparty session types [18] that handles partial failures by transforming programs to detect possible failures at a set of statically determined synchronization points. These approaches rely on a fixed communication topology, in which failure of a participant can be associated immediately with the protocols in which it is involved, and mechanisms such as depedency graphs or synchronization points can be used to determine which participants are affected when one participant fails. In the setting of binary channels, delegation implies location transparency and thus failure detection must be at the level of a *channel* as opposed to a *participant*.

5.2 Session Types and Distribution

Hu et al. [20] introduce Session Java (SJ), which allows distributed session-based communication in the Java programming language, making use of the Polyglot framework [33] to statically check session types. Every SJ session channel is associated with a socket, whereas we allow lightweight session channels within a single concurrency runtime.

The authors 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; this feature is not formalised.

5.3 Session Types via Affine Types

Rust [28] provides *ownership types* [6], ensuring that an object has a unique owner. Jespersen et al. [21] use Rust's ownership types to encode session types which may be used at most once; they are affine rather than linear. Following Mostrous and Vasconcelos [29], we implement affine session types by way of an explicit **cancel** operator for discarding linear channels. This is in contrast to the Rust implementation where channels are discarded implicitly. The Rust implementation is not distributed, but if it were then it would seem challenging to handle failure as the implicit discarding of channels offers no hook to notify peers that a channel has been cancelled.

6 Conclusion and Future Work

We have presented a practical extension of the Links web programming language to support distributed session-based communication including distributed delegation and failure handling. Our approach hinges on the first formal treatment of session typing with failure handling in a functional programming languages. We have extended GV with constructs for cancelling a session endpoint and exception handling. Our calculus enjoys progress even in the presence of exceptions and channel cancellation. Our implementation supports distribution and failure handling, making use of recent work on handlers for algebraic effects.

We identify three immediate areas of future work.

N-Tier Applications. Our current implementations retains the original three-tier design of Links. It would be interesting to investigate architectures with more than one server.

Input-guarded Choice. As in previous work on session-typed functional languages, our **receive** primitive works on a *single* channel. Process calculi such as the π-calculus and CSP [15], as well as programming languages such as Concurrent ML [39] allow *nondeterministic choice*—performing a communication action on one of multiple channels.

Hu et al. [19] investigate input-guarded choice in the setting of a dynamically-typed core calculus with set types and typecase. It would be interesting to study how to adapt their approach to a statically-typed setting. We also wonder to what extent one might simulate input-guarded choice using access points.

Multiparty Session Types. Honda et al. [18] describe a theory of *multiparty* session types, which has spawned a multitude of implementations. However, multiparty session types have not yet been incorporated as *first-class* constructs in a core functional language. A natural starting point would be a lambda calculus in which we can translate Carbone et. al's MCP calculus [4], which is based on classical linear logic.

References

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

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

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

[4] Marco Carbone, Sam Lindley, Fabrizio Montesi, Carsten Schürmann, and Philip Wadler. 2016. Coherence generalises duality: A logical explanation of multiparty session types. In *CONCUR*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[5] Tzu-Chun Chen, Malte Viering, Andi Bejleri, Lukasz Ziarek, and Patrick Eugster. 2016. *A Type Theory for Robust Failure Handling in Distributed Systems*. Springer, 96–113.

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

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

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

[9] Ornela Dardha, Elena Giachino, and Davide Sangiorgi. 2017. Session types revisited. *Information and Computation* (2017).

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

[11] Simon Fowler. 2016. An Erlang Implementation of Multiparty Session Actors. In *ICE*. Electronic Proceedings in Theoretical Computer Science.

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

[13] Daniel Hillerström and Sam Lindley. 2016. Liberating effects with rows and handlers. In *TyDe*. ACM, 15–27.

[14] Daniel Hillerström, Sam Lindley, Robert Atkey, and KC Sivaramakrishnan. 2017. Continuation passing style for effect handlers. In *FSCD*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[15] C.A.R. Hoare. 1985. *Communicating Sequential Processes*. Prentice-Hall.

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

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

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

[19] Raymond Hu, Dimitrios Kouzapas, Olivier Pernet, Nobuko Yoshida, and Kohei Honda. 2010. Type-Safe Eventful Sessions in Java.. In *ECOOP*, Vol. 10. Springer, 329–353.

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

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

[22] Naoki Kobayashi. 2003. *Type Systems for Concurrent Programs*. Springer Berlin Heidelberg, Berlin, Heidelberg, 439–453. https://doi.org/10.1007/978-3-540-40007-3_26

[23] Sam Lindley and J Garrett Morris. 2015. A Semantics for Propositions as Sessions. In *ESOP*, Vol. 15. 560–584.

[24] Sam Lindley and J Garrett Morris. 2016. Embedding Session Types in Haskell. In *Haskell Symposium*. ACM, 133–145.

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

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

[27] Sam Lindley and J Garrett Morris. 2017. Lightweight Functional Session Types (extended version). (2017).

[28] Nicholas D. Matsakis and Felix S. Klock, II. 2014. The Rust Language. *Ada Lett.* 34, 3 (Oct. 2014), 103–104. https://doi.org/10.1145/2692956.2663188

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

[30] Matthias Neubauer and Peter Thiemann. 2004. An implementation of session types. In *PADL*, Vol. 4. Springer, 56–70.

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

[32] Rumyana Neykova and Nobuko Yoshida. 2017. Multiparty Session Actors. *Logical Methods in Computer Science* Volume 13, Issue 1 (March 2017). https://doi.org/10.23638/LMCS-13(1:17)2017

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

[34] Dominic Orchard and Nobuko Yoshida. 2016. Effects as sessions, sessions as effects. In *Haskell Symposium*, Vol. 51. ACM, 568–581.

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

[36] Gordon Plotkin and Matija Pretnar. 2009. Handlers of algebraic effects. In *ESOP*. Springer, 80–94.

[37] Riccardo Pucella and Jesse A Tov. 2008. Haskell session types with (almost) no class. In *Haskell Symposium*, Vol. 44. ACM, 25–36.

[38] 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.

[39] J.H. Reppy. 1999. *Concurrent Programming in ML*. Cambridge University Press.

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

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

A Omitted Client Code

It remains to describe the implementation of the client component of the chat server application. The entry point is the `mainPage` function, which is called to generate a web page whenever the server receives a web request.

```
fun mainPage() {
  page
    <html>
      <head><title>Links chat</title></head>
      <div id="main">
        <div>
          <p>Nickname:</p>
          <form l:onsubmit="{connect()}">
            <input id="name_box" type="text"></input>
          </form>
        </div>
      </div>
    </html>
}
```

This function returns a simple HTML stub. We omit styling details, since they are orthogonal to the technical development. The initial content of the page is a text into which a user may enter a nickname.

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

Upon submission (as specified by the `l:onsubmit` property), a process will be spawned which requests a channel from the access point located on the server, and sends the nickname which has been entered into the text box. The server then either accepts (`Join`) or rejects (`Nope`) the nickname. If the nickname is accepted, then the server sends the current topic, list of nicknames, and a channel used to receive messages from the server, before invoking `beginChat`.

```
sig beginChat :
  (Topic, [Nickname], ClientReceive, ClientSend) ~> ()
fun beginChat(topic, nicks, incoming, outgoing) {
  var ap = (new() : AP(?Message.End));

  spawn {outgoingLoop(ap, outgoing)};
  spawn {incomingLoop(incoming)};

  fun chat() {
    ignore(send(getInputContents(chatBoxId), request(ap)));
    clearInput(chatBoxId)
  }

  var box =
    <div id="chatBox">
      <div id="topic">
        <p>Topic: {stringToXml(topic)}</p>
      </div>
      <div id="msgs"></div>
      <div>
        <form l:onsubmit="{chat()}">
          <input id="chat_box" type="text"/</input>
        </form>
      </div>
    </div>;

  domReplaceChildren(box, getNodeById("main"))
}
```

The `beginChat` function spawns processes for managing incoming and outgoing messages and returns HTML for the chatbox. Chat messages are dispatched from the form by being sent to the outgoing loop through a locally defined access point.

```
sig incomingLoop : (ClientReceive) ~> ()
fun incomingLoop(s) {
  offer(s) {
    case Chat(s) ->
      var ((nick, chatmsg), s) = receive(s);
      chat(nick, chatmsg);
      incomingLoop(s)
    case Join(s) ->
      var (nick, s) = receive(s);
      join(nick);
      incomingLoop(s)
    case NewTopic(s) ->
      var (topic, s) = receive(s);
      newTopic(topic);
      incomingLoop(s)
    case Leave(s) ->
      var (nick, s) = receive(s);
      removeUser(nick);
      incomingLoop(s)
  }
}
```

The `incomingLoop` function receives messages from the server, showing changes to the user. For example, upon receiving

a `Chat` message, the function calls the `chat` function which adds a chat message to the screen.

```
sig outgoingLoop : (AP(?Message.End), ClientSend) ~> ()
fun outgoingLoop(ap, s) {
  fun isTopicCmd(s) {
    (charAt(s, 0) == '/') &&
    (strlen(s) > 8) &&
    (strsub(s, 0, 7) == "/topic ")
  }

  fun getTopic(s) {
    strsub(s, 7, strlen(s) - 7)
  }

  fun receiveMsg() {
    var msgChan = accept(ap);
    var (msg, _) = receive(msgChan);
    msg
  }

  var msg = receiveMsg();

  if (isTopicCmd(msg)) {
    var s = select NewTopic s;
    var s = send(getTopic(msg), s);
    outgoingLoop(ap, s)
  } else {
    var s = select Chat s;
    var s = send(msg, s);
    outgoingLoop(ap, s)
  }
}
```

Finally, `outgoingLoop` listens for user input, and relays this information to the server.

B Metatheory of Exceptional GV

B.1 Typing of Configurations

To rule out deadlocked configurations, and to ensure that buffers contain values corresponding to the session types of their endpoints, we provide typing rules for configurations (Figure 13). The typing judgement is of the shape $\Gamma; \Delta \vdash^{\phi} \mathcal{C}$, which can be read "under term environment Γ, runtime typing environment Δ, and flag ϕ, configuration $\mathcal{C}$ is well-typed". We introduce *flags* to show that there can be at most one main thread which can return a value (T-MainThread); other threads must return the unit value (T-Thread). Rule T-Nu introduces a channel name of type $S^\sharp$, which may then be used as a channel name and as part of a buffer (T-Par-Split). Conversely, T-Nu-Cancel introduces a channel name of type $S^\flat$, which may be used as a channel cancellation and as part of a buffer (T-Par-Split-Cancel).

Rule T-Queue ensures that buffers contain values corresponding to the session types of their endpoints. To guarantee this, we firstly introduce a judgement $\Gamma \vdash \overrightarrow{V} : \overrightarrow{A}$ which states that under environment Γ, the sequence of values $\overrightarrow{V}$ have types $\overrightarrow{A}$. Secondly, we introduce a type quotienting judgement S/S', which allows us to reason about session types, discounting values contained in the buffer. The session types of two buffer endpoints are evalstarible if they are dual, up to values contained in the buffer. Rules T-Queue-Cancel-L and T-Queue-Cancel-LR are specialisations of this requirement for when endpoints of a buffer have been cancelled.

Even in the presence of channel cancellation, we retain many of GV's strong metatheoretic properties.

The key property when considering session-typed systems is that of session fidelity: that all communication along session channels follows the prescribed session types. This follows as a corollary of the preservation of configuration typeability under reduction.

Session calculi with roots in linear logic exhibit deadlock-freedom since interpreting the logical cut rule as parallel composition necessarily ensures acyclicity of configurations. Coupled with appropriate reduction rules, it is also possible to use deadlock-freedom to show global progress. The main technical metatheoretic contribution of Exceptional GV is to show that global progress holds *even in the presence of channel cancellation*. We show this directly, without needing to introduce catalyser processes as in [2, 29].

B.2 Preservation

Preservation on the functional fragment of Exceptional GV is standard.

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

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

Lemma B.2 (Typeability of subterms (term contexts)). *If $\mathbf{D}$ 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 $\mathbf{D}$ has a subderivation $\mathbf{D}'$ that concludes $\Gamma_1 \vdash M : B$, and the position of $\mathbf{D}'$ in $\mathbf{D}$ corresponds to the position of the hole in E.*

Proof. By induction on the structure of E. ◻

Lemma B.3 (Replacement (term contexts)). *If:*

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

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

Proof. By induction on the structure of E. ◻

Lemma B.4 (Typeability of subterms (handle contexts)). *If $\mathbf{D}$ is a derivation of $\Gamma \vdash H[M] : A$, then there exist Γ_1, Γ_2 and B such that $\Gamma = \Gamma_1, \Gamma_2$, that $\mathbf{D}$ has a subderivation $\mathbf{D}'$ that concludes $\Gamma_2 \vdash M : B$, and the position of $\mathbf{D}'$ in $\mathbf{D}$ corresponds to the position of the hole in H.*

Type Quotienting $\boxed{S/S'}$

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

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

$$\frac{\mathsf{un}(\Gamma)}{\Gamma \vdash \epsilon : \epsilon} \qquad\qquad \frac{\Gamma_1 \vdash V : A \qquad \Gamma_2 \vdash \overrightarrow{V} : \overrightarrow{A}}{\Gamma_1, \Gamma_2 \vdash V \cdot \overrightarrow{V} : A \cdot \overrightarrow{A}}$$

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

$$\textsc{T-Par-Split}\ \frac{\Gamma_1, a : S; \Delta_1 \vdash^{\phi_1} \mathcal{C}_1 \qquad \Gamma_2; \Delta_2, a : \overline{S} \vdash^{\phi_2} \mathcal{C}_2}{\Gamma_1, \Gamma_2; \Delta_1, \Delta_2, a : S^{\sharp} \vdash^{\phi_1 + \phi_2} \mathcal{C}_1 \parallel \mathcal{C}_2}$$

$$\textsc{T-Par-Split-Cancel}\ \frac{\Gamma_1; \Delta_1, \lightning a : S \vdash^{\phi_1} \mathcal{C}_1 \qquad \Gamma_2; \Delta_2, a : \overline{S} \vdash^{\phi_2} \mathcal{C}_2}{\Gamma_1, \Gamma_2; \Delta_1, \Delta_2, a : S^{\flat} \vdash^{\phi_1 + \phi_2} \mathcal{C}_1 \parallel \mathcal{C}_2}$$

$$\textsc{T-Par}\ \frac{\Gamma_1; \Delta_1 \vdash^{\phi_1} \mathcal{C}_1 \qquad \Gamma_2; \Delta_2 \vdash^{\phi_2} \mathcal{C}_2}{\Gamma_1, \Gamma_2; \Delta_1, \Delta_2 \vdash^{\phi_1 + \phi_2} \mathcal{C}_1 \parallel \mathcal{C}_2}$$

$$\textsc{T-Nu}\ \frac{\Gamma; \Delta, a : S^{\sharp} \vdash^{\phi} \mathcal{C}}{\Gamma; \Delta \vdash^{\phi} (\nu a)\mathcal{C}} \qquad \textsc{T-Nu-Cancel}\ \frac{\Gamma; \Delta, a : S^{\flat} \vdash^{\phi} \mathcal{C}}{\Gamma; \Delta \vdash^{\phi} (\nu a)\mathcal{C}} \qquad \textsc{T-Thread}\ \frac{\Gamma \vdash M : \mathbf{1} \qquad \mathsf{un}(\Delta)}{\Gamma; \Delta \vdash^{\circ} \circ M}$$

$$\textsc{T-MainThread}\ \frac{\Gamma \vdash M : A \qquad \mathsf{un}(\Delta)}{\Gamma; \Delta \vdash^{\bullet} \bullet M} \qquad \textsc{T-Halt}\ \frac{\mathsf{un}(\Gamma) \qquad \mathsf{un}(\Delta)}{\Gamma; \Delta \vdash^{\phi} \mathbf{halt}}$$

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

$$\textsc{T-Queue-Cancel-L}\ \frac{\Gamma \vdash \overrightarrow{W} : \overrightarrow{A} \qquad S = \overline{S'/\overrightarrow{A}} \qquad \mathsf{un}(\Delta)}{\Gamma; \Delta, a : S, b : S' \vdash^{\circ} a(\lightning) \leftrightsquigarrow b(\overrightarrow{W})}$$

$$\textsc{T-Queue-Cancel-R}\ \frac{\Gamma \vdash \overrightarrow{V} : \overrightarrow{A} \qquad S/\overrightarrow{A} = \overline{S'} \qquad \mathsf{un}(\Delta)}{\Gamma; \Delta, a : S, b : S' \vdash^{\circ} a(\overrightarrow{V}) \leftrightsquigarrow b(\lightning)} \qquad \textsc{T-Queue-Cancel-LR}\ \frac{S = \overline{S'} \qquad \mathsf{un}(\Gamma) \qquad \mathsf{un}(\Delta)}{\Gamma; \Delta, a : S, b : S' \vdash^{\circ} a(\lightning) \leftrightsquigarrow b(\lightning)}$$

$$\textsc{T-Par-Split-Cancel-Sym}\ \frac{\Gamma_1; \Delta_1, a : S \vdash^{\phi_1} \mathcal{C} \qquad \Gamma_2; \Delta_2, \lightning a : \overline{S} \vdash^{\phi_2} \mathcal{C}_2}{\Gamma_1, \Gamma_2; \Delta_1, \Delta_2, a : S^{\flat} \vdash^{\phi_1 + \phi_2} \mathcal{C}_1 \parallel \mathcal{C}_2}$$

$$\textsc{T-Par-Split-Sym}\ \frac{\Gamma_1; \Delta_1, a : S \vdash^{\phi_1} \mathcal{C}_1 \qquad \Gamma_2, a : \overline{S}; \Delta_2 \vdash^{\phi_2} \mathcal{C}_2}{\Gamma_1, \Gamma_2; \Delta_1, \Delta_2, a : S^{\sharp} \vdash^{\phi_1 + \phi_2} \mathcal{C}_1 \parallel \mathcal{C}_2} \qquad \textsc{T-Cancellation}\ \frac{\mathsf{un}(\Gamma) \qquad \mathsf{un}(\Delta)}{\Gamma; \Delta, \lightning a : S \vdash^{\circ} \lightning a}$$

Combination of Flags $\boxed{\phi_1 + \phi_2 = \phi_3}$

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

Figure 13. Typing of Configurations

Proof. By induction on the structure of H, making use of Lemma B.2. □

Lemma B.5 (Replacement (handle contexts)). *If:*

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

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

Proof. By induction on the structure of H, making use of Lemma B.3. □

Lemma B.6 (Preservation (Terms)). *If* $\Gamma \vdash M : A$ *and* $M \longrightarrow_{\mathsf{M}} M'$, *then* $\Gamma \vdash M' : A$.

Proof. By induction on the derivation of $M \longrightarrow_{\mathsf{M}} M'$, using Lemmas B.1, B.3, and B.5. □

To prove preservation on configurations, we must first establish some auxiliary results. Lemma B.7 states how we may type subconfigurations in a configuration context.

Lemma B.7 (Typeability of subconfigurations). *If **D** is a derivation of* $\Gamma; \Delta \vdash^{\phi} \mathcal{G}[\mathcal{C}]$, *then there exist* Γ', Δ', ϕ' *such that **D** has a subderivation **D′** that concludes* $\Gamma'; \Delta' \vdash^{\phi'} \mathcal{C}$, *and the position of **D′** in **D** corresponds to the position of the hole in* G.

Proof. By induction on the structure of G. □

Lemma B.8 states that we may replace a subconfiguration of a configuration context, given that the environments are related by the cancellation relation.

Lemma B.8 (Replacement (configurations)). *If:*

- ***D** is a derivation of* $\Gamma; \Delta \vdash^{\phi} \mathcal{G}[\mathcal{C}]$
- ***D′** is a subderivation of **D** concluding that* $\Gamma'; \Delta' \vdash^{\phi'} \mathcal{C}$ *for some* Γ', Δ', ϕ'

Reduction on Session Types $\boxed{S \longrightarrow S'}$

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

Reduction on Typing Environments $\boxed{\Gamma; \Delta \longrightarrow \Gamma'; \Delta'}$

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

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

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

Figure 14. Reduction on Session Types and Typing Environments

- *$\Gamma''; \Delta'' \vdash^{\phi'} C'$ for some Γ'', Δ'' such that $\Gamma'; \Delta' \longrightarrow^* \Gamma''; \Delta''$*
- *The position of **D** in **D**′ corresponds to that of the hole in 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 G. □

To prove preservation of configuration reduction, we must first introduce a relation describing the evolution of typing environments in the presence of cancellation, which can be found in Figure 14. The first rule states that a channel of type $S^{\sharp}$ can be cancelled, becoming $S^{\flat}$, and thus meaning that it cannot be split such that it can be used as a reference. The second states that a channel reference can be cancelled, and must be used as a channel cancellation instead. The remaining rules are administrative, capturing the evolution of session types.

Finally, we may prove preservation under configuration reduction. In contrast to many statements of type preservation, we require that the typing context Γ contains only channel names; whereas we may cancel channel endpoints, it makes little sense to cancel arbitrary linear variables.

Theorem B.9 (Preservation (Configurations))**.**
Assume Γ only contains entries of the form $a_i : S_i$.

If $\Gamma; \Delta \vdash^{\phi} C$ and $C \longrightarrow_C C'$, then there exist Γ', Δ' such that $\Gamma; \Delta \longrightarrow^* \Gamma'; \Delta'$ and $\Gamma'; \Delta' \vdash^{\phi} C'$.

Proof. By induction on the derivation of $C \longrightarrow_C C'$, making use of Lemmas B.6, B.7, and B.8.

As the proof cases are long, we describe them in Appendix C. □

B.2.1 Deadlock-freedom

Due to its correspondence with linear logic, GV is naturally deadlock-free. As we do not introduce any constructs which may introduce cycles between processes, the same is true for Exceptional GV. The technical development for showing deadlock-freedom is therefore similar to that of GV and FST [23, 27].

We begin by classifying the notion of a *blocked process*: that is, a process which is waiting to perform a communication on some channel endpoint.

Definition B.10 (Blocked predicate)**.** We say that a term M is *blocked on a channel a* if M is about to send on, receive on, or cancel a. Formally:

$$\begin{aligned} \mathsf{blocked}(a, M) \triangleq (M = H[\mathbf{send}\ V\ a]) \lor (M = H[\mathbf{receive}\ a]) \\ \lor (M = H[\mathbf{cancel}\ a]) \end{aligned}$$

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

Definition B.11 (Depends predicate)**.** We say that *b depends on a in C* if b appears in some thread blocked by a, or if b depends on some channel c which depends on a. Formally:

- $\mathsf{depends}(a, b, a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W}))$
- $\mathsf{depends}(a, b, \phi M) \triangleq \mathsf{blocked}(a, M) \land b \in \mathsf{fcvs}(M)$
- $\mathsf{depends}(a, b, C) \triangleq C \equiv \mathcal{G}[C_1 \parallel C_2] \land \exists c.\mathsf{depends}(a, c, C_1) \land \mathsf{depends}(c, b, C_2)$

Given the notion of dependency, we can characterise deadlock as a configuration containing cyclic dependencies.

Definition B.12 (Deadlocked predicate)**.** We say that a configuration is *deadlocked* if it contains cyclic dependencies:

$$\begin{aligned} \mathsf{deadlocked}(C) \triangleq C \equiv \mathcal{G}[C_1 \parallel C_2] \\ \land \exists a, b.\mathsf{depends}(a, b, C_1) \land \mathsf{depends}(b, a, C_2) \end{aligned}$$

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

Lemma B.13. *If $\Gamma; \Delta \vdash^{\phi} C$ and $C = \mathcal{G}[C_1 \parallel C_2]$, then $\mathsf{fv}(C_1) \cap \mathsf{fv}(C_2)$ is either $\emptyset$ or $\{a\}$ for some channel name a.*

Proof. By induction on the derivation of $\Gamma; \Delta \vdash^{\phi} C$, due to the partitioning of the environment in the typing rules for parallel composition. Rules T-Par-Split and T-Par-Split-Cancel allow one name to be shared, whereas T-Par forbids sharing of names. □

With this, we can show that well-typed configurations are deadlock-free.

Theorem B.14. *If $\Gamma; \Delta \vdash C$, then $\neg\mathsf{deadlocked}(C)$.*

Proof. By contradiction. By the definition of deadlocked, we know that there must be some cyclic dependency, which would be ill-typed due to Lemma B.13. □

B.2.2 Progress

Deadlock-freedom alone is not sufficient to prove progress, especially in the presence of channel cancellation. Our goal is to show that Exceptional GV retains presence even in the face of channel cancellation. At a high-level, we define a *canonical form* to reason about a configuration as a whole, and proceed to characterise well-typed configurations in canonical form which do not reduce. Finally, we show that well-typed configurations which do not reduce and do not contain free channels in their main thread are structurally congruent to either a value or to **halt**. Firstly, we define the set of names in a typing environment.

Definition B.15 (Names in a Typing Environment)**.**

- If $\Gamma = \{x_i{:}A_i\}_i \cup \{a_j{:}B_j\}_j$, then $\mathsf{names}(\Gamma) = \{a_j\}_j$.
- If $\Delta = \{a_i{:}A_i\}_i \cup \{\lightning b_j{:}B_j\}_j$, then $\mathsf{names}(\Delta) = \{a_i\}_i \cup \{b_j\}_j$.

The functional fragment of Exceptional GV enjoys progress, under an environment containing only runtime names. The proof is a standard induction on typing derivations.

Lemma B.16 (Progress: Term Reduction)**.** *If $\Gamma \vdash M : A$, where Γ consists only of names a_i, then either:*

- *M is a value*
- *There exists some M' such that $M \longrightarrow_M M'$*
- *M can be written $H[M']$, where M' is a communication and concurrency primitive (i.e.* ***fork*** *M,* ***send*** *V W,* ***receive*** *V, or* ***cancel*** *V)*

To reason about progress of configurations, we define a *canonical form*, which makes explicit the property that at most one name is shared between processes. Let K range over *configuration leaves*—that is, child threads, buffers, or channel cancellations. Let P range over the subset of configuration leaves that are child threads, buffers, or channel cancellations. Note that P does *not* include **halt** or main threads.

Definition B.17 (Canonical Form)**.** A process C is in *canonical form* if there is a sequence of names $a_1, \ldots, a_n$, a sequence of configuration leaves $P_1, \ldots, P_n$, and some configuration $\mathcal{D}$, such that:

$$C = (\nu a_1)(P_1 \parallel (\nu a_2)(P_2 \parallel \cdots \parallel (\nu a_n)(P_n \parallel \mathcal{D}) \ldots))$$

where $\mathcal{D}$ is either $\bullet M$, or **halt**.

The canonical form ensures that *exactly* one channel is shared between each configuration. In the case that two configurations share no channels (that is, two processes composed using T-Par), we may obtain a canonical form by introducing a channel of unrestricted type end. Thus, well-typed configurations can be written in canonical form.

Proposition B.18 (Canonical Forms)**.** *Suppose $\Gamma; \Delta \vdash^\bullet C$. There exists some $C' \equiv C$ such that $\Gamma; \Delta \vdash^\bullet C'$, with*

$$C' = (\nu a_1)(P_1 \parallel (\nu a_2)(P_2 \parallel \cdots (\nu a_n)(P_n \parallel \mathcal{D}) \cdots))$$

where $\mathcal{D} = \bullet M$ for some M, or $\mathcal{D} =$ ***halt****.*

The proof is by induction on the count of ν-bound variables, with appeal to a counting argument. Armed with a canonical form, we can proceed to classify the nature of configurations which do not reduce. In particular, each configuration leaf must be either a buffer, a thread blocked on a preceding ν-bound name, or a thread blocked on a name in the environment.

Lemma B.19. *Let $\Gamma; \Delta \vdash^\bullet C$, with $C \not\longrightarrow_C$ and let Γ consist only of names a_i. Let $C' = (\nu a_1)(P_1 \parallel (\nu a_2)(P_2 \parallel \cdots \parallel (\nu a_n)(P_n \parallel \mathcal{D})) \ldots))$ be a canonical form of C. Then:*

1. *For $1 \leq i \leq n$, either:*
   a. *P_i is a buffer*
   b. *P_i is a channel cancellation $\lightning b$ for some $b \in \{a_j \mid 1 \leq j \leq i\} \cup \mathsf{names}(\Gamma) \cup \mathsf{names}(\Delta)$*
   c. *$P_i = \phi M_i$ for some M_i such that either M_i is a value V_i; or $\mathsf{blocked}(b, M_i)$ for some $b \in \{a_j \mid 1 \leq j \leq i\} \cup \mathsf{names}(\Gamma)$*
2. *Either:*
   a. *$\mathcal{D} = \bullet N$, where either N is a value, or $\mathsf{blocked}(b, N)$ for some $b \in \{a_i \mid 1 \leq i \leq n\} \cup \mathsf{names}(\Gamma)$*
   b. *$\mathcal{D} =$* ***halt***

The proof considers the form of each subconfiguration, ruling out **halt** configurations by definition. By Lemma B.16, we know that each thread must either be a value or written in the form $H[M]$, where M is a communication or concurrency action. This cannot be fork since it could reduce, thus the thread must be blocked on a channel, which by the typing rules must either be in the environment or a preceding ν-bound variable.

We can get a more precise statement by considering only closed-configurations. In particular, each P_i must either be a value, a buffer, or a blocked on a_i.

Proposition B.20. *Let $\cdot; \cdot \vdash^\bullet C$, with $C \not\longrightarrow_C$ and let $C' = (\nu a_1)(P_1 \parallel (\nu a_2)(P_2 \parallel \cdots \parallel (\nu a_n)(P_n \parallel \mathcal{D}) \ldots))$ be a canonical form of C. Then:*

1. *For $1 \leq i \leq n$, either:*
   a. *P_i is a buffer*
   b. *P_i is a channel cancellation $\lightning a_i$*
   c. *$P_i = \circ M_i$ for some M_i such that either M_i is a value V_i, or $\mathsf{blocked}(a_i, M_i)$*
2. *Either $\mathcal{D} = \bullet V$, or $\mathcal{D} =$* ***halt****.*

The proof relies on the observation that each name must appear at most once as a reference or channel cancellation, and at most once as part of a buffer. The final auxiliary result states that in a well-typed closed configuration where the main thread contains no free variables, that no thread may be blocked. This is established since each name must occur at most once as a reference or cancellation and at most once as a buffer endpoint; the typing of configurations; and acyclicity of configurations.

Lemma B.21. *Let $\cdot;\cdot \vdash^{\bullet} C$ with $C \not\longrightarrow_C$, and let*

$$C' = (\nu a_1)(P_1 \parallel (\nu a_2)(P_2 \parallel \cdots \parallel (\nu a_n)(P_n \parallel \mathcal{D})))$$

be a canonical form of C. Let $M_1, \ldots, M_m$ be the threads in C. If $\mathcal{D} = \bullet V$, where $\mathsf{fcvs}(V) = \emptyset$, or $\mathcal{D} = \textbf{halt}$, then it is not the case that $\mathsf{blocked}(a_i, M_j)$ for any $1 \leq j \leq m$.

If the main thread has reduced to a value which contains no free channels, we can show that the entire configuration is structurally congruent to a value. If the main thread has been cancelled, we can show that the entire configuration is structurally congruent to halt. This is established as a consequence of Proposition B.20 and Lemma B.21, along with the garbage collection congruences in Figure 9.

Theorem B.22 (Progress: Configurations)**.** *Let $\cdot;\cdot \vdash^{\bullet} C$, with $C \not\longrightarrow_C$, and let*

$$C' = (\nu a_1)(P_1 \parallel (\nu a_2)(P_2 \parallel \cdots \parallel (\nu a_n)(P_n \parallel \mathcal{D})\ldots))$$

be a canonical form of C. If $\mathcal{D} = \bullet V$ and $\mathsf{fcvs}(V) = \emptyset$, then $C \equiv \bullet V$. If $\mathcal{D} = \textbf{halt}$, then $C \equiv \textbf{halt}$.

C Proof Cases

Theorem B.9 (Preservation (Configurations))
Assume Γ only contains entries of the form $a_i : S_i$.

If $\Gamma;\Delta \vdash^{\phi} C$ and $C \longrightarrow_C C'$, then there exist Γ', Δ' such that $\Gamma;\Delta \longrightarrow^* \Gamma';\Delta'$ and $\Gamma';\Delta' \vdash^{\phi} C'$.

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

Without loss of generality, we consider unrestricted environments appearing in the premises of configuration typing rules $\mathsf{un}(\Gamma)$ and $\mathsf{un}(\Delta)$ to be empty.

Case E-Fork

$$F[\textbf{fork}\,\lambda x.M] \longrightarrow_C (\nu a)(\nu b)(F[a] \parallel M\{b/x\} \parallel a(\epsilon)\leftrightsquigarrow b(\epsilon))$$

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

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

Let $\mathbf{D}$ be the derivation of $\Gamma \vdash^{\bullet} H[\textbf{fork}\,\lambda x.M] : A$.
By Lemma B.4:

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

By inversion on T-Fork:

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

By Lemma B.5:
- $\Gamma_1, a : \overline{S} \vdash H[a] : A$

By inversion on T-Abs:
- $\Gamma_2, x : S \vdash M : \mathbf{1}$

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

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

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

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

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

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

By T-Nu:
- $\Gamma_1, \Gamma_2; \cdot \vdash^{\bullet} (\nu a)(\nu b)(H[a] \parallel \circ M\{b/x\} \parallel a(\epsilon)\leftrightsquigarrow b(\epsilon))$

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

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

as required.

Case E-Send

$F[\textbf{send}\ V'\ a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W}) \longrightarrow_{\mathrm{C}}$
$F[a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W} \cdot V')$

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

By inversion on T-ParSplit:

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

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

- $\exists H . F = \bullet(H[\textbf{send}\ V'\ a])$

By T-MainThread:

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

Let $\mathbf{D}$ be the derivation of $\Gamma_1, a : S \vdash H[\textbf{send}\ V'\ a] : C$.

By Lemma B.4:

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

By inversion on T-Send:

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

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

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

By the definition of duality:

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

By inversion on T-Queue:

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

By the definition of the quotienting function:

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

It follows that:

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

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

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

Thus by T-Queue:

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

By Lemma B.5:

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

By T-MainThread:

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

By the definition of F:

- $F[a] = \bullet H[a]$

So

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

By T-ParSplit:

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

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

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

as required.

Case E-Receive

$F[\textbf{receive}\ a] \parallel a(V' \cdot \overrightarrow{V}) \leftrightsquigarrow b(Q) \longrightarrow_{\mathrm{C}}$
$F[(V', a)] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(Q)$

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

By inversion on T-ParSplit:

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

By the definition of F:

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

By T-MainThread:

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

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

By Lemma B.4:

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

By inversion on T-Recv:

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

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

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

We have two subcases, based on whether or not Q is a buffer or has been cancelled.

Subcase $Q = \overrightarrow{W}$

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

By inversion on T-Queue:

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

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

  By the definition of $\Gamma_5 \vdash V' \cdot \overrightarrow{V} : A \cdot \overrightarrow{A}$:
- $\exists \Gamma_7, \Gamma_8 . \Gamma_5 = \Gamma_7, \Gamma_8$
- $\Gamma_7 \vdash V' : A$
- $\Gamma_8 \vdash \overrightarrow{V} : \overrightarrow{A}$

  By Lemma B.5:
- $\Gamma_3, \Gamma_7, a : S' \vdash H[(V', a)] : C$

  By T-MainThread:
- $\Gamma_3, \Gamma_7, a : S'; \cdot \vdash^{\bullet} H[(V', a)] : C$

  By the definition of the quotienting function:
- $\overline{S'}/\overrightarrow{A} = \overline{T/\overrightarrow{B}}$

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

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

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

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

  as required.

Subcase $Q = \lightning$

We have that $\Gamma_2; \Delta_2, a : !A.\overline{S'}, b : T \vdash^{\circ} a(V' \cdot \overrightarrow{V}) \leftrightsquigarrow b(\lightning)$.

By inversion on T-Queue-Cancel-Sym:
- $\Gamma_2 \vdash V' \cdot \overrightarrow{V} : A \cdot \overrightarrow{A}$
- $!A.\overline{S'}/A \cdot \overrightarrow{A} = \overline{T}$

  By the definition of $\Gamma_2 \vdash V' \cdot \overrightarrow{V} : A \cdot \overrightarrow{A}$:
- $\Gamma_2 = \Gamma_5, \Gamma_6$
- $\Gamma_5 \vdash V' : A$
- $\Gamma_6 \vdash \overrightarrow{V} : \overrightarrow{A}$

  By Lemma B.5:
- $\Gamma_3, \Gamma_5, a : S' \vdash H[(V', a)] : C$

  By T-MainThread:
- $\Gamma_3, \Gamma_5, a : S' \vdash^{\bullet} \bullet H[(V', a)] : C$

  By the definition of the quotienting function:
- $\overline{S'}/\overrightarrow{A} = \overline{T}$

  By T-Queue-Cancel-Sym:
- $\Gamma_6; a : \overline{S'}, b : T \vdash^{\circ} a(\overrightarrow{V}) \leftrightsquigarrow b(\lightning)$

  By T-Par-Split:
- $\Gamma_3, \Gamma_5, \Gamma_6; a : S'^{\sharp}, b : T \vdash^{\bullet} H[(V', a)] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\lightning)$

  By T-Nu:
- $\Gamma_3, \Gamma_5, \Gamma_6; b : T \vdash^{\bullet} (\nu a)(H[(V', a)] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\lightning))$

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

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

  as required.

Case E-Cancel

$F[\mathbf{cancel} a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(Q)) \longrightarrow_{\mathrm{C}} F[()] \parallel \lightning a \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(Q)$

Assumption: $\Gamma; \Delta \vdash^{\bullet} F[\mathbf{cancel}\ a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(Q)$

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

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

By T-MainThread:
- $\Delta_1 = \cdot$
- $\Gamma_1, a : S \vdash H[\mathbf{cancel}\ a] : C$

By Lemma B.4:
- $\exists \Gamma_3, \Gamma_4 . \Gamma_1, a : S = \Gamma_3, \Gamma_4, a : S$
- $\exists \mathbf{D'}$ such that $\mathbf{D'}$ is a subderivation of $\mathbf{D}$ concluding $\Gamma_4, a : S \vdash \mathbf{cancel}\ a : B$
- The position of $\mathbf{D'}$ in $\mathbf{D}$ corresponds to the position of the hole in H.

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

By Lemma B.5:
- $\Gamma_3 \vdash H[()] : C$

By T-MainThread:
- $\Gamma_3; \cdot \vdash^{\bullet} H[()]$

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

By T-Par-Split-Cancel:
- $\Gamma_2; \Delta_2, a : S^{\flat} \vdash^{\circ} \lightning a \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(Q)$

By T-Par:
- $\Gamma_2, \Gamma_3; \Delta_2, a : S^{\flat} \vdash^{\bullet} H[()] \parallel \lightning a \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(Q) \parallel \lightning a$

It is the case that $\Gamma_2, \Gamma_3; \Delta_2, a : S^{\sharp} \longrightarrow^{*} \Gamma_2, \Gamma_3; \Delta_2, a : S^{\flat}$. Since $\Gamma = \Gamma_2, \Gamma_3$ and $\Delta = \Delta_2, a : S^{\sharp}$, we have that
- $\Gamma; \Delta_2, a : S^{\flat} \vdash^{\bullet} H[()] \parallel \lightning a \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(Q)$

as required.

Case E-Chan-Cancel

$\lightning a \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(Q)) \longrightarrow_{\mathrm{C}} \lightning a \parallel \lightning c_1 \parallel \cdots \parallel \lightning c_n \parallel a(\lightning) \leftrightsquigarrow b(Q)$

Assumptions:
- $\Gamma; \Delta \vdash^{\circ} \lightning a \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(Q)$
- $\mathrm{fcvs}(\overrightarrow{V}) = \{c_i\}_i$

By inversion (T-Par-Split-Cancel):
- $\Delta = \Delta', a : S^{\flat}$
- $\cdot; \lightning a : S \vdash^{\circ} \lightning a$
- $\Gamma; \Delta, a : \overline{S} \vdash^{\circ} a(\overrightarrow{V}) \leftrightsquigarrow b(Q)$

We now have two subcases based on whether Q is a buffer or has been cancelled.

Subcase $Q = \overrightarrow{W}$

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

By inversion (T-Queue):

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

By definition of quotienting:

- $\overline{S} = !A_1 . \cdots .!A_n . \overline{S'}$

By duality:

- $S = ?A_1 . \cdots .?A_n . S'$

By assumptions (Γ only contains channel variables; $\text{fcvs}(\overrightarrow{V}) = \{c_i\}_i$):

- $\Gamma_1 = c_1 : S_{c_1}, \ldots, c_n : S_{c_n}$

By T-Queue-Cancel:

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

By T-Par (repeated applications):

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

By T-Cancellation:

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

By T-Par-Split-Cancel:

- $\Gamma_2; a : S'^{\flat}, b : T, \lightning c_1 : S_{c_1}, \ldots, \lightning c_n : S_{c_n} \vdash^{\circ} \lightning a \parallel \lightning c_1 \parallel \cdots \parallel \lightning c_n \parallel a(\lightning) \leftrightsquigarrow b(\overrightarrow{W})$

Recall $\Delta' = b : T$ and $\Gamma_1 = c_1 : S_{c_1}, \ldots, c_n : S_{c_n}$.

Recalling $S = ?A_1 . \cdots .?A_n . S'$, by repeated applications of $\longrightarrow$:

- $a : ?A_1 . \cdots .?A_n . S' \longrightarrow^{*} S'$

By definition of $\longrightarrow^{*}$:

- $\Gamma_2; c_1 : S_{c_1}, \ldots, c_n : S_{c_n}; \Delta', a : S^{\flat} \longrightarrow^{*} \Gamma_2; \Delta \vdash^{\circ} \lightning c_1 : S_{c_1}, \ldots, \lightning c_n : S_{c_n}, a : S'^{\flat}$

Thus

- $\Gamma_2; \Delta', \lightning c_1 : S_{c_1}, \ldots, \lightning c_n : S_{c_n}, a : S'^{\flat} \vdash^{\circ} \lightning a \parallel \lightning c_1 \parallel \cdots \parallel \lightning c_n \parallel a(\lightning) \leftrightsquigarrow b(\overrightarrow{W})$

as required.

Subcase $Q = \lightning$

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

By inversion (T-Queue-Cancel-Sym):

- $\Delta = b : T$
- $\Gamma \vdash \overrightarrow{V} : \overrightarrow{A}$
- $\overline{S}/\overrightarrow{A} = \overline{T}$

By assumptions (Γ only contains channel variables; $\text{fcvs}(\overrightarrow{V}) = \{c_i\}_i$):

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

By definition of quotienting:

- $\overline{S} = !A_1 . \cdots .!A_n . \overline{S'}$

By definition of duality:

- $S = ?A_1 . \cdots .?A_n . S'$

By T-CancelledQueue:

- $\cdot; a : S', b : T \vdash^{\circ} a(\lightning) \leftrightsquigarrow b(\lightning)$

By T-Par (repeated applications):

- $\Gamma_2; a : \overline{S'}, b : T, \lightning c_1 : S_{c_1}, \ldots, \lightning S_{c_n} \vdash^{\circ} \lightning c_1 \parallel \cdots \parallel \lightning c_n \parallel a(\lightning) \leftrightsquigarrow b(\lightning)$

By T-Cancellation:

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

By T-Par-Split-Cancel:

- $\Gamma_2; a : S'^{\flat}, b : T, \lightning c_1 : S_{c_1}, \ldots, \lightning c_n : S_{c_n} \vdash^{\circ} \lightning a \parallel \lightning c_1 \parallel \cdots \parallel \lightning c_n \parallel a(\lightning) \leftrightsquigarrow b(\lightning)$

Recall $\Delta' = b : T$ and $\Gamma_1 = c_1 : S_{c_1}, \ldots, c_n : S_{c_n}$.

Recalling $S = ?A_1 . \cdots .?A_n . S'$, by repeated applications of $\longrightarrow$:

- $a : ?A_1 . \cdots .?A_n . S' \longrightarrow^{*} S'$

By definition of $\longrightarrow^{*}$:

- $\Gamma_2; c_1 : S_{c_1}, \ldots, c_n : S_{c_n}; \Delta', a : S^{\flat} \longrightarrow^{*} \Gamma_2; \Delta \vdash^{\circ} \lightning c_1 : S_{c_1}, \ldots, \lightning c_n : S_{c_n}, a : S'^{\flat}$

Thus

- $\Gamma_2; \Delta', \lightning c_1 : S_{c_1}, \ldots, \lightning c_n : S_{c_n}, a : S'^{\flat} \vdash^{\circ} \lightning a \parallel \lightning c_1 \parallel \cdots \parallel \lightning c_n \parallel a(\lightning) \leftrightsquigarrow b(\lightning)$

as required.

Case E-Send-Bad

$F[\textbf{send}\ W\ a] \parallel \lightning b \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\lightning)) \longrightarrow_{\mathrm{C}}$
$F[a] \parallel \lightning c_1 \parallel \cdots \parallel \lightning c_n \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\lightning)$ where $\text{fcvs}(W) = c_1, \cdots, c_n$.

Assumption: $\Gamma; \Delta \vdash F[\textbf{send}\ W\ a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\lightning))$

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

By inversion on T-Par-Split:

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

By T-MainThread:

- $\Delta_1 = \cdot$
- $\Gamma_1, a : S \vdash H[\textbf{send}\ V\ a] : B$

By Lemma B.4:

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

So

- $S =!C.S'$

By inversion on T-Send:

- $\Gamma_4 \vdash W : C$

Since Γ contains only runtime names,

- $\Gamma_4 = c_1 : S_1, \cdots, c_n : S_n$.

By Lemma B.5:

- $\Gamma_3, a : S' \vdash H[a] : B$

By inversion on T-Queue-Cancel-R, knowing that duality is involutive:

- $\Gamma_2 \vdash \overrightarrow{V} : \overrightarrow{A}$
- $\Delta_2 = \cdot$
- $\overline{!C.S'}/\overrightarrow{A} = T$

Expanding the duality:

- $?C.\overline{S'}/\overrightarrow{A} = T$

By the definition of quotienting:

- $\overrightarrow{V}/\overrightarrow{A} = \epsilon$

So

- $T = !C.T'$

Thus, we can show:

- $\Gamma_2; \Delta_2, a : \overline{S'}, b : T' \vdash^{\circ} a(\overrightarrow{V}) \leftrightsquigarrow b(\lightning)$

By repeated applications of T-Par:

- $\Gamma_2; \Delta_2, a : \overline{S'}, b : T', \lightning c_1 : S_1, \ldots, \lightning c_n : S_n \vdash^{\circ} \lightning c_1 \parallel \cdots \parallel \lightning c_n \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\lightning)$

By T-Par-Split:

- $\Gamma_2, \Gamma_3; \Delta_2, a : S'^{\sharp}, b : T', \lightning c_1 : S_1, \ldots, \lightning c_n : S_n \vdash^{\bullet} F[a] \parallel \lightning c_1 \parallel \cdots \parallel \lightning c_n \parallel \lightning b \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\lightning)$

Since:

- $S = !A.S'$ and $!A.S' \longrightarrow S'$;
- $T = ?A.T'$ and $?A.T' \longrightarrow T'$;
- $\Gamma = \Gamma_2, \Gamma_3, a_1 : S_{c_1}, \cdots, a_n : S_{c_n}$
- $\Delta = \cdot, \Delta_2, a : S^{\sharp}$

we have that
$\Gamma; \Delta \longrightarrow^* \Gamma_2, \Gamma_3; \Delta_2, a : S'^{\sharp}, b : T', \lightning c_1 : S_1, \ldots, \lightning c_n : S_n$
with

- $\Gamma_2, \Gamma_3; \Delta_2, a : S'^{\sharp}, b : T', \lightning c_1 : S_1, \ldots, \lightning c_n : S_n \vdash^{\bullet} F[a] \parallel \lightning c_1 \parallel \cdots \parallel \lightning c_n \parallel \lightning b \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\lightning)$

as required.

Case E-Recv-Bad

$F[\textbf{receive}\ a] \parallel a(\epsilon) \leftrightsquigarrow b(\lightning) \longrightarrow_{\mathrm{C}}$
$F[\textbf{raise}] \parallel \lightning a \parallel a(\epsilon) \leftrightsquigarrow b(\lightning)$

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

By inversion on T-Par-Split:

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

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

By T-MainThread:

- $\Delta_1 = \cdot$
- $\Gamma_1 \vdash H[\textbf{receive}\ a] : B$

By Lemma B.4:

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

By inversion on T-Recv:

- $\Gamma_4 = \cdot$

By Lemma B.5:

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

By T-MainThread:

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

By T-Cancellation:

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

By T-Par:

- $\Gamma_3; \lightning a : S \vdash^{\bullet} F[\textbf{raise}] \parallel \lightning a$

By T-Par-Split-Cancel:

- $\Gamma_2, \Gamma_3; \Delta_2, a : S^{\flat} \vdash^{\bullet} F[\textbf{raise}] \parallel \lightning a \parallel a(\epsilon) \leftrightsquigarrow b(\lightning)$

with the required $\longrightarrow^*$ relation holding, as required.

Case E-Raise-UnH

$D[\textbf{raise}] \longrightarrow_{\mathrm{C}} \textbf{halt} \parallel \lightning a_1 \parallel \cdots \parallel \lightning a_n$ where $\mathrm{fcvs}(F[\textbf{raise}]) = \{a_i\}_{i \in 1..n}$ Assumption: $\Gamma; \Delta \vdash^{\bullet} H[\textbf{raise}]$

By inversion on T-MainThread, we have that $\exists E. D = \bullet E[\textbf{raise}]$

- $\Gamma \vdash E[\textbf{raise}] : A$
- $\mathrm{un}(\Gamma)$

By Lemma B.2:

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

By inversion on T-Raise:

- $\Gamma_1 = \cdot$

So

- $\Gamma = \Gamma_2$

Since Γ contains only runtime names, we have that

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

By T-Halt:

- $\cdot; \cdot \vdash^{\bullet} \textbf{halt}$

By T-Cancellation, we have:

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

for all a_i

By repeated applications of T-Par, we have:

- $\cdot; \lightning a_1 : S_1, \cdots, \lightning a_n : S_n \vdash^{\bullet} \textbf{halt} \parallel \lightning a_1 \parallel \cdots \parallel \lightning a_n$

Let $\Delta' = \lightning a_1 : S_1, \cdots, \lightning a_n : S_n$

Since $\Gamma; \cdot \longrightarrow^* \cdot; \Delta'$:

We have that $\cdot; \Delta' \vdash^{\bullet} \textbf{halt} \parallel \lightning a_1 \parallel \cdots \parallel \lightning a_n$ as required.

Case E-Raise-H

$F[\textbf{try}\ E[\textbf{raise}]\ \textbf{as}\ x\ \textbf{in}\ M\ \textbf{otherwise}\ N] \longrightarrow_{\mathrm{C}}$
$F[N] \parallel \lightning a_1 \parallel \cdots \parallel \lightning a_n$ where $\mathrm{fcvs}(E[\textbf{raise}]) = \{a_i\}_{i \in 1..n}$

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

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

By Lemma B.4:

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

By inversion on T-Try:

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

Since Γ contains only runtime names, we have that

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

By Lemma B.5:

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

By T-Cancellation, we have:

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

for all a_i

By repeated applications of T-Par, we have:

- $\Gamma_2; \lightning a_1 : S_1, \cdots, \lightning a_n : S_n \vdash^{\circ} F[N] \parallel \lightning a_1 \parallel \cdots \parallel \lightning a_n$

Let $\Delta' = \lightning a_1 : S_1, \cdots, \lightning a_n : S_n$

Since $\Gamma_2; \cdot \longrightarrow^* \cdot; \Delta'$:

We have that $\Gamma_2; \Delta' \vdash^{\circ} F[N] \parallel \lightning a_1 \parallel \cdots \parallel \lightning a_n$ as required.

Case E-Lift

$$\mathcal{G}[C] \longrightarrow_{\mathsf{C}} \mathcal{G}[C']$$

if $C \longrightarrow_{\mathsf{C}} C'$.

Assumption:

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

Call this derivation $\mathbf{D}$.

By Lemma B.7:

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

By the induction hypothesis:

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

By Lemma B.8:

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

and $\Gamma; \Delta \longrightarrow^* \Gamma'''; \Delta'''$, as required.

Case E-LiftM

$$\phi M \longrightarrow_{\mathsf{C}} \phi M'$$

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

Assumption:

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

We take the case where $\phi = \bullet$; the case with $\phi = \circ$ is similar.

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

By inversion on T-MainThread:

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

By Lemma B.6, we have that:

- $\Gamma \vdash M' : A$

By T-MainThread:

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

as required.

□