Polymorphic Imperative Session Types

ANONYMOUS AUTHOR(S)

The predominant formalizations of session-typed communication are either based on process calculi or on linear lambda calculus. An alternative model that models sessions using context-sensitive typing and typestate has been neglected due to its apparent restrictions. We call this model imperative because it treats communication channels like mutable variables and may look more familiar to imperative programmers.

We show that higher-order polymorphism and existential types are key to lift the restrictions imposed by previous work. On this bases, we define PolyVGR, the system of polymorphic imperative session types and establish its basic metatheory, type preservation and progress.

Additional Key Words and Phrases: Session types, polymorphism, existential types

1 INTRODUCTION

When Honda and others [Honda 1993; Takeuchi et al. 1994] proposed session types, little did they know that their system would become a cornerstone for type disciplines for communication protocols. Their original system describes bidirectional, typed message exchange between two processes in pi-calculus. It also contains facilities for offering and accepting choices in the protocol.

Subsequent work adds a plethora of features to the original system. One strand of ongoing work considers session-typed embeddings of communication primitives in functional and object-oriented languages, both theoretically and practically oriented [Gay and Vasconcelos 2010; Hu et al. 2010; Lindley and Morris 2017; Padovani 2017; Scalas and Yoshida 2016].

As an example of the style imposed by the functional embeddings [Gay and Vasconcelos 2010; Lindley and Morris 2017], consider this code fragment, which highlights the issues with this style:

```
let (x, c2) = receive c1 in       (* c1 : ?Int.?Int.!Int.s0 *)
let (y, c3) = receive c2 in       (* c2 : ?Int.!Int.s0 *)
let c4 = send (x + y, c3) in ... (* c3 : !Int.s0 *)
```

Listing 1. Channel-passing style

We enter this code with the typing `c1 : ?Int.?Int.!Int.s0` which means that `c1` is a channel ready to receive two integers, then send one, and continue the protocol according to session type `s0`. In these systems, channels are linear resources, so `c1` must be used exactly once: it is consumed in line 1 and cannot be used thereafter. The operation **receive** has type $?T.S \to (T \times S)$. When it consumes `c1`, it returns `c2 : ?Int.!Int.s0`, which is further transformed to `c3 : !Int.s0` by the next **receive**, and finally to `c4 : s0` by the **send** operation of type $(T \times !T.S) \to S$.

Writing a program in this style is cumbersome as programmers have to thread the channel explicitly through the program. Generally, this style is not safe for session types because most languages do not enforce the linearity needed to avoid aliasing of channel ends at compile time (see Scalas and Yoshida [2016] for an exception). Wrapping the channel handling in a parameterized monad [Atkey 2009] would accommodate the typing requirements and enforce linearity, but it is again cumbersome to scale the monadic style to programs handling more than one channel at the same time. Nevertheless, Pucella and Tov [2008] developed a Haskell representation in this style. In object-oriented languages, fluent interfaces enable the chaining of method calls for sending data [Hu and Yoshida 2016], but have similar issues as the functional style when scaling to multiple channels and new issues with receiving values which seems to require mutable references as shown in Listing 2.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

```
fun server u =
  let x = receive u in
  let y = receive u in
  send x + y on u
```

Listing 3. Example server

```
fun server' () =
  let x = receive u in
  let y = receive u in
  send x + y on u
```

Listing 4. Example server with capture

```
var x = new Ref<Int>();
var y = new Ref<Int>();
var c4 = c1.receive(x).receive(y).send (x.val + y.val);
```

Listing 2. Fluent interface with references

However, there is an alternative approach inspired by systems with typestate [Strom and Yemini 1986]. Vasconcelos et al. [2006] proposed a multithreaded functional language on this basis. This language, which we call VGR, enables a programming style which does not require linearity up-front and which easily scales to multiple channels. VGR lets us write the program fragment from Listing 1 as shown in Listing 3. The parameter `u` of the `server` function is a *channel reference*. The operation **`receive`** takes a channel reference associated with session type `!`**`Int`**`.S` and returns an integer. As a side effect, **`receive`** changes the type of the channel referred to by `u` to `S`, which clearly indicates that the VGR calculus is a typestate-based system [Strom and Yemini 1986]. Similarly, the function `send_on_` takes an integer to transmit and a channel reference associated with session type `!`**`Int`**`.S`. It returns a unit value and updates the channel's type to `S`.

Hence, the `server` function in Listing 3 expects a channel reference `u` referring to a channel of type `?`**`Int`**`.?`**`Int`**`.!`**`Int`**`.S` as an argument and leaves the channel in state/type `S` on exit. This change in typestate is reflected in the shape of a function type in VGR: $\Sigma_1; T_1 \;\rightarrow\; T_2; \Sigma_2$. Here, T_1 and T_2 are argument and return type of the function. The environments Σ_1 and Σ_2 map channel names to session types. They reflect the state (session type) of the channels before (Σ_1) and after (Σ_2) calling the function. Henceforth, we call these environments states. The type of a channel reference, **Chan** α, refers to the entry for α in the current state. Channels in T_1 refer to entries in Σ_1 and channels in T_2 refer to entries in Σ_2. The incoming state Σ_1 may refer to further channels that are consumed by the function; the outgoing state Σ_2 may contain further channels created by the function. The type of `server` in Listing 3 is given by

$$\{\alpha : ?\,\mathsf{Int}.?\,\mathsf{Int}.!\,\mathsf{Int}.S\}; \textbf{Chan}\ \alpha \rightarrow \textbf{Unit}; \{\alpha : S\}, \tag{1}$$

for some channel name α and session type S.

Listing 3 demonstrates that VGR does **not** impose linear handling of channel references, as there are multiple uses of variable `u`. Instead, it keeps track of the current state of every channel using a state Σ, which is threaded linearly through the typing rules. This style of typing comes with particular restrictions, which we outline now and discuss in detail in § 2.

To this end, let us examine the type of `server` in eq. (1) more closely. First, the type is monomorphic so that the continuation of the session, S, has to match any call site precisely. This restriction is shared with many functional systems [Fowler et al. 2019; Gay and Vasconcelos 2010], but there are also systems that employ polymorphism pervasively [Almeida et al. 2021; Lindley and Morris 2017].

Second, monomorphism extends to the symbolic *channel name* α used in the type. It is not a variable, but it is an arbitrary fixed name, like the label of a record or variant type, that identifies the channel so that the function can only be invoked on the channel called α.

Third, a function can be typechecked without knowledge of the channel names that are in use and their current state. This feature enables typing of a function that creates a new channel, but at the same time it makes it difficult to define the `server` function in a library, because the type-checker does not allow us to call the function on a channel named differently than α, even if its session type matches. Moreover, it is not possible to call the function `server` on two *different* channels. It is also not possible to invoke a function that creates a new channel twice in a row, because the two channel names would conflict. Hence, there is no meaningful abstraction over channel creation in VGR.

Sometimes the fixation on a particular channel name is required as we can see by examining Listing 4. The function `server'` is closed over a channel reference `u` of type **Chan** α. It can only be used in a context that provides the same channel α, which is reflected in the type of `server'`:

$$\{\alpha : ?\,\mathsf{Int}.?\,\mathsf{Int}.!\,\mathsf{Int}.S\}; \mathbf{Unit} \rightarrow \mathbf{Unit}; \{\alpha : S\}. \tag{2}$$

Clearly, it would not be safe to use this function with any other channel, because it is not possible to replace a channel reference captured in the closure for `server'`. We can still invoke a function of type (2) any time the channel α is in a state matching the "before" session type of the function. In contrast, we would like to invoke a function like `server` on any channel reference that matches the required session type, but VGR does not allow us to do so.

In the most closely related work, Saffrich and Thiemann [2021] study the connection between VGR and the synchronous variant of the calculus GV [Gay and Vasconcelos 2010]. They show that there is a typing and semantics preserving translation from VGR to GV, but that the back translation in the other direction, from GV to VGR, is only semantics preserving. They introduce a severely restricted version of GV's type system that essentially introduces channel identities and show that the back translation from restrictive GV to VGR is typing and semantics preserving. All calculi involved in their investigation are monomorphic.

Contributions

- We define PolyVGR, a novel polymorphic imperative session type system that lifts all restrictions imposed by earlier related systems, but still operates on the basis of the same semantics. Our type system exhibits a novel use of higher-kinded polymorphism to enable quantification over types that contain an a-priori unknown number of channel references.
- We establish type preservation and progress for PolyVGR.

2 MOTIVATION

We demonstrate how polymorphism in the form of universal and existential quantification lifts various restrictions of the VGR calculus. In particular, VGR is monomorphic with respect to channel names and states and it requires different operations (with different types) to transmit data and (single) channels. All these restrictions disappear in PolyVGR. Moreover, universal polymorphism gives us fine grained control over ownership passing of channels whereas existentials lead to a clean encoding of the concept of freshness required for channel creation and ownership passing.

The next few subsections draw the line between VGR and PolyVGR. Types and code fragments for the new calculus PolyVGR appear in grey boxes. The final subsection demonstrates the newly gained expressiveness of PolyVGR with several examples. We start with channel creation as it enables us to systematically explain the components of function types in PolyVGR.

2.1 Channel Creation

Channel creation in VGR works in two steps. First, we create an access point of type $[S]$, where S is a session type. This access point needs to be known to all threads that wish to communicate and it can be shared freely. Second, the client thread requests a connection on the access point and the server must accept it on the same access point. This rendezvous creates a communication channel with one end of type S on the server and the other end of type $\overline{S}$ (the dual type of S) on the client.

$$\textsc{C-Accept}\ \frac{\Gamma; v \mapsto [S] \qquad \text{fresh}\, c}{\Gamma; \Sigma; \mathsf{accept}\, v \mapsto \Sigma; \mathbf{Chan}\ c; \{c : S\}} \qquad \textsc{C-Request}\ \frac{\Gamma; v \mapsto [S] \qquad \text{fresh}\, c}{\Gamma; \Sigma; \mathsf{request}\, v \mapsto \Sigma; \mathbf{Chan}\ c; \{c : \overline{S}\}}$$

The VGR typing rules for these operations demonstrate that new channels just show up with a fresh name in the outgoing state of the expression typing. Similarly, if a function of type $\Sigma_1; T_1 \to T_2; \Sigma_2$ creates a new channel, then its name and session type just appear in Σ_2. Incoming channels described in Σ_1 are either passed through to Σ_2 or they are closed in the function. All channels mentioned in Σ_2, but not in Σ_1 are considered new.

As the channel names in states must all be different, the number of simultaneously open channels in a VGR program is bounded by the number of occurrences of the C-Accept and C-Request rules. VGR has recursive functions, but they are monomorphic with respect to incoming and outgoing states. In consequence, abstraction over channel creation is not possible.

In contrast, PolyVGR's function type indicates channel creation explicitly by using existential quantification. As an example, consider abstracting over the accept operation:

$$\begin{aligned} acc &= \Lambda(\sigma\colon \mathsf{Session}).\lambda(\,\cdot\,; x\colon [\sigma]).\mathsf{accept}\, x \\ &: \forall(\sigma\colon \mathsf{Session}).(\,\cdot\,; [\sigma] \to \exists\gamma\colon \mathsf{Dom}(\mathbb{X}).\gamma \mapsto \sigma; \mathsf{Chan}\,\gamma) \end{aligned} \tag{3}$$

The core of this type still has a shape like the VGR type $\Sigma_1; T_1 \to T_2; \Sigma_2$, but with some additions and changes. The most prominent change is that the outgoing type and state are swapped in a function type resulting in a structure like this: $(\Sigma_1;\ T_1 \to \exists\alpha\colon\ \mathsf{Dom}(n).\Sigma_2;\ T_2)$. The actual type in (3) starts with some straightforward universal quantification. Channel creation should work with access points of any session type $\sigma : \mathsf{Session}$. The incoming state Σ_1 only specifies the part of the state that is needed by the function; it can be applied in any state Σ that provides the required channels or more. In this case, the empty state $\cdot$ is required, so the function can be called in any state. On return, the function provides one new entry, which is disjointly added to the calling state.

On the left of the arrow, we have the required incoming state $\cdot$ and argument type $[\sigma]$ (access point for σ). On the right of the arrow, the existential quantification $\exists\gamma : \mathsf{Dom}(\mathbb{X})$ is a new ingredient that manages the abstraction over the created channel. The kind $\mathsf{Dom}(\mathbb{X})$ indicates abstraction over exactly one channel name. The number of channels is specified with the *shape constant* $\mathbb{X}$, which indicates one channel.[1] Hence, the variable γ can be used like a channel name in constructing an extended state. The returned value is a channel reference for γ. Thanks to the existential, which is the logical pendant of the fresh c constraint, we can invoke *acc* multiple times and obtain different channels from every invocation.

2.2 Channel Abstraction

The discussion of VGR's function type $\Sigma_1; T_1 \to T_2; \Sigma_2$ in the introduction shows that a function that takes a channel as a parameter can only be applied to a single channel. A function like `server` (Listing 3) must be applied to the channel of type **Chan** α, for some fixed name α.

To lift this restriction, we apply the standard recipe of universal quantification, i.e., polymorphism over channel names. Thus, the type of `server` generalizes as shown in (4) so that it can be applied to any channel of type **Chan** α regardless of the name α. The existential quantification runs empty (and hence omitted) in this type because the function does not return new channels.

[1]We defer further discussion of other shapes n and the meaning of $\mathsf{Dom}(n)$ to Sections 2.3.3 and 2.3.4.

$$\forall(\alpha\colon \mathsf{Dom}(\mathbb{X})).\forall(\sigma\colon \mathsf{Session}).\\ (\alpha \mapsto ?\,\mathsf{Int}.?\,\mathsf{Int}.!\,\mathsf{Int}.\sigma;\ \mathsf{Chan}\ \alpha \to \alpha \mapsto \sigma;\ \mathsf{Chan}\ \alpha) \tag{4}$$

We write the type of `server'` analogously to (2) by omitting the universal quantification over α:

$$\forall(\sigma\colon \mathsf{Session}).\\ (\alpha \mapsto ?\,\mathsf{Int}.?\,\mathsf{Int}.!\,\mathsf{Int}.\sigma;\ \mathsf{Unit} \to \alpha \mapsto \sigma;\ \mathsf{Unit}). \tag{5}$$

In this type, the channel name α is fixed as a value of this type is necessarily closed over a particular channel that cannot change between applications of the same function.

2.3 Data Transmission vs Channel Transmission

In VGR, it is possible to pass channels from one thread to another. The session type $!S'.S$ classifies a channel on which we can send a channel of type S'. The operation to send a channel has the following typing rule in VGR:

$$\textsc{C-SendS}\quad \frac{\Gamma; v \mapsto \mathbf{Chan}\ \beta \qquad \Gamma; v' \mapsto \mathbf{Chan}\ \alpha}{\Gamma; \Sigma, \alpha : !S'.S, \beta : S'; \mathsf{send}\ v\ \mathsf{on}\ v' \mapsto \Sigma; \mathbf{Unit}; \alpha : S}$$

The premises are *value typings* that indicate that v and v' are references to fixed channels β and α under variable environment Γ. The conclusion is an *expression typing* of the form $\Gamma; \Sigma; e \mapsto \Sigma_1; T; \Sigma_2$ where Σ is the incoming state, Σ_1 is the part of Σ that is passed through without change, and Σ_2 is the outgoing state after the operation indicated by expression e which returns a result of type T. The rule states that channels β and α have session type S' and $!S'.S$, respectively. The channel β is consumed (because it is sent to the other end of channel α) and α gets updated to session type S.

Compared to the function type, sending a channel is more flexible. Any channel of type S' can be passed because β is not part of channel α's session type. If the sender still holds references to channel β, then these references can no longer be exercised as β has been removed from Σ. So one can say that rule C-SendS passes ownership of channel β to the receiver.

However, there is another way to send a channel reference over a channel, namely if it is captured in a closure. To see what happens in this case, we look at VGR's rules for sending and receiving data of type D.

$$\textsc{C-SendD}\quad \frac{\Gamma; v \mapsto D \qquad \Gamma; v' \mapsto \mathbf{Chan}\ \alpha}{\Gamma; \Sigma, \alpha : !D.S; \mathsf{send}\ v\ \mathsf{on}\ v' \mapsto \Sigma; \mathbf{Unit}; \alpha : S} \qquad \textsc{C-ReceiveD}\quad \frac{\Gamma; v \mapsto \mathbf{Chan}\ \alpha}{\Gamma; \Sigma, \alpha : ?D.S; \mathsf{receive}\ v \mapsto \Sigma; D; \alpha : S}$$

One possibility for type D is a function type like $D_1 = \{\beta : S'\}; \mathbf{Unit} \to \mathbf{Unit}; \{\beta : S''\}$. A function of this type captures a channel named β which may or may not occur in Σ. It is instructive to see what happens at the receiving end in rule C-ReceiveD. If we receive a function of type D_1 and Σ already contains channel β of appropriate session type, then we will be able to invoke the function.

If channel β is not yet present at the receiver, we may want to send it along later. However, we find that this is not possible as the received channel gets assigned a fresh name d:

$$\textsc{C-ReceiveS}\quad \frac{\Gamma; v \mapsto \mathbf{Chan}\ \alpha \qquad \text{fresh}\ d}{\Gamma; \Sigma, \alpha : ?S'.S; \mathsf{receive}\ v \mapsto \Sigma; \mathbf{Chan}\ d; d : S', \alpha : S}$$

For the same reason, it is impossible to send channel β first and then the closure that refers to it: β gets renamed to some fresh d while the closure still refers to β. Sending the channel effectively cuts all previous connections.

To address this issue, PolyVGR adds quantified states to session types and lifts all restrictions on the type of values that may be transmitted. The revised grammar of session types admits (channel) existential packages as type for the payload:

$$S ::= {!}(\exists\alpha\colon \mathsf{Dom}(N).\Sigma;\ T).S \mid {?}(\exists\alpha\colon \mathsf{Dom}(N).\Sigma;\ T).S \mid \dots$$

A channel package can be instantiated by a state Σ and a payload type T. All channels referenced in T must be bound in Σ so that the sending and the receiving end of the channel agree about the channels sent along with the value of type T. That is, sending any value that contains channel references also transfers the underlying referenced channels to the receiver. Thus, sending a reference transfers ownership of the underlying channel. Moreover, a value may contain several channel references that need not all be different: A single underlying channel can be referenced many times, but at least once.

The "size" of Σ is gauged with α which determines its domain as indicated in its kind $\mathsf{Dom}(N)$ where N is the shape of the domain. Shapes range over $\mathbb{I}$ (the empty shape), $\mathbb{X}$ (the shape with one binding), and $N_1 \fatsemi N_2$ which forms the disjoint combination of shapes N_1 and N_2.

2.3.1 *No channels.*

To gain some intuition with this type construction, we start with a type for sending a primitive value of type Int, say. Beginning from the inside, we need to specify the session type. The general pattern is ${!}(\exists\alpha\colon \mathsf{Dom}(N).\Sigma;\ T).S$. Making it concrete, we find that

- $T = \mathsf{Int}$;
- $\Sigma = \cdot$, the empty state, as an Int value contains no channels;
- the type variable α specifies the domain of $\cdot$, which is also empty, indicated with $N = \mathbb{I}$.

Here is the resulting term and type, where we quantify over a continuation session type σ and a channel name c (its kind $\mathsf{Dom}(\mathbb{X})$ indicates that it is a single channel):

$$\begin{aligned} \mathit{send0} = {} & \Lambda(c\colon \mathsf{Dom}(\mathbb{X})).\Lambda(\sigma\colon \mathsf{Session}). \\ & \lambda(\cdot\,;\ x\colon \mathsf{Int}).\lambda(c \mapsto {!}(\exists\alpha\colon \mathsf{Dom}(\mathbb{I}).\cdot\,;\ \mathsf{Int}).\sigma;\ y\colon \mathsf{Chan}\, c). \\ & \mathsf{send}\ x\ \mathsf{on}\ y \\ :\ & \forall(c\colon \mathsf{Dom}(\mathbb{X})).\forall(\sigma\colon \mathsf{Session}). \\ & (\cdot\,;\ \mathsf{Int} \rightarrow\ \cdot\,;\ (c \mapsto {!}(\exists\alpha\colon \mathsf{Dom}(\mathbb{I}).\cdot\,;\ \mathsf{Int}).\sigma;\ \mathsf{Chan}\, c \rightarrow c \mapsto \sigma;\ \mathsf{Unit})) \end{aligned} \tag{6}$$

2.3.2 *One channel.*

We instantiate the general pattern ${!}(\exists\alpha\colon \mathsf{Dom}(N).\Sigma;\ T).S$ as follows to send a channel of type S'.

- Σ is now a state with a single binding, so that α must range over $\mathsf{Dom}(\mathbb{X})$;
- consequently, Σ has the form $\alpha \mapsto S'$; and
- $T = \mathsf{Chan}\,\alpha$;

We omit the term, which is similar to the one in (6), and just spell out the type. We quantify over the continuation session type and the names of the two channels involved. There is one novelty here: we have to declare that the two channels α and c are different, so that they can be used as keys in the state. The *disjointness constraint* $(\alpha \# c)$ does this job. It specifies that all channel names in α are disjoint from all channel names in c.

$$\begin{aligned} \mathit{send1}\colon & \forall(\alpha\colon \mathsf{Dom}(\mathbb{X})).\forall(c\colon \mathsf{Dom}(\mathbb{X})).\ (\alpha \# c) \Rightarrow \forall(\sigma\colon \mathsf{Session}). \\ & (\cdot\,;\ \mathsf{Chan}\,\alpha \rightarrow\ \cdot\,;\ (\alpha \mapsto S', c \mapsto {!}(\exists\alpha\colon \mathsf{Dom}(\mathbb{X}).\alpha \mapsto S';\ \mathsf{Chan}\,\alpha).\sigma;\ \mathsf{Chan}\, c \rightarrow c \mapsto \sigma;\ \mathsf{Unit})) \end{aligned}$$

2.3.3 *Two channels.*

Sending two channels of type S' and S'' requires new ingredients and illustrates the general case. The instantiation of the pattern $!(\exists\alpha\colon \mathsf{Dom}(N).\Sigma;\, T).S$ is as follows:

- the state Σ must have two bindings, one for each payload channel, so that α must range over a two element domain, e.g., $\mathsf{Dom}(\mathbb{X} \mathbin{⨟} \mathbb{X})$;
- to write down Σ, we need notation to address the $\mathbb{X}$-shaped components of α as in $\pi_1\,\alpha$ and $\pi_2\,\alpha$, so that we have $\Sigma = \pi_1\,\alpha \mapsto S', \pi_2\,\alpha \mapsto S''$;
- for simplicity, let's say we just send a pair of channels: $T = \mathsf{Chan}\,(\pi_1\,\alpha) \times \mathsf{Chan}\,(\pi_2\,\alpha)$.

$$\begin{array}{l}
\textbf{let } S(\sigma) = !(\exists\alpha\colon \mathsf{Dom}(\mathbb{X} \mathbin{⨟} \mathbb{X}).\pi_1\,\alpha \mapsto S', \pi_2\,\alpha \mapsto S'';\, \mathsf{Chan}\,(\pi_1\,\alpha) \times \mathsf{Chan}\,(\pi_2\,\alpha)).\sigma \textbf{ in}\\
\forall(\alpha\colon \mathsf{Dom}(\mathbb{X})).\forall(\beta\colon \mathsf{Dom}(\mathbb{X})).\ (\alpha \# \beta) \Rightarrow \forall(c\colon \mathsf{Dom}(\mathbb{X})).\ (\alpha \# c, \beta \# c) \Rightarrow \forall(\sigma\colon \mathsf{Session}).\\
(\,\cdot\,;\, \mathsf{Chan}\,\alpha \times \mathsf{Chan}\,\beta \to \,\cdot\,;\, (\alpha \mapsto S', \beta \mapsto S'', c \mapsto S(\sigma);\, \mathsf{Chan}\,c \to c \mapsto \sigma;\, \mathsf{Unit}))
\end{array}$$

We use the let-notation informally to improve readability. It is not part of the type system. Close study of the type reveals a discrepancy between the “curried” way to pass the arguments $\alpha\colon\ \mathsf{Dom}(\mathbb{X})$ and $\beta\colon\ \mathsf{Dom}(\mathbb{X})$ and the “uncurried” kind $\mathsf{Dom}(\mathbb{X} \mathbin{⨟} \mathbb{X})$ expected by the existential. To rectify this discrepancy, the term pairs the two domains to obtain some $\gamma = (\alpha, \beta)$ with $\gamma\colon\ \mathsf{Dom}(\mathbb{X} \mathbin{⨟} \mathbb{X})$ as needed for the existential. This definition of γ implies that $\alpha = \pi_1\,\gamma$ and $\beta = \pi_2\,\gamma$ which are needed obtain the correct state and type for the body of the existential.

2.3.4 *The general case.*

Finally, we proceed to the general case where a value can refer to an arbitrary number of channels, which is not a-priori fixed by the type. We exhibit and discuss the type of a general send function *gsend* and show how to obtain the previous examples by instantiation.

$$\begin{array}{ll}
\mathit{gsend}\colon & \forall(n\colon \mathsf{Shape}).\forall(\alpha\colon \mathsf{Dom}(n)).\\
& \forall(\hat\Sigma\colon \mathsf{Dom}(n) \to \mathsf{State}).\forall(\hat T\colon \mathsf{Dom}(n) \to \mathsf{Type}).\\
& \forall(c\colon \mathsf{Dom}(\mathbb{X})).\ (\alpha \# c) \Rightarrow \forall(\sigma\colon \mathsf{Session}).\\
& (\,\cdot\,;\, \hat T\,\alpha \to \,\cdot\,;\, (\hat\Sigma\,\alpha, c \mapsto !(\exists\alpha\colon \mathsf{Dom}(n).\hat\Sigma\,\alpha;\, \hat T\,\alpha).\sigma;\, \mathsf{Chan}\,c \to c \mapsto \sigma;\, \mathsf{Unit}))
\end{array} \tag{7}$$

As the shape is not fixed, we abstract over the shape and the corresponding domain. As the state depends on the domain α, we supply it as a closed function $\hat\Sigma$ from the domain so that its components can only be constructed from the domain elements. We supply the type in the same way as a closed function $\hat T$ from the domain. The remaining quantification over the channel name and the continuation session is as usual. The disjointness constraint forces the channel name to be different from any name in α. In the body of the type we have a function that takes an argument of type $\hat T\,\alpha$. It returns a function that takes a channel c along with the resources provided by the state $\hat\Sigma\,\alpha$. It returns the updated channel type and removes the resources which are on the way to the receiver.

The previous examples correspond to the following instantiations of *gsend*:

- *send0* = *gsend* $\mathbb{I}\ *\ (\lambda\_.\cdot)\ (\lambda\_.\,\mathsf{Int})$ where $*\colon\ \mathsf{Dom}(\mathbb{I})$ is the unique value of this type;
- *send1* = $\Lambda(\alpha\colon \mathsf{Dom}(\mathbb{X}))$.
  gsend $\mathbb{X}\ \alpha\ (\lambda\alpha.\alpha \mapsto S')\ (\lambda\alpha.\,\mathsf{Chan}\,\alpha)$;
- *send2* = $\Lambda(\alpha\colon \mathsf{Dom}(\mathbb{X})).\Lambda(\beta\colon \mathsf{Dom}(\mathbb{X}))$.
  gsend $(\mathbb{X} \mathbin{⨟} \mathbb{X})\ (\alpha, \beta)\ (\lambda\gamma.\pi_1\,\gamma \mapsto S', \pi_2\,\gamma \mapsto S'')\ (\lambda\gamma.\,\mathsf{Chan}\,(\pi_1\,\gamma) \times \mathsf{Chan}\,(\pi_2\,\gamma))$

2.4 Channel Aliasing

The VGR paper proposes the following function sendSend.

```
fun sendSend u v = send 1 on u; send 2 on v
```

It takes two channels and sends a number on each. This use is reflected in the following typing.

$$\texttt{sendSend} : \Sigma_1; \textbf{Chan}\ u \longrightarrow (\Sigma_1; \textbf{Chan}\ v \longrightarrow \textbf{Unit}; \Sigma_2); \Sigma_1 \tag{8}$$

with $\Sigma_1 = \{u : !\,\mathsf{Int}.S_u, v : !\,\mathsf{Int}.S_v\}$ and $\Sigma_2 = \{u : S_u, v : S_v\}$.

Ignoring the types we observe that it would be semantically sound to pass a reference to the same channel `w`, say, of session type `!Int.!Int.End` for `u` and `v`. However, `sendSend w w` does not type check with the type in (8) because `w` would have to have identity u and v at the same time, but state formation mandates they must be different.

In PolyVGR, the type for `sendSend` would be universally quantified over the channel names:

$$\begin{array}{l}\forall(\alpha\colon \mathsf{Dom}(\mathbb{X})).\forall(\beta\colon \mathsf{Dom}(\mathbb{X})).\ (\alpha \# \beta) \Rightarrow \forall(\sigma_1\colon \mathsf{Session}).\forall(\sigma_2\colon \mathsf{Session}).\\ (\,\cdot\,; \mathsf{Chan}\,\alpha \longrightarrow \cdot\,; (\left\{\begin{array}{l}\alpha \mapsto !(\exists\alpha\colon \mathsf{Dom}(\mathbb{I}).\,\cdot\,; \mathsf{Int}).\sigma_1,\\ \beta \mapsto !(\exists\alpha\colon \mathsf{Dom}(\mathbb{I}).\,\cdot\,; \mathsf{Int}).\sigma_2\end{array}\right\}; \mathsf{Chan}\,\beta \longrightarrow \left\{\begin{array}{l}\alpha \mapsto \sigma_1,\\ \beta \mapsto \sigma_2\end{array}\right\}; \mathsf{Unit}))\end{array}$$

Wellformedness of states requires that α and β are different because they index the same state. Hence, `sendSend w w` does not type check in PolyVGR, either.

Another VGR typing of the same code would be $\texttt{sendSend} : \Sigma_1; \textbf{Chan}\ w \longrightarrow (\Sigma_1; \textbf{Chan}\ w \longrightarrow \textbf{Unit}; \Sigma_2); \Sigma_1$ with $\Sigma_1 = \{w : !\,\mathsf{Int}.!\,\mathsf{Int}.S_w\}$ and $\Sigma_2 = \{w : S_w\}$. With this typing, `sendSend w w` type checks. Indeed, the typing forces the two arguments to be aliases!

A similar type could be given in PolyVGR:

$$\begin{array}{l}\forall(\alpha\colon \mathsf{Dom}(\mathbb{X})).\forall(\sigma\colon \mathsf{Session}).\\ (\,\cdot\,; \mathsf{Chan}\,\alpha \longrightarrow \cdot\,; (\alpha \mapsto !(\exists\alpha\colon \mathsf{Dom}(\mathbb{I}).\,\cdot\,; \mathsf{Int}).!(\exists\alpha\colon \mathsf{Dom}(\mathbb{I}).\,\cdot\,; \mathsf{Int}).\sigma; \mathsf{Chan}\,\alpha \longrightarrow \alpha \mapsto \sigma; \mathsf{Unit}))\end{array}$$

Presently it is not possible to give a single type to both aliased and unaliased uses of the function.

2.5 Higher-Order Functions

The preceding subsections have shown that VGR lacks facilities for abstraction. In this subsection, we give further indication of the flexibility of our system by discussing different typings for a simple higher-order function.

Consider a higher-order function that is a prototype for a protocol adapter. Given an argument function that runs a protocol, the adapter adds a prefix to the protocol, perhaps for authentication or accounting. To keep our example simple, the prefix consists of sending a single number, but more elaborate protocols are possible. The implementation is straightforward:

```
fun adapter f c =
  send 32168 on c;
  f c
```

The first type for `f` combines the channel creation pattern from Section 2.1 with the flexibility of supporting arbitrarily many channels as in Section 2.3.4.

$$\begin{array}{l}\forall(n\colon \mathsf{Shape}).\forall(\hat{\Sigma}\colon \mathsf{Dom}(n) \longrightarrow \mathsf{State}).\forall(\hat{T}\colon \mathsf{Dom}(n) \longrightarrow \mathsf{Type}).\\ \forall(\gamma\colon \mathsf{Dom}(\mathbb{X})).\forall(\sigma\colon \mathsf{Session}).\forall(\sigma'\colon \mathsf{Session}).\\ (\,\cdot\,; (\gamma \mapsto \sigma; \mathsf{Chan}\,\gamma \longrightarrow \exists\alpha\colon\ \mathsf{Dom}(n).\hat{\Sigma}\ \alpha, \gamma \mapsto \sigma'; \hat{T}\ \alpha) \longrightarrow\\ \qquad \cdot\,; (\gamma \mapsto !(\exists\alpha\colon \mathsf{Dom}(\mathbb{I}).\,\cdot\,; \mathsf{Int}).\sigma; \mathsf{Chan}\,\gamma \longrightarrow \exists\alpha\colon\ \mathsf{Dom}(n).\hat{\Sigma}\ \alpha, \gamma \mapsto \sigma'; \hat{T}\ \alpha))\end{array} \tag{9}$$

With this PolyVGR type, the function parameter `f` can create arbitrary many channels and it can return arbitrary values that may or may not include channels. In the degenerate case where n is $\mathbb{I}$, the universal quantification $\forall(\hat{T}\colon \mathsf{Dom}(\mathbb{I}) \longrightarrow \mathsf{Type}).\ldots$ quantifies over types that do not contain channel references. Existentially quantified variables, like α, carry an implicit disjointness constraint with any other domain variable in scope. This constraint ensures that $\hat{\Sigma}\ \alpha, \gamma \mapsto \sigma'$ is wellformed.

However, there is an issue that the type in (9) does not address. The function parameter f cannot be closed over further channels! To address that shortcoming requires another style of parameterization over the incoming and the outgoing state of f. The shapes of these states are unknown and the may be different, so we need two shape parameters. Moreover, these two states never mix, so their domains α' and α'' need *not* be disjoint. For simplicity, we first consider functions that do not create new channels, although both parameterizations can be combined.

$$\begin{aligned}
&\forall(n'\colon \mathsf{Shape}).\forall(\hat{\Sigma}'\colon \mathsf{Dom}(n') \to \mathsf{State}).\\
&\forall(n''\colon \mathsf{Shape}).\forall(\hat{\Sigma}''\colon \mathsf{Dom}(n'') \to \mathsf{State}).\forall(\hat{T}''\colon \mathsf{Dom}(n'') \to \mathsf{Type}).\\
&\forall(\alpha'\colon \mathsf{Dom}(n')).\forall(\alpha''\colon \mathsf{Dom}(n'')).\forall(\gamma\colon \mathsf{Dom}(\mathbb{X})).\ (\alpha' \# \gamma, \alpha'' \# \gamma) \Rightarrow\\
&\forall(\sigma\colon \mathsf{Session}).\forall(\sigma'\colon \mathsf{Session}).\\
&(\ \cdot\ ; (\hat{\Sigma}'\ \alpha', \gamma \mapsto \sigma; \mathsf{Chan}\,\gamma \to \hat{\Sigma}''\ \alpha'', \gamma \mapsto \sigma'; \hat{T}''\ \alpha'') \to\\
&\qquad \cdot\ ; (\hat{\Sigma}'\ \alpha', \gamma \mapsto\ !(\exists\alpha\colon \mathsf{Dom}(\mathbb{I}).\ \cdot\ ; \mathsf{Int}).\sigma; \mathsf{Chan}\,\gamma \to \hat{\Sigma}''\ \alpha'', \gamma \mapsto \sigma'; \hat{T}''\ \alpha''))
\end{aligned} \tag{10}$$

To parameterize over functions with arbitrary free channels and which may create channels, we need one more ingredient to describe the shape of the new state that contains the descriptions of the newly created channels. This extra shape and its use is highlighted in the type.

$$\begin{aligned}
&\forall(n'\colon \mathsf{Shape}).\forall(\hat{\Sigma}'\colon \mathsf{Dom}(n') \to \mathsf{State}).\\
&\forall(n''\colon \mathsf{Shape}).\\
&\forall(n\colon \mathsf{Shape}).\forall(\hat{\Sigma}''\colon \mathsf{Dom}(n \fatsemi n'') \to \mathsf{State}).\forall(\hat{T}''\colon \mathsf{Dom}(n \fatsemi n'') \to \mathsf{Type}).\\
&\forall(\alpha'\colon \mathsf{Dom}(n')).\forall(\alpha''\colon \mathsf{Dom}(n'')).\forall(\gamma\colon \mathsf{Dom}(\mathbb{X})).\ (\alpha' \# \gamma, \alpha'' \# \gamma) \Rightarrow\\
&\forall(\sigma\colon \mathsf{Session}).\forall(\sigma'\colon \mathsf{Session}).\\
&(\ \cdot\ ; (\hat{\Sigma}'\ \alpha', \gamma \mapsto \sigma; \mathsf{Chan}\,\gamma \to \exists\alpha\colon\ \mathsf{Dom}(n).\hat{\Sigma}''\ (\alpha, \alpha''), \gamma \mapsto \sigma'; \hat{T}''\ (\alpha, \alpha'')) \to\\
&\qquad \cdot\ ; (\hat{\Sigma}'\ \alpha', \gamma \mapsto\ !(\exists\alpha\colon \mathsf{Dom}(\mathbb{I}).\ \cdot\ ; \mathsf{Int}).\sigma; \mathsf{Chan}\,\gamma \to \exists\alpha\colon\ \mathsf{Dom}(n).\hat{\Sigma}''\ (\alpha, \alpha''), \gamma \mapsto \sigma'; \hat{T}''\ (\alpha, \alpha'')))
\end{aligned} \tag{11}$$

This example relies on shape combination with the _ ⨟ _ operator: in this case, the shape of the state captured in the closure and the shape of the state of newly created channels. Domain variables are combined accordingly using the _, _ operator.

A remaining restriction is that the number of channels that are handled is always fixed at compile time. Lifting this restriction would go along with support for recursive datatypes, a topic of future work that we outline in Section ??.

2.6 Executive Summary

PolyVGR offers the following key benefits over previous work.

- A function can be applied to different channel arguments if its type is polymorphic over channel names (see (4)).
- A function can abstract over the creation of an arbitrary number of channels because the names of newly created channels are existentially quantified (see (3) and (9)).
- Arbitrary data structures can be transmitted. Ownership of all channels contained in the data structure is transferred to the receiver (see (7)).
- Abstraction over transmission operations is possible. In particular, a type can be given to a fully flexible send or receive operation (see (7)).

Kinds	$K ::= \mathsf{Type} \mid \mathsf{Session} \mid \mathsf{State} \mid \mathsf{Shape} \mid \mathsf{Dom}(N) \mid K \to K$
Labels	$\ell ::= 1 \mid 2$
Types	$T, S, N, D, \Sigma ::= \alpha \mid T\,T \mid \lambda(\alpha\colon \mathsf{Dom}(N)).T \mid$
Expression Types	$\forall(\alpha\colon K).\,\mathbb{C} \Rightarrow T \mid (\Sigma;\, T \to \exists\Gamma.\Sigma;\, T) \mid$
	$\mathsf{Chan}\,D \mid [S] \mid \mathsf{Unit} \mid T \times T \mid$
Session Types	$!(\exists\alpha\colon \mathsf{Dom}(N).\Sigma;\, T).S \mid ?(\exists\alpha\colon \mathsf{Dom}(N).\Sigma;\, T).S \mid$
	$S \oplus S \mid S \,\&\, S \mid \mathsf{End} \mid \overline{S} \mid$
Shapes	$\mathbb{I} \mid \mathbb{X} \mid N \fatsemi N \mid$
Domains	$* \mid D, D \mid \pi_\ell\, D \mid$
Session State	$\cdot \mid D \mapsto S \mid \Sigma, \Sigma$
Type Environments	$\Gamma, \mathbb{C} ::= \cdot \mid \Gamma, x : T \mid \Gamma, \alpha : K \mid \Gamma, D \# D$
Expressions	$e ::= v \mid \mathsf{let}\, x = e \,\mathsf{in}\, e \mid v\,v \mid \pi_\ell\, v \mid v[T] \mid$
	$\mathsf{fork}\, v \mid \mathsf{accept}\, v \mid \mathsf{request}\, v \mid \mathsf{send}\, v \,\mathsf{on}\, v \mid \mathsf{receive}\, v \mid$
	$\mathsf{select}\, \ell \,\mathsf{on}\, v \mid \mathsf{case}\, v \,\mathsf{of}\, \{e;\, e\} \mid \mathsf{close}\, v \mid \mathsf{new}\, S$
Values	$v ::= x \mid \mathsf{chan}\,\alpha \mid \lambda(\Sigma;\, x\colon T).e \mid \Lambda(\alpha\colon K).\,\mathbb{C} \Rightarrow v \mid \mathsf{unit} \mid (v, v)$
Configurations	$C ::= e \mid (C \parallel C) \mid \nu\alpha, \alpha \mapsto S.\, C \mid \nu x\colon [S].\, C$
Expression Contexts	$\mathcal{E} ::= \square \mid \mathsf{let}\, x = \mathcal{E} \,\mathsf{in}\, e$
Configuration Contexts	$\mathcal{C} ::= \square \mid \nu\alpha, \alpha \mapsto S.\, \mathcal{C} \mid \nu x\colon [S].\, \mathcal{C} \mid (\mathcal{C} \parallel C)$

Fig. 1. Syntax of PolyVGR

3 POLYMORPHIC IMPERATIVE SESSION TYPES

This section introduces PolyVGR formally. We start with the syntax, proceed to the static sematics for types, then expressions. We finish with defining the notions of expression reduction and process reduction.

3.1 Syntax

Figure 1 defines the syntax starting with kinds and types. We use different metavariables for types indicating their kinds with T as a fallback. Kinds K distinguish between plain types (Type), session types (Session ranged over by metavariable S), states (State ranged over by Σ), shapes (Shape ranged over by N), domains ($\mathsf{Dom}(N)$ ranged over by D), and arrow kinds. The kind for domains depends on types in the form of shapes. This dependency as well as the introduction rules for arrow kinds are very limited as they are tailored to express channel references as discussed in Section 2.3.

The type language comprises variables α, application, and abstraction over domains to support arrow kinds. Universal quantification over types of any kind is augmented with constraints $\mathbb{C}$, function types contain pre- and post-states as well as existential quantification as explained in Section 2.1. There are channel references that refer to a domain, access points that refer to a session type, the unit type (representing base types), and products to characterize the values of expressions. Session type comprise sending and receiving (cf. Section 2.3), as well as choice and branch types limited to two alternatives, End to indicate the end of a protocol, and $\overline{S}$ to indicate the dual of a session type (which flips sending and receiving operations). Shapes comprise the empty shape $\mathbb{I}$, the single-channel shape $\mathbb{X}$, and the combination of two shapes $\_ \mathbin{⨟} \_$. The corresponding domains are the empty domain $*$, the combination of two domains $\_, \_$, and the first/second projection of a domain. The latter selects a component of a combined domain. A session state can be empty, a binding of a single-channel domain to a session type, or a combination of states. Most of the time, the domain in the binding is a variable.

Type environments Γ contain bindings for expression variables and type variables, as well as disjointness constraints between domains. The same syntax is used for constraints $\mathbb{C}$.

Following VGR [Vasconcelos et al. 2006], the expression language is structured in A-normal form [Flanagan et al. 1993], which means that the subterms of each non-value expression are syntactic values v and sequencing of execution is expressed using a single let expression. This choice simplifies the dynamics as there is only one kind of evaluation context $\mathcal{E}$, which selects the header expression of a let. The type system performs best (i.e., it is most permissive) on expressions in strict A-normal form, where the body of a let is either another let or a syntactic value. Any expression can be transformed into strict A-normal form with a simple variation of the standard transformation from the literature. Strict A-normal form is closed under reduction.

Besides values and the let expression, there is function application, projecting a pair, type application, fork to start processes, accepting and requesting a channel, sending and receiving, selection (i.e. sending) of a label and branching on a received label, and closing a channel.

Values are variables, channel references, lambda abstractions, type abstractions with constraints — their body is restricted to a value to avoid unsoundness in the presence of effects, the unit value, and pairs of values.

Configurations C describes processes. They are either expression processes, parallel processes, channel abstraction — it abstracts the two ends of a channel at once, and access point creation.

We already discussed expression contexts. Configuration contexts $\mathcal{C}$ enable reduction in any configuration context, also under channel and access point abstractions.

3.2 Statics for types

Many of the judgments defining the type-level statics are mutually recursive. For example, the first three judgments we cover are

- context formation $\vdash \Gamma$,
- kind formation $\Gamma \vdash K$,
- type formation $\Gamma \vdash T : K$.

All judgments depend on context formation, which unavoidably depends on kind and type formation. Based on these notions we define

- type conversion $T \equiv T$,
- constraint entailment $\Gamma \vdash \mathbb{C}$,
- context restriction operators $\lfloor \Gamma \rfloor$ and $\lceil \Gamma \rceil$,
- disjoint context extension operator $\Gamma ,_{\#} \Gamma$.

Context formation (Figure 2) is standard up to the case for disjointness constraints. For those, we have to show that each domain is wellformed with respect to the current context Γ, which may be needed to construct the shape and then the domain.

Kind formation (Figure 3) is straightforward: most kinds are constants, domains must be indexed by shapes, arrow kinds are standard.

Figures 4 and 5 contain the rules for type formation and kinding. The rules for variables and application are standard. Abstractions (rule K-Lam) are severely restricted. Their argument must be a domain and their result must be Type or Shape. Moreover, the body can only refer to the

CF-Empty
$$\vdash \cdot$$

CF-ConsKind
$$\frac{\vdash \Gamma \qquad \Gamma \vdash K \qquad \alpha \notin \mathrm{dom}(\Gamma)}{\vdash \Gamma, \alpha : K}$$

CF-ConsType
$$\frac{\vdash \Gamma \qquad \Gamma \vdash T : \mathsf{Type} \qquad x \notin \mathrm{dom}(\Gamma)}{\vdash \Gamma, x : T}$$

CF-ConsCstr
$$\frac{\vdash \Gamma \qquad \Gamma \vdash D_1 : \mathsf{Dom}(N_1) \qquad \Gamma \vdash D_2 : \mathsf{Dom}(N_2)}{\vdash \Gamma, D_1 \mathbin{\#} D_2}$$

Fig. 2. Context formation ($\vdash \Gamma$)

KF-Type
$$\Gamma \vdash \mathsf{Type}$$

KF-Session
$$\Gamma \vdash \mathsf{Session}$$

KF-State
$$\Gamma \vdash \mathsf{State}$$

KF-Shape
$$\Gamma \vdash \mathsf{Shape}$$

KF-Dom
$$\frac{\Gamma \vdash N : \mathsf{Shape}}{\Gamma \vdash \mathsf{Dom}(N)}$$

KF-Arr
$$\frac{\Gamma \vdash K_1 \qquad \Gamma \vdash K_2}{\Gamma \vdash K_1 \longrightarrow K_2}$$

Fig. 3. Kind formation ($\Gamma \vdash K$)

argument domain; all other domains are removed from the assumptions. Constrained universal quantification (rule K-All) is standard.

To form a function type, rule K-Arr asks that the argument state and type are wellformed with respect to the assumptions. The return state and type must be wellformed with respect to the assumptions extended with the state Γ_2 of channels created by the function. This state must be disjoint from the assumptions as indicated by $\Gamma_1 ,_{\#} \Gamma_2$ (see Figure 9). We also make sure that Γ_2 only contains domains.

A channel type can be formed from any single-channel domain of shape $\mathbb{X}$ (rule K-Chan). The rules for access points, unit, and pairs are straightforward and standard.

The rule K-Send and K-Recv control wellformedness of sending and receiving types. In both cases, we require that both the state and the type describing the transmitted value can only reference the domain abstracted in the existential. This restriction is necessary to enforce proper transfer of channel ownership between sender and receiver.

The remaining rules for session types are standard.

Figure 5 contains the rules for shapes, domains, and states. We discussed shapes with their syntax already. The domain rules are similar to product rules with the additional disjointness constraint on the components of the combined domain. Empty states are trivially wellformed. A single binding is wellformed if it maps a single-channel domain to a session type.

Figure 6 defines type conversion, where we omit the standard rules for reflexivity, transitivity, symmetry, and congruence. Conversion comprises beta reduction for functions and pairs, as well as the simplification of the dual operator: End is self-dual, the dual operator is involutory, for sending/receiving as well as for choice/branch the dual operator flips the direction of the communication.

Conversion is needed in the context of the dual operator, because a programmer may use the dual operator in a type. If this type is polymorphic over a session-kinded type variable α, then the operator cannot be fully eliminated as in $\overline{\alpha}$. Once a type application instantiates α, we invoke conversion to enable pushing the dual operator further down into the session type.

The conversion judgment does not destroy the simple inversion properties of the expression and value typing rules as it is explicitly invoked in just two expression typing rules: T-Send for the $\mathsf{send} \cdot \mathsf{on} \cdot$ operation and T-TApp for type application (see Figure 11).

K-Var
$$\Gamma, \alpha : K \vdash \alpha : K$$

K-App
$$\frac{\Gamma \vdash T_1 : K_1 \to K_2 \qquad \Gamma \vdash T_2 : K_1}{\Gamma \vdash T_1\ T_2 : K_2}$$

K-Lam
$$\frac{\Gamma \vdash N : \mathsf{Shape} \qquad \lfloor \Gamma \rfloor, \alpha : \mathsf{Dom}(N) \vdash T : K \qquad K \in \{\mathsf{Type}, \mathsf{State}\}}{\Gamma \vdash \lambda(\alpha\colon \mathsf{Dom}(N)).T : \mathsf{Dom}(N) \to K}$$

K-All
$$\frac{\vdash \Gamma, \alpha : K, \mathbb{C} \qquad \Gamma, \alpha : K, \mathbb{C} \vdash T : \mathsf{Type}}{\Gamma \vdash \forall(\alpha\colon K).\, \mathbb{C} \Rightarrow T : \mathsf{Type}}$$

K-Arr
$$\frac{\begin{array}{cc} \Gamma_1 \vdash \Sigma_1 : \mathsf{State} & \Gamma_1 \vdash T_1 : \mathsf{Type} \\ \Gamma_1 ,_{\#} \Gamma_2 \vdash \Sigma_2 : \mathsf{State} & \Gamma_1 ,_{\#} \Gamma_2 \vdash T_2 : \mathsf{Type} \\ \vdash \Gamma_1 ,_{\#} \Gamma_2 & \Gamma_2 = \lceil \Gamma_2 \rceil \end{array}}{\Gamma_1 \vdash (\Sigma_1;\ T_1 \to \exists \Gamma_2.\Sigma_2;\ T_2) : \mathsf{Type}}$$

K-Chan
$$\frac{\Gamma \vdash D : \mathsf{Dom}(\mathbb{X})}{\Gamma \vdash \mathsf{Chan}\, D : \mathsf{Type}}$$

K-AccessPoint
$$\frac{\Gamma \vdash S : \mathsf{Session}}{\Gamma \vdash [S] : \mathsf{Type}}$$

K-Unit
$$\frac{}{\Gamma \vdash \mathsf{Unit} : \mathsf{Type}}$$

K-Pair
$$\frac{\Gamma \vdash T_1 : \mathsf{Type} \qquad \Gamma \vdash T_2 : \mathsf{Type}}{\Gamma \vdash T_1 \times T_2 : \mathsf{Type}}$$

K-Send
$$\frac{\lfloor \Gamma \rfloor, \alpha : \mathsf{Dom}(N) \vdash \Sigma : \mathsf{State} \qquad \begin{array}{c} \Gamma \vdash N : \mathsf{Shape} \\ \lfloor \Gamma \rfloor, \alpha : \mathsf{Dom}(N) \vdash T : \mathsf{Type} \end{array} \qquad \Gamma \vdash S : \mathsf{Session}}{\Gamma \vdash\ !(\exists \alpha\colon \mathsf{Dom}(N).\Sigma;\ T).S : \mathsf{Session}}$$

K-Recv
$$\frac{\lfloor \Gamma \rfloor, \alpha : \mathsf{Dom}(N) \vdash \Sigma : \mathsf{State} \qquad \begin{array}{c} \Gamma \vdash N : \mathsf{Shape} \\ \lfloor \Gamma \rfloor, \alpha : \mathsf{Dom}(N) \vdash T : \mathsf{Type} \end{array} \qquad \Gamma \vdash S : \mathsf{Session}}{\Gamma \vdash\ ?(\exists \alpha\colon \mathsf{Dom}(N).\Sigma;\ T).S : \mathsf{Session}}$$

K-Branch
$$\frac{\Gamma \vdash S_1 : \mathsf{Session} \qquad \Gamma \vdash S_2 : \mathsf{Session}}{\Gamma \vdash S_1 \,\&\, S_2 : \mathsf{Session}}$$

K-Choice
$$\frac{\Gamma \vdash S_1 : \mathsf{Session} \qquad \Gamma \vdash S_2 : \mathsf{Session}}{\Gamma \vdash S_1 \oplus S_2 : \mathsf{Session}}$$

K-End
$$\Gamma \vdash \mathsf{End} : \mathsf{Session}$$

K-Dual
$$\frac{\Gamma \vdash S : \mathsf{Session}}{\Gamma \vdash \overline{S} : \mathsf{Session}}$$

Fig. 4. Type formation, Part I ($\Gamma \vdash T : K$)

Constraint entailment is defined entirely structurally in Figure 7. Disjointness of domains can hold by assumption. Disjointness is symmetric. The empty domain is disjoint with any other domain. Disjointness distributes over combination of domains and is compatible with projections. It extends to conjunctions of constraints in the obvious way.

Figure 8 contains the definition of the context restriction operators, which are mainly technical. Both operators keep only bindings of type variables. One removes all domain bindings and the other removes all non-domain bindings.

$$\textsc{K-ShapeEmpty}\quad \Gamma \vdash \mathbb{I} : \mathsf{Shape} \qquad \textsc{K-ShapeChan}\quad \Gamma \vdash \mathbb{X} : \mathsf{Shape} \qquad \textsc{K-ShapePair}\quad \frac{\Gamma \vdash N_1 : \mathsf{Shape} \qquad \Gamma \vdash N_2 : \mathsf{Shape}}{\Gamma \vdash N_1 \fatsemi N_2 : \mathsf{Shape}} \qquad \textsc{K-DomEmpty}\quad \Gamma \vdash * : \mathsf{Dom}(\mathbb{I})$$

$$\textsc{K-DomMerge}\quad \frac{\Gamma \vdash D_1 : \mathsf{Dom}(N_1) \qquad \Gamma \vdash D_2 : \mathsf{Dom}(N_2) \qquad \Gamma \vdash D_1 \# D_2}{\Gamma \vdash D_1, D_2 : \mathsf{Dom}(N_1 \fatsemi N_2)} \qquad \textsc{K-DomProj}\quad \frac{\Gamma \vdash D : \mathsf{Dom}(N_1 \fatsemi N_2)}{\Gamma \vdash \pi_\ell\, D : \mathsf{Dom}(N_\ell)}$$

$$\textsc{K-StEmpty}\quad \Gamma \vdash \cdot : \mathsf{State} \qquad \textsc{K-StChan}\quad \frac{\Gamma \vdash D : \mathsf{Dom}(\mathbb{X}) \qquad \Gamma \vdash S : \mathsf{Session}}{\Gamma \vdash D \mapsto S : \mathsf{State}}$$

$$\textsc{K-StMerge}\quad \frac{\Gamma \vdash \Sigma_1 : \mathsf{State} \qquad \Gamma \vdash \Sigma_2 : \mathsf{State} \qquad \Gamma \vdash \mathrm{dom}(\Sigma_1) \# \mathrm{dom}(\Sigma_2)}{\Gamma \vdash \Sigma_1, \Sigma_2 : \mathsf{State}}$$

Fig. 5. Type formation, Part II ($\Gamma \vdash T : K$)

$$\textsc{TC-TApp}\quad (\lambda(\alpha\colon \mathsf{Dom}(N)).T_1)\ T_2 \equiv \{T_2/\alpha\}T_1 \qquad \textsc{TC-Proj}\quad \pi_\ell\,(D_1, D_2) \equiv D_\ell \qquad \textsc{TC-DualEnd}\quad \overline{\mathsf{End}} \equiv \mathsf{End} \qquad \textsc{TC-DualVar}\quad \overline{\overline{\alpha}} \equiv \alpha$$

$$\textsc{TC-DualSend}\quad \overline{!(\exists\alpha\colon \mathsf{Dom}(N).\Sigma;\ T).S} \equiv ?(\exists\alpha\colon \mathsf{Dom}(N).\Sigma;\ T).\overline{S} \qquad \textsc{TC-DualChoice}\quad \overline{S_1 \oplus S_2} \equiv \overline{S_1} \,\&\, \overline{S_2}$$

$$\textsc{TC-DualRecv}\quad \overline{?(\exists\alpha\colon \mathsf{Dom}(N).\Sigma;\ T).S} \equiv !(\exists\alpha\colon \mathsf{Dom}(N).\Sigma;\ T).\overline{S} \qquad \textsc{TC-DualBranch}\quad \overline{S_1 \,\&\, S_2} \equiv \overline{S_1} \oplus \overline{S_2}$$

Fig. 6. Type conversion ($T \equiv T$)

$$\textsc{CE-Axiom}\quad \Gamma, D_1 \# D_2 \vdash D_1 \# D_2 \qquad \textsc{CE-Sym}\quad \frac{\Gamma \vdash D_2 \# D_1}{\Gamma \vdash D_1 \# D_2} \qquad \textsc{CE-Empty}\quad \Gamma \vdash D \# * \qquad \textsc{CE-Split}\quad \frac{\Gamma \vdash D \# (D_1, D_2)}{\Gamma \vdash D \# D_1 \qquad \Gamma \vdash D \# D_2}$$

$$\textsc{CE-Merge}\quad \frac{\Gamma \vdash D \# D_1 \qquad \Gamma \vdash D \# D_2}{\Gamma \vdash D \# (D_1, D_2)} \qquad \textsc{CE-ProjMerge}\quad \frac{\Gamma \vdash D_1 \# \pi_1\, D_2 \qquad \Gamma \vdash D_1 \# \pi_2\, D_2}{\Gamma \vdash D_1 \# D_2} \qquad \textsc{CE-ProjSplit}\quad \frac{\Gamma \vdash D_1 \# D_2}{\Gamma \vdash D_1 \# \pi_\ell\, D_2}$$

$$\textsc{CE-Empty}\quad \Gamma \vdash \cdot \qquad \textsc{CE-Cons}\quad \frac{\Gamma \vdash \mathbb{C} \qquad \Gamma \vdash D_1 \# D_2}{\Gamma \vdash \mathbb{C}, D_1 \# D_2}$$

Fig. 7. Constraint entailment ($\Gamma \vdash \mathbb{C}$)

Removing bindings, which might contain free domain variables

$$\lfloor \Gamma \rfloor = \begin{cases} \cdot & \text{if } \Gamma = \cdot \\ \lfloor \Gamma' \rfloor, \alpha : K & \text{if } \Gamma = \Gamma', \alpha : K \wedge K \in \{\mathsf{Shape}, \mathsf{Session}, \mathsf{Dom}(N) \longrightarrow \mathsf{Type}, \mathsf{Dom}(N) \longrightarrow \mathsf{State}\} \\ \lfloor \Gamma' \rfloor & \text{if } \Gamma = \Gamma', \alpha : K \wedge K \in \{\mathsf{Dom}(N), \mathsf{State}, \mathsf{Type}\} \\ \lfloor \Gamma' \rfloor & \text{if } \Gamma = \Gamma', x : T \\ \lfloor \Gamma' \rfloor & \text{if } \Gamma = \Gamma', D_1 \# D_2 \end{cases}$$

Removing non-domain bindings

$$\lceil \Gamma \rceil = \begin{cases} \cdot & \text{if } \Gamma = \cdot \\ \lceil \Gamma' \rceil, \alpha : \mathsf{Dom}(N) & \text{if } \Gamma = \Gamma', \alpha : \mathsf{Dom}(N) \\ \lceil \Gamma' \rceil & \text{if } \Gamma = \Gamma', \alpha : K \wedge K \neq \mathsf{Dom}(N) \\ \lceil \Gamma' \rceil & \text{if } \Gamma = \Gamma', x : T \\ \lceil \Gamma' \rceil & \text{if } \Gamma = \Gamma', D_1 \# D_2 \end{cases}$$

Fig. 8. Context restriction ($\lfloor \Gamma \rfloor$ and $\lceil \Gamma \rceil$)

$$\Gamma_1 ,_{\#} \Gamma_2 = \Gamma_1, \Gamma_2, \mathbb{C}_2, \mathbb{C}_{12} \text{ where}$$

$$\mathbb{C}_2 = \{\alpha_1 \# \alpha_2 \mid \alpha_1, \alpha_2 \in \mathrm{dom}(\lceil \Gamma_2 \rceil), \alpha_1 \neq \alpha_2\}$$
$$\mathbb{C}_{12} = \{\alpha_1 \# \alpha_2 \mid \alpha_1 \in \mathrm{dom}(\lceil \Gamma_1 \rceil), \alpha_2 \in \mathrm{dom}(\lceil \Gamma_2 \rceil)\}$$

Fig. 9. Disjoint context extension ($\Gamma ,_{\#} \Gamma$)

T-Var
$$\Gamma, x : T \vdash x : T$$

T-Unit
$$\Gamma \vdash \mathsf{unit} : \mathsf{Unit}$$

T-Pair
$$\frac{\Gamma \vdash v_1 : T_1 \qquad \Gamma \vdash v_2 : T_2}{\Gamma \vdash (v_1, v_2) : T_1 \times T_2}$$

T-TAbs
$$\frac{\Gamma \vdash \forall(\alpha : K).\, \mathbb{C} \Rightarrow T : \mathsf{Type} \qquad \Gamma, \alpha : K, \mathbb{C} \vdash v : T}{\Gamma \vdash \Lambda(\alpha : K).\, \mathbb{C} \Rightarrow v : \forall(\alpha : K).\, \mathbb{C} \Rightarrow T}$$

T-Chan
$$\frac{\Gamma \vdash D : \mathsf{Dom}(\mathbb{X})}{\Gamma \vdash \mathsf{chan}\, D : \mathsf{Chan}\, D}$$

T-Abs
$$\frac{\Gamma_1 \vdash (\Sigma_1;\ T_1 \longrightarrow \exists \Gamma_2.\Sigma_2;\ T_2) : \mathsf{Type} \qquad \Gamma_1, x : T_1; \Sigma_1 \vdash e : \exists \Gamma_2.\Sigma_2; T_2}{\Gamma_1 \vdash \lambda(\Sigma_1;\ x : T_1).e : (\Sigma_1;\ T_1 \longrightarrow \exists \Gamma_2.\Sigma_2;\ T_2)}$$

Fig. 10. Value typing ($\Gamma \vdash v : T$)

Figure 9 defines the operator $\Gamma_1 ,_{\#} \Gamma_2$. The assumption is that Γ_1 is known to contain disjoint bindings. The generated constraints $\mathbb{C}_2$ make sure that Γ_2's bindings are also disjoint and $\mathbb{C}_{12}$ ensures that they are also disjoint from Γ_1's bindings.

3.3 Statics for expressions and processes

We already indicated that the syntax is formulated in the style of A-normal form. Hence, there are three main judgments

- value typing $\Gamma \vdash v : T$,
- expression typing $\Gamma; \Sigma \vdash e : \exists\Gamma.\Sigma; T$, and
- process (or configuration) typing $\Gamma; \Sigma \vdash C$.

The rules in Figure 10 define the value typing judgment that applies to syntactic values. The most notable issue with these rules is that they do not handle states. As syntactic values have no effect, they cannot affect the state and this restriction is already stated in the typing judgment.

The rules for variables, unit, pairs, and type abstraction are standard. Channel values refer to single-channel domains. Rule T-Abs for lambda abstraction checks wellformedness of the function type with respect to the current environment and invokes expression typing to obtain the return state and type.

Figure 11 contains the rules for expression typing. We concentrate on the state-handling aspect as the value level is mostly standard. Recall that we assume expressions are in strict A-normal form, which means that every expression consists of a cascade of let expressions that ends in a syntactic value. Rule T-Val embeds values in expression typing. It is special as it threads the entire state Σ even though it makes no use of it. This special treatment is needed at the end of a let cascade because rule T-Let splits the incoming state for $\mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2$ into the part Σ_1 required by the header expression e_1 and Σ_2 for the continuation e_2, but then it feeds the entire outgoing state of e_1 combined with Σ_2 into the continuation e_2. All remaining rules only take the portion of the incoming state that is processed by the operation, so they are designed to be applied in the header position e_1 of a let. Thankfully, this use is guaranteed by strict A-normal form.

The remaining rules all assume the expression is used in header position of a let. Projection (rule T-Proj) requires no state. Type application (rule T-TApp) checks the constraints after instantiation and enables conversion of the instantiated type. Conversion is needed (among others) to expose the session type operators (see discussion for Figure 6).

Function application (rule T-App) just rewrites the function type to an expression judgment. The existential part of this judgment is reintegrated into the state in the T-Let rule, which inserts the necessary disjointness constraints via the disjoint append-operator $\_ ,_{\#} \_$. As the T-Let rule presents the function application exactly with the state it can handle, we must delay the creation of the constraints to the let-expression because it is here that the return state must be merged with the state for the continuation, which may contain additional domains. Given that the existentially bound domains are subject to α-renaming, we can freely impose the corresponding disjointness constraints to force local freshness of the domains. Explicit disjointness is required because of the axiomatic nature of our constraint system.

The new expression creates an access point which requires no state (rule T-New). The rules T-Request and T-Accept type the establishment of a connection via an access point. They return one end of the freshly created channel, so that the channel's domain is existentially quantified. The kind of this domain is $\mathsf{Dom}(\mathbb{X})$ (omitted in the rules as it is implied by the binding).

The rule T-Send for sending is particularly interesting. It splits the incoming state into the channel D on which the sending takes place and the state Σ, which will be passed along with the value. The rule guesses a domain D' such that the state expected in the session type matches the state Σ and the type expected by the session type matches the type of the provided argument. This matching is achieved with a type conversion judgment that implements reduction for functions and pairs at the type level (see Figure 6). The outgoing state only retains the channel D bound to the continuation session S.

T-Let

$$\frac{\Gamma_1;\Sigma_1 \vdash e_1 : \exists\Gamma_2.\Sigma_2';T_1 \qquad \Gamma_1 ,_{\#} \Gamma_2, x : T_1;\Sigma_2,\Sigma_2' \vdash e_2 : \exists\Gamma_3.\Sigma_3;T_2 \qquad \Gamma_1 ,_{\#} \Gamma_2, x : T_1 \vdash \Sigma_2,\Sigma_2' : \mathsf{State}}{\Gamma_1;\Sigma_1,\Sigma_2 \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \exists\Gamma_2,\Gamma_3.\Sigma_3;T_2}$$

T-Val

$$\frac{\Gamma \vdash v : T}{\Gamma;\Sigma \vdash v : \exists\cdot.\Sigma;T}$$

T-Proj

$$\frac{\Gamma \vdash v : T_1 \times T_2}{\Gamma;\cdot \vdash \pi_\ell\, v : \exists\cdot.\cdot;T_\ell}$$

T-App

$$\frac{\Gamma_1 \vdash v_1 : (\Sigma_1;\ T_1 \to \exists\Gamma_2.\Sigma_2;\ T_2) \qquad \Gamma_1 \vdash v_2 : T_1}{\Gamma_1;\Sigma_1 \vdash v_1\ v_2 : \exists\Gamma_2.\Sigma_2;T_2}$$

T-TApp

$$\frac{\Gamma \vdash v : \forall(\alpha\colon K).\ \mathbb{C} \Rightarrow T \qquad \Gamma \vdash T' : K \qquad \Gamma \vdash \{T'/\alpha\}\mathbb{C} \qquad \{T'/\alpha\}T \equiv T''}{\Gamma;\cdot \vdash v[T'] : \exists\cdot.\cdot;T''}$$

T-New

$$\frac{\Gamma \vdash S : \mathsf{Session}}{\Gamma;\cdot \vdash \mathsf{new}\, S : \exists\cdot.\cdot;[S]}$$

T-Request

$$\frac{\Gamma \vdash v : [S]}{\Gamma;\cdot \vdash \mathsf{request}\, v : \exists\alpha : \mathsf{Dom}(\mathbb{X}).\alpha \mapsto S;\mathsf{Chan}\,\alpha}$$

T-Accept

$$\frac{\Gamma \vdash v : [S]}{\Gamma;\cdot \vdash \mathsf{accept}\, v : \exists\alpha : \mathsf{Dom}(\mathbb{X}).\alpha \mapsto \overline{S};\mathsf{Chan}\,\alpha}$$

T-Send

$$\frac{\begin{array}{c}\Gamma \vdash D' : \mathsf{Dom}(N) \qquad \{D'/\alpha'\}\Sigma' \equiv \Sigma \qquad \{D'/\alpha'\}T' \equiv T \\ \Gamma \vdash D : \mathsf{Dom}(\mathbb{X}) \qquad \Gamma \vdash v_1 : T \qquad \Gamma \vdash v_2 : \mathsf{Chan}\,D\end{array}}{\Gamma;\Sigma, D \mapsto\ !(\exists\alpha'\colon \mathsf{Dom}(N).\Sigma';\ T').S \vdash \mathsf{send}\, v_1\ \mathsf{on}\ v_2 : \exists\cdot.D \mapsto S;\mathsf{Unit}}$$

T-Recv

$$\frac{\Gamma \vdash D : \mathsf{Dom}(\mathbb{X}) \qquad \Gamma \vdash v : \mathsf{Chan}\,D}{\Gamma;D \mapsto\ ?(\exists\alpha'\colon \mathsf{Dom}(N).\Sigma';\ T').S \vdash \mathsf{receive}\, v : \exists(\alpha' : \mathsf{Dom}(N)).\Sigma', D \mapsto S;T'}$$

T-Fork

$$\frac{\Gamma \vdash v : (\Sigma;\ \mathsf{Unit} \to \cdot;\ \mathsf{Unit})}{\Gamma;\Sigma \vdash \mathsf{fork}\, v : \exists\cdot.\cdot;\mathsf{Unit}}$$

T-Close

$$\frac{\Gamma \vdash v : \mathsf{Chan}\,D}{\Gamma;D \mapsto \mathsf{End} \vdash \mathsf{close}\, v : \exists\cdot.\cdot;\mathsf{Unit}}$$

T-Select

$$\frac{\Gamma \vdash v : \mathsf{Chan}\,D}{\Gamma;D \mapsto S_1 \oplus S_2 \vdash \mathsf{select}\,\ell\ \mathsf{on}\ v : \exists\cdot.D \mapsto S_\ell;\mathsf{Unit}}$$

T-Case

$$\frac{\Gamma_1 \vdash v : \mathsf{Chan}\,D \qquad (\forall\ell)\ \Gamma_1;\Sigma_1, D \mapsto S_\ell \vdash e_\ell : \exists\Gamma_2.\Sigma_2;T}{\Gamma_1;\Sigma_1, D \mapsto S_1\ \&\ S_2 \vdash \mathsf{case}\, v\ \mathsf{of}\,\{e_1;\ e_2\} : \exists\Gamma_2.\Sigma_2;T}$$

Fig. 11. Expression typing ($\Gamma;\Sigma \vdash e : \exists\Gamma.\Sigma;T$)

Receiving (rule T-Recv) is much simpler: we treat the received channels like new created one in the existential component of the typing judgment.

T-Exp
$$\frac{\Gamma; \Sigma \vdash e : \exists \Gamma'.\cdot; T}{\Gamma; \Sigma \vdash e}$$

T-Par
$$\frac{\Gamma; \Sigma_1 \vdash C_1 \qquad \Gamma; \Sigma_2 \vdash C_2}{\Gamma; \Sigma_1, \Sigma_2 \vdash C_1 \parallel C_2}$$

T-NuChan
$$\frac{\alpha, \alpha' \text{ not free in } \Gamma \qquad \Gamma \vdash S : \mathsf{Session} \qquad \Gamma \mathbin{,_{\#}} \alpha : \mathsf{Dom}(\mathbb{X}) \mathbin{,_{\#}} \alpha' : \mathsf{Dom}(\mathbb{X}); \Sigma, \alpha \mapsto S, \alpha' \mapsto \overline{S} \vdash C}{\Gamma; \Sigma \vdash \nu\alpha, \alpha' \mapsto S.\, C}$$

T-NuChanClosed
$$\frac{\alpha, \alpha' \text{ not free in } \Gamma \qquad \Gamma \mathbin{,_{\#}} \alpha : \mathsf{Dom}(\mathbb{X}) \mathbin{,_{\#}} \alpha' : \mathsf{Dom}(\mathbb{X}); \Sigma \vdash C}{\Gamma; \Sigma \vdash \nu\alpha, \alpha' \mapsto \mathsf{End}.\, C}$$

T-NuAccess
$$\frac{x \text{ not free in } \Gamma \qquad \Gamma \vdash S : \mathsf{Session} \qquad \Gamma, x : [S]; \Sigma \vdash C}{\Gamma; \Sigma \vdash \nu x \colon [S].\, C}$$

Fig. 12. Configuration typing ($\Gamma; \Sigma \vdash C$)

ER-BetaFun
$$(\lambda(\Sigma;\, x \colon T).e_1)\ v_2 \hookrightarrow_e \{v_2/x\}e_1$$

ER-BetaAll
$$(\Lambda(\alpha \colon K).\, \mathbb{C} \Rightarrow v)[T] \hookrightarrow_e \{T/\alpha\}v$$

ER-BetaLet
$$\mathsf{let}\ x = v_1\ \mathsf{in}\ e_2 \hookrightarrow_e \{v_1/x\}e_2$$

ER-BetaPair
$$\pi_\ell\ (v_1, v_2) \hookrightarrow_e v_\ell$$

ER-Lift
$$\frac{e_1 \hookrightarrow_e e_2}{\mathsf{let}\ x = e_1\ \mathsf{in}\ e \hookrightarrow_e \mathsf{let}\ x = e_2\ \mathsf{in}\ e}$$

Fig. 13. Expression reduction ($e \hookrightarrow_e e$)

Forking (rule T-Fork) starts a new process from a Unit $\rightarrow$ Unit function. The new process takes ownership of all incoming state. Closing a channel (rule T-Close) just requires a single channel with type End and returns an empty state.

Rule T-Select performs the standard rewrite of the session type for selecting a branch in the protocol. The dual rule T-Case is slightly more subtle. It requires that both branches end in the same state, that is, they must create channels and operate on open channels in the same way (or close them before returning from the branch).

Figure 12 contains the typing rules for *PolyVGR* processes. They are straightforward with one exception. In rule T-NuChan, we need to make sure that the newly introduced channel ends are disjoint (i.e., different) from each other and from previously defined domains. Rule T-NuChanClosed replaces T-NuChan after the channel is closed. The difference is that it no longer places the channels in the state Σ. This way, operations on the closed channel are disabled, but it is still possible to have references to it in dead code.

3.4 Dynamics

Figure 13 defines expression reduction, which is standard for a polymorphic call-by-value lambda calculus. Recall that an evaluation context just selects the header of a let expression.

Figure 14 defines a congruence relation on processes. This standard relation (process composition is commutative, associative with the unit process as a neutral element, and compatible with

$$\textsc{CC-Null}\quad C \parallel \mathsf{unit} \equiv C \qquad \textsc{CC-Comm}\quad C_1 \parallel C_2 \equiv C_2 \parallel C_1 \qquad \textsc{CC-Assoc}\quad C_1 \parallel (C_2 \parallel C_3) \equiv (C_1 \parallel C_2) \parallel C_3$$

$$\textsc{CC-Swap}\quad \nu\alpha,\alpha' \mapsto S.\, C \equiv \nu\alpha',\alpha \mapsto \overline{S}.\, C \qquad \textsc{CC-Lift}\quad \frac{C_1 \equiv C_2}{\mathcal{C}[C_1] \equiv \mathcal{C}[C_2]}$$

$$\textsc{CC-Scope-Chan}\quad \frac{\alpha,\alpha' \text{ not free in } C_1}{C_1 \parallel (\nu\alpha,\alpha' \mapsto S.\, C_2) \equiv \nu\alpha,\alpha' \mapsto S.\, (C_1 \parallel C_2)} \qquad \textsc{CC-Scope-Access}\quad \frac{x \text{ not free in } C_1}{C_1 \parallel (\nu x\colon [S].\, C_2) \equiv \nu x\colon [S].\, (C_1 \parallel C_2)}$$

Fig. 14. Configuration congruence ($C \equiv C$)

$$\textsc{CR-Fork}\quad \mathcal{C}[\mathcal{E}[\mathsf{fork}\, v]] \hookrightarrow_C \mathcal{C}[(v\ \mathsf{unit}) \parallel \mathcal{E}[\mathsf{unit}]] \qquad \textsc{CR-New}\quad \frac{x \text{ fresh}}{\mathcal{C}[\mathcal{E}[\mathsf{new}\, S]] \hookrightarrow_C \mathcal{C}[\nu x\colon [S].\, \mathcal{E}[x]]}$$

$$\textsc{CR-RequestAccept}\quad \frac{\alpha,\alpha' \text{ fresh} \qquad C \equiv \mathcal{C}[\nu x\colon [S].\, (\mathcal{E}_1[\mathsf{request}\, x] \parallel \mathcal{E}_2[\mathsf{accept}\, x] \parallel C')]}{C \hookrightarrow_C \mathcal{C}[\nu x\colon [S].\, \nu\alpha,\alpha' \mapsto S.\, (\mathcal{E}_1[\mathsf{chan}\, \alpha] \parallel \mathcal{E}_2[\mathsf{chan}\, \alpha'] \parallel C')]}$$

$$\textsc{CR-SendRecv}\quad \frac{C \equiv \mathcal{C}[\nu\alpha,\alpha' \mapsto !(\exists\alpha\colon \mathsf{Dom}(N).\Sigma;\, T).S.\, (\mathcal{E}_1[\mathsf{send}\, v\, \mathsf{on}\, \mathsf{chan}\, \alpha] \parallel \mathcal{E}_2[\mathsf{receive}\, \mathsf{chan}\, \alpha'] \parallel C')]}{C \hookrightarrow_C \mathcal{C}[\nu\alpha,\alpha' \mapsto S.\, (\mathcal{E}_1[\mathsf{unit}] \parallel \mathcal{E}_2[v] \parallel C')]}$$

$$\textsc{CR-SelectCase}\quad \frac{C \equiv \mathcal{C}[\nu\alpha,\alpha' \mapsto S_1 \oplus S_2.\, (\mathcal{E}_1[\mathsf{select}\, \ell\, \mathsf{on}\, \mathsf{chan}\, \alpha] \parallel \mathcal{E}_2[\mathsf{case}\, \mathsf{chan}\, \alpha'\, \mathsf{of}\{e_1;\, e_2\}] \parallel C')]}{C \hookrightarrow_C \mathcal{C}[\nu\alpha,\alpha' \mapsto S_\ell.\, (\mathcal{E}_1[\mathsf{unit}] \parallel \mathcal{E}_2[e_\ell] \parallel C')]}$$

$$\textsc{CR-Close}\quad \frac{C \equiv \mathcal{C}[\nu\alpha,\alpha' \mapsto \mathsf{End}.\, (\mathcal{E}_1[\mathsf{close}\, \mathsf{chan}\, \alpha] \parallel \mathcal{E}_2[\mathsf{close}\, \mathsf{chan}\, \alpha'] \parallel C')]}{C \hookrightarrow_C \mathcal{C}[\nu\alpha,\alpha' \mapsto \mathsf{End}.\, (\mathcal{E}_1[\mathsf{unit}] \parallel \mathcal{E}_2[\mathsf{unit}] \parallel C')]} \qquad \textsc{CR-Expr}\quad \frac{e_1 \hookrightarrow_e e_2}{\mathcal{C}[e_1] \hookrightarrow_C \mathcal{C}[e_2]}$$

Fig. 15. Configuration reduction ($C \hookrightarrow_C C$)

channel and scope abstractions) enables us to reorganize processes such that process reductions are simple to state. Channel abstraction may swap the channel names.

Figure 15 defines reduction for processes. Rules CR-Fork and CR-New apply to an expression process. The fork expression creates a new process that applies the fork's argument to unit while the old process continues with unit. The new expression creates a new access point and leaves its name in the evaluation context.

The remaining rules all concern rendezvous between two processes. Our rule have explicit assumptions that congruence rearranges processes as needed for the reductions to apply. All these rules involve binders and assume an additional process C' running in parallel with the processes participating in the redex, which keeps the processes with references to the binder.

Rule CR-RequestAccept creates a channel when there is a request and an accept on the same access point. The reduction creates the two ends of the new channel and passes them to the processes.

Rules CR-SendRecv and CR-SelectCase are standard. They could be blocked without the congruence rule CC-Swap in place.

Rule CR-Close is slightly unusual for readers familiar with linear session type calculi. The rule does not remove the closed channel from the configuration because the process under the binder may still contain (dead) references to the channel. This design makes reasoning about configurations in final state slightly more involved.

4 METATHEORY

We establish session fidelity and type soundness by applying the usual syntactic methods based on subject reduction and progress. Our subject reduction result for expressions applies in any context. As the type system of PolyVGR includes a conversion judgment, we can only prove subject reduction up to conversion. Subject reduction also holds for configurations.

All proofs along with additional lemmas etc may be found in the supplemental material.

Lemma 4.1 (Subject Reduction).

$$(1)\quad \frac{\vdash \Gamma_1 \qquad \Gamma_1 \vdash \Sigma_1 : \mathsf{State} \qquad \Gamma_1; \Sigma_1 \vdash e : \exists\Gamma_2.\Sigma_2; T \qquad e \hookrightarrow_e e'}{\exists T'.\ \Gamma_1; \Sigma_1 \vdash e' : \exists\Gamma_2.\Sigma_2; T' \wedge T' \equiv T}$$

$$(2)\quad \frac{\vdash \Gamma \qquad \Gamma \vdash \Sigma : \mathsf{State} \qquad \Gamma; \Sigma \vdash C \qquad C \hookrightarrow_C C'}{\exists \Sigma'.\ \Gamma; \Sigma' \vdash C'}$$

As configuration reduction is applied modulo the congruence relation, we also need to show that congruence preserves typing.

Lemma 4.2 (Subject Congruence).

$$\frac{\vdash \Gamma \qquad \Gamma \vdash \Sigma : \mathsf{State} \qquad \Gamma; \Sigma \vdash C \qquad C \equiv C'}{\Gamma; \Sigma \vdash C'}$$

It is always tricky to state a progress property in the context of processes, in particular when deadlocks may occur. Hence, we define several predicates on expressions to state progress concisely. The Value e predicate should be self explanatory. The Comm e predicate characterizes expressions that cannot reduce on the expression level, but require reduction at the level of configurations. Of those, the fork _ case is harmless, but the other cases require interaction with other processes to reduce.

Definition 4.3. The predicates Value e and Comm e are defined inductively.

- Value e if exists v such that $e = v$.
- Comm e if one of the following cases applies
  - $e = \mathsf{fork}\ \lambda(\Sigma;\ x\colon T).e_1$,
  - $e = \mathsf{new}\ S$,
  - $e = \mathsf{accept}\ v$,
  - $e = \mathsf{request}\ v$,
  - $e = \mathsf{send}\ v\ \mathsf{on\ chan}\ D$,
  - $e = \mathsf{receive\ chan}\ D$,
  - $e = \mathsf{select}\ \ell\ \mathsf{on\ chan}\ D$,
  - $e = \mathsf{case\ chan}\ D\ \mathsf{of}\{e_1;\ e_2\}$,
  - $e = \mathsf{close\ chan}\ D$, or

  – $e = \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2$ where $\mathrm{Comm}\, e_1$.

We also need a predicate that characterizes contexts built in a configuration. Besides type variables and constraints, they can only bind access points.

Definition 4.4. The predicate $\mathrm{Outer}\,\Gamma$ is defined by

- $\mathrm{Outer}\,\cdot$,
- $\mathrm{Outer}\,(\Gamma, \alpha : K)$ if $\mathrm{Outer}\,\Gamma$,
- $\mathrm{Outer}\,(\Gamma, x : T)$ if $\mathrm{Outer}\,\Gamma$ and $T = [S]$, and
- $\mathrm{Outer}\,(\Gamma, D_1 \,,_{\#}\, D_2)$ if $\mathrm{Outer}\,\Gamma$.

We are now ready to state progress for expressions. A typed expression is either a value, stuck on a communication (or fork), or it reduces.

Lemma 4.5 (Progress for expressions).

$$\frac{\vdash \Gamma \qquad \mathrm{Outer}\,\Gamma \qquad \Gamma \vdash \Sigma : \mathsf{State} \qquad \Gamma; \Sigma \vdash e : \exists \Gamma'.\Sigma'; T'}{\mathrm{Value}\, e \lor \mathrm{Comm}\, e \lor \exists e'.\, e \hookrightarrow_e e'}$$

We also need to characterize configurations. A final configuration cannot reduce in a good way: All processes are reduced to values, all protocols on channels have concluded as indicated by their session type End, and there may be access points.

Definition 4.6. The predicate $\mathrm{Final}\, C$ is defined inductively by the following cases:

- $\mathrm{Final}\, v$ (an expression process reduced to a value),
- $\mathrm{Final}\,(C_1 \parallel C_2)$ if $\mathrm{Final}\, C_1$ and $\mathrm{Final}\, C_2$,
- $\mathrm{Final}\,(\nu x\colon [S].\, C_1)$ if $\mathrm{Final}\, C_1$, or
- $\mathrm{Final}\,(\nu \alpha, \alpha' \mapsto \mathsf{End}.\, C_1)$ if $\mathrm{Final}\, C_1$.

The other possibility is that a configuration is deadlocked. The following definition lists all the ways in which reduction of a configuration may be disabled.

Definition 4.7. The predicate $\mathrm{Deadlock}\, C$ holds for a configuration C iff:

(1) For all configuration contexts $\mathcal{C}$, if $C = \mathcal{C}[e]$, then either $\mathrm{Value}\, e$ or $\mathrm{Comm}\, e$ and $e \neq \mathsf{fork}\, v$ and $e \neq \mathsf{new}\, S$.
(2) For all configuration contexts $\mathcal{C}$, if $C = \mathcal{C}[\nu x\colon [S].\, C']$, then
   - if $C' = \mathcal{C}_1[\mathcal{E}_1[\mathsf{request}\, x]]$, then there is no $\mathcal{C}_2, \mathcal{E}_2$ such that $C' = \mathcal{C}_2[\mathcal{E}_2[\mathsf{accept}\, x]]$,
   - if $C' = \mathcal{C}_1[\mathcal{E}_1[\mathsf{accept}\, x]]$, then there is no $\mathcal{C}_2, \mathcal{E}_2$ such that $C' = \mathcal{C}_2[\mathcal{E}_2[\mathsf{request}\, x]]$.
(3) For all configuration contexts $\mathcal{C}$, if $C = \mathcal{C}[\nu \alpha_1, \alpha_2 \mapsto S.\, C']$, then
   - if $C' = \mathcal{C}_1[\mathcal{E}_1[\mathsf{send}\, v\, \mathsf{on}\, \mathsf{chan}\, \alpha_\ell]]$,
     then there is no $\mathcal{C}_2, \mathcal{E}_2$ such that $C' = \mathcal{C}_2[\mathcal{E}_2[\mathsf{receive}\, \mathsf{chan}\, \alpha_{3-\ell}]]$,
   - if $C' = \mathcal{C}_1[\mathcal{E}_1[\mathsf{receive}\, \mathsf{chan}\, \alpha_\ell]]$,
     then there is no $\mathcal{C}_2, \mathcal{E}_2$ such that $C' = \mathcal{C}_2[\mathcal{E}_2[\mathsf{send}\, v\, \mathsf{on}\, \mathsf{chan}\, \alpha_{3-\ell}]]$,
   - if $C' = \mathcal{C}_1[\mathcal{E}_1[\mathsf{select}\, \ell'\, \mathsf{on}\, \mathsf{chan}\, \alpha_\ell]]$,
     then there is no $\mathcal{C}_2, \mathcal{E}_2$ such that $C' = \mathcal{C}_2[\mathcal{E}_2[\mathsf{case}\, \mathsf{chan}\, \alpha_{3-\ell}\, \mathsf{of}\{e_1;\, e_2\}]]$,
   - if $C' = \mathcal{C}_1[\mathcal{E}_1[\mathsf{case}\, \mathsf{chan}\, \alpha_\ell\, \mathsf{of}\{e_1;\, e_2\}]]$,
     then there is no $\mathcal{C}_2, \mathcal{E}_2$ such that $C' = \mathcal{C}_2[\mathcal{E}_2[\mathsf{select}\, \ell'\, \mathsf{on}\, \mathsf{chan}\, \alpha_{3-\ell}]]$,
   - if $C' = \mathcal{C}_1[\mathcal{E}_1[\mathsf{close}\, \mathsf{chan}\, \alpha_\ell]]$,
     then there is no $\mathcal{C}_2, \mathcal{E}_2$ such that $C' = \mathcal{C}_2[\mathcal{E}_2[\mathsf{close}\, \mathsf{chan}\, \alpha_{3-\ell}]]$.

Lemma 4.8 (Progress for configurations).

$$\frac{\vdash \Gamma \qquad \mathrm{Outer}\,\Gamma \qquad \Gamma \vdash \Sigma : \mathsf{State} \qquad \Gamma; \Sigma \vdash C}{\mathrm{Final}\, C \lor \mathrm{Deadlock}\, C \lor \exists C'.\, C \hookrightarrow_C C'}$$

5 RELATED WORK

We do not attempt to survey the vast amount of work in the session type community, but refer the reader to recent survey papers and books [Ancona et al. 2016; Bartoletti et al. 2015; Gay and Ravara 2017; Hüttel et al. 2016]. Instead we comment on the use of polymorphism in session types, the modeling of disjointness in the context of polymorphism, and potential connections to other work.

5.1 Polymorphism and Session Types

Polymorphism for session types was ignored for quite a while, although there are low-hanging fruit like parameterizing over the continuation session. The story starts with an investigation of bounded polymorphism over the type of transmitted values to avoid problems with subtyping in a π-calculus setting [Gay 2008].

Wadler [2014] includes polymorphism on session types where the quantifiers $\forall$ and $\exists$ are interpreted as sending and receiving types, similar to Turner's polymorphic π-calculus [Turner 1996]. Caires et al. [2013]; Pérez et al. [2014] consider impredicative quantifiers with session types using the same interpretation.

Dardha et al. [2017] extend an encoding of session types into π-types with parametric and bounded polymorphism. Lindley and Morris [2017] rely on row polymorphism to abstract over the irrelevant labels in a choice, thereby eliding the need for supporting subtyping. Their calculus FST (lightweight functional session types) supports polymorphism over kinded type variables $\alpha :: K(Y, Z)$ where $K = \mathit{Type}$, $Y = \circ$, and $Z = \pi$ indicates a variable ranging over session types; choosing $K = \mathit{Row}$ yields a row variable.

Almeida et al. [2021] consider impredicative polymorphism in the context of context-free session types. Their main contribution is the integration of algorithmic type checking for context-free sessions with polymorphism.

A common theme of the practically oriented works [Almeida et al. 2021; Lindley and Morris 2017] is their elaborate kind system that distinguishes linear from non-linear values, session types from non-session types, and rows from types (in the case of FST). PolyVGR follows suit in that its kinds distinguish session types and non-session types. Linearity is elided, but kinds for states, shapes, and domains are needed to handle channels. As a major novelty, PolyVGR includes arrow kinds and type-level lambda abstraction, but restricted such that abstraction ranges solely over domains.

5.2 Polymorphism and Disjointness

Alias types [Smith et al. 2000] is a type system for a low-level language where the type of a function expresses the shape of the store on which the function operates. For generality, function types can abstract over store locations α and the shape of the store is described by *aliasing constraints* of the form $\{\alpha \mapsto T\}$. Constraint composition resembles separating conjunction [Reynolds 2002] and ensures that locations are unique. Analogous to the channel types in our system, pointers in the alias types system have a singleton type that indicates their (abstract) store location and they can be duplicated. Alias types also include non-linear constraints, which are not required in our system. Alias types do not provide the means to abstract over groups of store locations as is possible with our domain/shape approach. It would be interesting to investigate such an extension to alias types.

Low-level liquid types [Rondon et al. 2010] use a similar setup as alias types with location-sensitive types to track pointers and pointer arithmetic as well as to enable strong updates in a verification setting for a C-like language. They also provide a mechanism of unfolding and folding to temporarily strengthen pointers so that they can be strongly updated. Such a mechanism is not needed for our calculus as channel resources are never aliased.

Disjoint intersection types [d. S. Oliveira et al. 2016] have been conceived to address the coherence problem of intersection type systems with an explicit merge operator: if the two "components" of the merge have the same type, then it is not clear which value should be chosen by the semantics. They rule out this scenario by requiring different types for all components of an intersection. Disjoint polymorphism [Alpuim et al. 2017] lifts this idea to a polymorphic calculus where type variables are introduced with disjointness constraints that rule out overlapping instantiations. Xie et al. [2020] show that calculi for disjoint polymorphic intersection types are closely related to polymorphic record calculi with symmetric concatenation [Harper and Pierce 1991].

Disjointness constraints for record types are related to our setting, but the labels in the records types are fixed and two records are still deemed disjoint if they share labels, as long as the corresponding field types are disjoint. In contrast, we have universal and existential quantification over domains (generalizing channel names) and single-channel domains disjoint by our axiomatic construction when composing states.

Morris and McKinna [2019] propose a generic system Rose for typing features based on row types. Its basis is a partial monoid of rows, which is chosen according to the application. Using rows for record types, Rose can be instantiated to support symmetric concatenation of records, shadowing concatenation, or even to allow several occurrences of the same label. While channel names are loosely related to record label and states might be represented as records, our axiomatic approach to maintaining disjointness is significantly different from their Rose system.

5.3 Diverse Topics

Pucella and Tov [2008] give an embedding of a session type calculus in Haskell. Their embedding is based on Atkey's parameterized monads [Atkey 2009], layered on top of the IO monad using phantom types. Their phantom type structure resembles our states where de Bruijn indices serve as channels names. Linear handling of the state is enforced by the monad abstraction, while channel references can be handled freely. The paper comes with a formalization and a soundness proof of the implementation. Sackman and Eisenbach [2008] also encode session types for a single channel in Haskell using an indexed (parameterized) monad.

A similar idea is the basis for work by Saffrich and Thiemann [2021]. They also start from VGR, point out some of its restrictions, but then continue to define a translation into a linear parameterized monad, which can be implemented in an existing monomorphic functional session type calculus [Gay and Vasconcelos 2010], extended with some syntactic sugar in the form of linear records. They prove that there are semantics- and typing-preserving translations forth and back, provided the typing of the functional calculus is severely restricted. Our work removes most of the restrictions of VGR's type system by using higher-order polymorphism. It remains to complete the diagram and identify a polymorphic functional session type calculus (most likely FST) which is suitable as a translation target.

Hinrichsen et al. [2021] describe semantic session typing as an alternative way to establish sound session type regimes. Instead of delving into syntactic type soundness proofs, they suggest to define a semantic notion of types and typing on top of an untyped semantics. Their proposal is based on (step-indexed) logical relations defined in terms of a suitable program logic [Dreyer et al. 2011] and it is fully mechanized in Coq. Starting from a simple session type system, they add polymorphism, subtyping, recursion, and more. It seems plausible that their model would scale to provide mechanized soundness proofs for PolyVGR.

6 CONCLUSION

We started this work on two premises:

- We believe it is important to map this so far unexplored part of the design space of session type systems.
- We believe that there is some potential in being able to write programs with session types in direct style as in Listing 3.

Looking back, we find that the direct style is scalable, it should be on the map as it more easily integrates with imperative programming styles and languages, and PolyVGR explains in depth the type system ingredients needed to decently program with session types in direct style. On the other hand, the amount of parameterization required in PolyVGR is significant and probably burdensome for programmers. Presently, we do not have practical experience with the language and we have not yet investigated some practical properties relating to the type system: Is type checking decidable? (We believe it is.) Is type inference possible? If so, the system may end up being practically useful, even if inferred types are complicated.

There is ample opportunity for future work. We want to investigate decidability of type checking as well as ways to perform type inference to obtain an implementation. We expect that a type preserving mapping from PolyVGR to FST [Lindley and Morris 2017] can be established analogous to the one outlined by Saffrich and Thiemann [2021]. There are promising ideas to integrate algebraic datatypes into PolyVGR.

REFERENCES

Bernardo Almeida, Andreia Mordido, Peter Thiemann, and Vasco T. Vasconcelos. 2021. Polymorphic Context-free Session Types. *CoRR* abs/2106.06658 (2021). arXiv:2106.06658 https://arxiv.org/abs/2106.06658

João Alpuim, Bruno C. d. S. Oliveira, and Zhiyuan Shi. 2017. Disjoint Polymorphism. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10201)*, Hongseok Yang (Ed.). Springer, 1–28. https://doi.org/10.1007/978-3-662-54434-1_1

Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. 2016. Behavioral Types in Programming Languages. *Found. Trends Program. Lang.* 3, 2-3 (2016), 95–230. https://doi.org/10.1561/2500000031

Robert Atkey. 2009. Parameterised Notions of Computation. *J. Funct. Program.* 19, 3-4 (2009), 335–376. https://doi.org/10.1017/S095679680900728X

Massimo Bartoletti, Ilaria Castellani, Pierre-Malo Deniélou, Mariangiola Dezani-Ciancaglini, Silvia Ghilezan, Jovanka Pantovic, Jorge A. Pérez, Peter Thiemann, Bernardo Toninho, and Hugo Torres Vieira. 2015. Combining behavioural types with security analysis. *J. Log. Algebraic Methods Program.* 84, 6 (2015), 763–780. https://doi.org/10.1016/j.jlamp.2015.09.003

Luís Caires, Jorge A. Pérez, Frank Pfenning, and Bernardo Toninho. 2013. Behavioral Polymorphism and Parametricity in Session-Based Communication. In *ESOP (LNCS, Vol. 7792)*, Matthias Felleisen and Philippa Gardner (Eds.). Springer, 330–349. https://doi.org/10.1007/978-3-642-37036-6_19

Bruno C. d. S. Oliveira, Zhiyuan Shi, and João Alpuim. 2016. Disjoint Intersection Types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM, 364–377. https://doi.org/10.1145/2951913.2951945

Ornela Dardha, Elena Giachino, and Davide Sangiorgi. 2017. Session Types Revisited. *IC* 256 (2017), 253–286. https://doi.org/10.1016/j.ic.2017.06.002

Derek Dreyer, Amal Ahmed, and Lars Birkedal. 2011. Logical Step-Indexed Logical Relations. *Log. Methods Comput. Sci.* 7, 2 (2011). https://doi.org/10.2168/LMCS-7(2:16)2011

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, Robert Cartwright (Ed.). ACM, 237–247. https://doi.org/10.1145/155090.155113

Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. 2019. Exceptional Asynchronous Session Types: Session Types Without Tiers. *Proc. ACM Program. Lang.* 3, POPL (2019), 28:1–28:29. https://doi.org/10.1145/3290341

Simon Gay and António Ravara (Eds.). 2017. *Behavioural Types: from Theory to Tools.* River Publishers. https://doi.org/10.13052/rp-9788793519817

Simon J. Gay. 2008. Bounded Polymorphism in Session Types. *Math. Struct. Comput. Sci.* 18, 5 (2008), 895–930. https://doi.org/10.1017/S0960129508006944

Simon J. Gay and Vasco Thudichum Vasconcelos. 2010. Linear Type Theory for Asynchronous Session Types. *J. Funct. Program.* 20, 1 (2010), 19–50. https://doi.org/10.1017/S0956796809990268

Robert Harper and Benjamin C. Pierce. 1991. A Record Calculus Based on Symmetric Concatenation. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida, USA, January 21-23, 1991*, David S. Wise (Ed.). ACM Press, 131–142. https://doi.org/10.1145/99583.99603

Jonas Kastberg Hinrichsen, Daniël Louwrink, Robbert Krebbers, and Jesper Bengtson. 2021. Machine-checked semantic session typing. In *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021*, Catalin Hritcu and Andrei Popescu (Eds.). ACM, 178–198. https://doi.org/10.1145/3437992.3439914

Kohei Honda. 1993. Types for Dyadic Interaction. In *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings (Lecture Notes in Computer Science, Vol. 715)*, Eike Best (Ed.). Springer, 509–523. https://doi.org/10.1007/3-540-57208-2_35

Raymond Hu, Dimitrios Kouzapas, Olivier Pernet, Nobuko Yoshida, and Kohei Honda. 2010. Type-Safe Eventful Sessions in Java. In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6183)*, Theo D'Hondt (Ed.). Springer, 329–353. https://doi.org/10.1007/978-3-642-14107-2_16

Raymond Hu and Nobuko Yoshida. 2016. Hybrid Session Verification Through Endpoint API Generation. In *Fundamental Approaches to Software Engineering - 19th International Conference, FASE 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9633)*, Perdita Stevens and Andrzej Wasowski (Eds.). Springer, 401–418.

https://doi.org/10.1007/978-3-662-49665-7_24

Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. 2016. Foundations of Session Types and Behavioural Contracts. *ACM Comput. Surv.* 49, 1 (2016), 3:1–3:36. https://doi.org/10.1145/2873052

Sam Lindley and J. Garrett Morris. 2017. Lightweight Functional Session Types. In *Behavioural Types: from Theory to Tools*, Simon Gay and António Ravara (Eds.). River Publishers. Extended version at https://homepages.inf.ed.ac.uk/slindley/papers/fst-extended.pdf.

J. Garrett Morris and James McKinna. 2019. Abstracting Extensible Data Types: or, Rows by Any Other Name. *Proc. ACM Program. Lang.* 3, POPL (2019), 12:1–12:28. https://doi.org/10.1145/3290325

Luca Padovani. 2017. A Simple Library Implementation of Binary Sessions. *J. Funct. Program.* 27 (2017), e4. https://doi.org/10.1017/S0956796816000289

Jorge A. Pérez, Luís Caires, Frank Pfenning, and Bernardo Toninho. 2014. Linear logical relations and observational equivalences for session-based concurrency. *IC* 239 (2014), 254–302. https://doi.org/10.1016/j.ic.2014.08.001

Riccardo Pucella and Jesse A. Tov. 2008. Haskell Session Types With (Almost) no Class. In *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*, Andy Gill (Ed.). ACM, 25–36. https://doi.org/10.1145/1411286.1411290

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. IEEE Computer Society, 55–74. https://doi.org/10.1109/LICS.2002.1029817

Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2010. Low-Level Liquid Types. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, Manuel V. Hermenegildo and Jens Palsberg (Eds.). ACM, 131–144. https://doi.org/10.1145/1706299.1706316

Matthew Sackman and Susan Eisenbach. 2008. Session Types in Haskell Updating Message Passing for the 21st Century. (2008). https://spiral.imperial.ac.uk:8443/handle/10044/1/5918.

Hannes Saffrich and Peter Thiemann. 2021. Relating Functional and Imperative Session Types. In *Coordination Models and Languages - 23rd IFIP WG 6.1 International Conference, COORDINATION 2021 (Lecture Notes in Computer Science, Vol. 12717)*, Ferruccio Damiani and Ornela Dardha (Eds.). Springer, 61–79. https://doi.org/10.1007/978-3-030-78142-2_4

Alceste Scalas and Nobuko Yoshida. 2016. Lightweight Session Programming in Scala. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy (LIPIcs, Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 21:1–21:28. https://doi.org/10.4230/LIPIcs.ECOOP.2016.21

Frederick Smith, David Walker, and J. Gregory Morrisett. 2000. Alias Types. In *Programming Languages and Systems, 9th European Symposium on Programming, ESOP 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000, Proceedings (Lecture Notes in Computer Science, Vol. 1782)*, Gert Smolka (Ed.). Springer, 366–381. https://doi.org/10.1007/3-540-46425-5_24

Robert E. Strom and Shaula Yemini. 1986. Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Trans. Software Eng.* 12, 1 (1986), 157–171. https://doi.org/10.1109/TSE.1986.6312929

Kaku Takeuchi, Kohei Honda, and Makoto Kubo. 1994. An Interaction-based Language and its Typing System. In *PARLE '94: Parallel Architectures and Languages Europe, 6th International PARLE Conference, Athens, Greece, July 4-8, 1994, Proceedings (Lecture Notes in Computer Science, Vol. 817)*, Constantine Halatsis, Dimitris G. Maritsas, George Philokyprou, and Sergios Theodoridis (Eds.). Springer, 398–413. https://doi.org/10.1007/3-540-58184-7_118

David N. Turner. 1996. *The polymorphic Pi-calculus : theory and implementation*. Ph. D. Dissertation. University of Edinburgh, UK. http://hdl.handle.net/1842/395

Vasco Thudichum Vasconcelos, Simon J. Gay, and António Ravara. 2006. Type Checking a Multithreaded Functional Language With Session Types. *Theor. Comput. Sci.* 368, 1-2 (2006), 64–87. https://doi.org/10.1016/j.tcs.2006.06.028

Philip Wadler. 2014. Propositions as Sessions. *JFP* 24, 2-3 (2014), 384–418. http://dx.doi.org/10.1017/S095679681400001X

Ningning Xie, Bruno C. d. S. Oliveira, Xuan Bi, and Tom Schrijvers. 2020. Row and Bounded Polymorphism via Disjoint Polymorphism. In *34th European Conference on Object-Oriented Programming, ECOOP 2020, July, 2020, Potsdam, Germany (LIPIcs, Vol. 109)*, Robert Hirschfeld (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 27:1–27:29. https://doi.org/10.4230/LIPIcs.ECOOP.2020.27

A APPENDIX

A.1 Metatheory

LEMMA A.1 (CONTEXT RESTRICTION PRESERVES KIND FORMATION).

$$(1)\ \frac{\Gamma \vdash T : \mathsf{Shape}}{\lfloor\Gamma\rfloor \vdash T : \mathsf{Shape}} \qquad (2)\ \frac{\Gamma \vdash K}{\lfloor\Gamma\rfloor \vdash K} \qquad (3)\ \frac{\vdash \Gamma}{\vdash \lfloor\Gamma\rfloor}$$

PROOF.

(1) Straightforward induction over the derivations with kind Shape. The case of type application is not possible, since type lambdas cannot have codomain Shape.
(2) Straightforward induction over the kind formation using (1) in the case KF-DOM.
(3) Straightforward induction over $\vdash \Gamma$. Since $\lfloor\Gamma\rfloor$ removes all value-level and constraint bindings, only the well-kindedness of bound type-variables needs to be preserved, which follows via (2). □

Definition A.2 (Order Preserving Embedding (OPE)). Γ_2 is an *Order Preserving Embedding* of Γ_1, written as $\Gamma_1 \overset{\text{OPE}}{\Rightarrow} \Gamma_2$, iff

(1) $\vdash \Gamma_1$
(2) $\vdash \Gamma_2$
(3) $\forall b \in \Gamma_1 . b \in \Gamma_2$

LEMMA A.3 (CONTEXT RESTRICTION PRESERVES OPE).

$$\frac{\Gamma_1 \overset{\text{OPE}}{\Rightarrow} \Gamma_2}{\lfloor\Gamma_1\rfloor \overset{\text{OPE}}{\Rightarrow} \lfloor\Gamma_2\rfloor}$$

PROOF. Axioms (1) and (2) follow via Lemma A.1.3; axiom (3) holds, because if $\lfloor\cdot\rfloor$ removes a binding from Γ_2, then that binding is either not present in Γ_1 or also removed in $\lfloor\Gamma_1\rfloor$. □

LEMMA A.4 (CONTEXT EXTENSION PRESERVES OPE).

$$(1)\ \frac{\Gamma_1 \overset{\text{OPE}}{\Rightarrow} \Gamma_2 \qquad \vdash \Gamma_1, \Gamma_3 \qquad \vdash \Gamma_2, \Gamma_3}{(\Gamma_1, \Gamma_3) \overset{\text{OPE}}{\Rightarrow} (\Gamma_2, \Gamma_3)} \qquad (2)\ \frac{\Gamma_1 \overset{\text{OPE}}{\Rightarrow} \Gamma_2 \qquad \vdash \Gamma_1 ,_{\#} \Gamma_3 \qquad \vdash \Gamma_2 ,_{\#} \Gamma_3}{(\Gamma_1 ,_{\#} \Gamma_3) \overset{\text{OPE}}{\Rightarrow} (\Gamma_2 ,_{\#} \Gamma_3)}$$

PROOF.

(1) Same as (2) but without the additional complication of constraints.
(2) OPE Axiom (1) and (2) follow by assumption. For Axiom (3) we need to show that

$$\forall b \in \Gamma_1, \Gamma_3, \mathbb{C}_{13}, \mathbb{C}_3.\ b \in \Gamma_2, \Gamma_3, \mathbb{C}_{23}, \mathbb{C}_3$$

where the $\mathbb{C}_i$ are defined as in Figure 9. Any binding from Γ_1 is also contained in Γ_2, due to Axiom (3) of $\Gamma_1 \overset{\text{OPE}}{\Rightarrow} \Gamma_2$. Hence, it also holds that $\mathrm{dom}(\Gamma_1) \subseteq \mathrm{dom}(\Gamma_2)$, which implies $\mathbb{C}_{13} \subseteq \mathbb{C}_{23}$. □

Lemma A.5 (Context Formation and Concatenation). $\vdash \Gamma_1, \Gamma_2 \iff \vdash \Gamma_1 \,,_{\#}\, \Gamma_2$

Proof.

$\Leftarrow$: Straightforward, because $\Gamma_1 \,,_{\#}\, \Gamma_2 = \Gamma_1, \Gamma_2, \mathbb{C}_{12}, \mathbb{C}_2$, so we can just split off the constraints from the context formation by repeated case analysis.

$\Rightarrow$: To append the constraints $\mathbb{C}_{12}$ and $\mathbb{C}_2$ to the context formation, we need to repeatedly apply CF-ConsCstr, which requires all constraint axioms to use well-kinded domains. By definition of $\mathbb{C}_{12}$ and $\mathbb{C}_2$, all domains are type variables from $\mathrm{dom}(\Gamma_1) \cup \mathrm{dom}(\Gamma_2)$, and are hence well-kinded. □

Lemma A.6 (Weakening). *If* $\Gamma_1 \overset{\mathrm{OPE}}{\Rightarrow} \Gamma_2$ *then*

$$(1)\ \frac{\Gamma_1 \vdash \mathbb{C}}{\Gamma_2 \vdash \mathbb{C}} \qquad (2)\ \frac{\vdash \Gamma_1, \Gamma_3}{\vdash \Gamma_2, \Gamma_3} \qquad (3)\ \frac{\vdash \Gamma_1 \,,_{\#}\, \Gamma_3}{\vdash \Gamma_2 \,,_{\#}\, \Gamma_3} \qquad (4)\ \frac{\Gamma_1 \vdash K}{\Gamma_2 \vdash K} \qquad (5)\ \frac{\Gamma_1 \vdash T : K}{\Gamma_2 \vdash T : K}$$

$$(6)\ \frac{\Gamma_1 \vdash v : T}{\Gamma_2 \vdash v : T} \qquad (7)\ \frac{\Gamma_1; \Sigma_1 \vdash e : \exists \Gamma_3.\Sigma_2; T_2}{\Gamma_2; \Sigma_1 \vdash e : \exists \Gamma_3.\Sigma_2; T_2} \qquad (8)\ \frac{\Gamma_1; \Sigma \vdash C}{\Gamma_2; \Sigma \vdash C}$$

Proof. By mutual induction over the derivation to be weakened:

(1) • *Case* CE-Axiom. Direct consequence of Axiom (3) of $\Gamma_1 \overset{\mathrm{OPE}}{\Rightarrow} \Gamma_2$.
 • All other cases are immediate by the induction hypotheses.

(2) By induction over Γ_3:
 • *Case* $\Gamma_3 = \cdot$. Immediate from Axiom (2) of $\Gamma_1 \overset{\mathrm{OPE}}{\Rightarrow} \Gamma_2$.
 • *Case* $\Gamma_3 = \Gamma_3', \alpha : K$. In this case we have

$$\frac{\vdash \Gamma_1, \Gamma_3' \qquad \Gamma_1, \Gamma_3' \vdash \alpha : K}{\vdash \Gamma_1, \Gamma_3', \alpha : K}\ \text{CF-ConsKind}$$

and need to show

$$\frac{\text{(A)}\ \dfrac{}{\vdash \Gamma_2, \Gamma_3'} \qquad \dfrac{}{\Gamma_2, \Gamma_3' \vdash \alpha : K}\ \text{(B)}}{\vdash \Gamma_2, \Gamma_3', \alpha : K}\ \text{CF-ConsKind}$$

(A) follows from the inner induction hypothesis; (B) follows from the outer induction hypothesis for (5).
 • The cases for value-level and constraint bindings are analogous to the previous case.

(3) Follows by applying Lemma A.5 to both the premise and conclusion of (2).

(4) All cases are immediate by the induction hypotheses.

(5) • *Case* K-Var. Follows directly from Axiom (3) of $\Gamma_1 \overset{\mathrm{OPE}}{\Rightarrow} \Gamma_2$.
 • *Case* K-Lam, K-Send, K-Recv. Here we have assumptions using context restriction, like $\lfloor \Gamma_1 \rfloor, \alpha : \mathsf{Dom}(N) \vdash \Sigma : \mathsf{State}$. In order to apply the induction hypothesis on those assumptions, we rely on Lemma A.3, which given $\Gamma_1 \overset{\mathrm{OPE}}{\Rightarrow} \Gamma_2$ yields $\lfloor \Gamma_1 \rfloor \overset{\mathrm{OPE}}{\Rightarrow} \lfloor \Gamma_2 \rfloor$.
 • *Case* K-Arr. Here we have assumptions using disjoint context concatenation, like $\Gamma_1 \,,_{\#}\, \Gamma_2' \vdash \Sigma_2 : \mathsf{State}$. In order to apply the induction hypothesis on those assumptions, we rely on Lemma A.4.(2), which given $\Gamma_1 \overset{\mathrm{OPE}}{\Rightarrow} \Gamma_2$ yields $(\Gamma_1 \,,_{\#}\, \Gamma_2') \overset{\mathrm{OPE}}{\Rightarrow} (\Gamma_2 \,,_{\#}\, \Gamma_2')$.
 • All other cases are immediate by the induction hypotheses. Going under binders requires the Barendregt Convention and Lemma A.4.(1) to extend the OPE.

(6) • *Case* T-Var. Same as K-Var.

- All other cases are immediate by the induction hypotheses. Going under binders requires the Barendregt Convention and Lemma A.4.(1) to extend the OPE.

(7)
- *Case* T-Let. Same as K-Arr.
- All other cases are immediate by the induction hypotheses.

(8)
- *Case* T-NuChan. Same as K-Arr.
- All other cases are immediate by the induction hypotheses. Going under binders requires the Barendregt Convention and Lemma A.4.(1) to extend the OPE. □

Definition A.7 (Substitution Typing). We write $\vdash \sigma : \Gamma_1 \Rightarrow \Gamma_2$ iff

(1) $\vdash \Gamma_1$
(2) $\vdash \Gamma_2$
(3) $\forall (x : T) \in \Gamma_1.\ \Gamma_2 \vdash \sigma x : \sigma T$,
(4) $\forall (\alpha : K) \in \Gamma_1.\ \Gamma_2 \vdash \sigma\alpha : \sigma K$, and
(5) $\forall (D_1 \# D_2) \in \Gamma_1.\ \Gamma_2 \vdash \sigma D_1 \# \sigma D_2$.

Lemma A.8 (Typing of the Identity Substitution). *If* $\vdash \Gamma$, *then* $\vdash id : \Gamma \Rightarrow \Gamma$.

Proof. (1) and (2) follow by assumption.
(3), (4), and (5) follow via T-Var, K-Var, and CE-Axiom, respectively. □

Lemma A.9 (Extending Substitution Typings).

(1) $$\frac{\vdash \sigma : \Gamma_1 \Rightarrow \Gamma_2 \qquad \Gamma_1 \vdash T : \mathsf{Type} \qquad \Gamma_2 \vdash v : \sigma T \qquad x \notin \mathrm{dom}(\Gamma_1)}{\vdash (\sigma, x \mapsto v) : (\Gamma_1, x : T) \Rightarrow \Gamma_2}$$

(2) $$\frac{\vdash \sigma : \Gamma_1 \Rightarrow \Gamma_2 \qquad \Gamma_1 \vdash K \qquad \Gamma_2 \vdash T : \sigma K \qquad \alpha \notin \mathrm{dom}(\Gamma_1)}{\vdash (\sigma, \alpha \mapsto T) : (\Gamma_1, \alpha : K) \Rightarrow \Gamma_2}$$

(3) $$\frac{\vdash \sigma : \Gamma_1 \Rightarrow \Gamma_2 \qquad \vdash \Gamma_1, \mathbb{C} \qquad \Gamma_2 \vdash \sigma\mathbb{C}}{\vdash \sigma : (\Gamma_1, \mathbb{C}) \Rightarrow \Gamma_2}$$

Proof. Straightforward case analysis and rule applications. □

Lemma A.10 (Weakening Substitution Typings).

$$\frac{\vdash \sigma : \Gamma_1 \Rightarrow \Gamma_2 \qquad \vdash \Gamma_2, \Gamma_2'}{\vdash \sigma : \Gamma_1 \Rightarrow \Gamma_2, \Gamma_2'}$$

Proof. By Definition A.7, we have to prove:

(1) $\vdash \Gamma_1$, which follows by $\vdash \sigma : \Gamma_1 \Rightarrow \Gamma_2$.
(2) $\vdash \Gamma_2, \Gamma_2'$, which follows by assumption.
(3) $\forall (x : T) \in \Gamma_1.\ \Gamma_2, \Gamma_2' \vdash \sigma x : \sigma T$. Let $(x : T) \in \Gamma_1$, then by $\vdash \sigma : \Gamma_1 \Rightarrow \Gamma_2$ it follows that
$$\Gamma_2 \vdash \sigma x : \sigma T$$
which by weakening via Lemma A.6 yields
$$\Gamma_2, \Gamma_2' \vdash \sigma x : \sigma T$$
(4) $\forall (\alpha : K) \in \Gamma_1.\ \Gamma_2, \Gamma_2' \vdash \sigma\alpha : \sigma K$, which follows similarly by weakening.
(5) $\forall (D_1 \# D_2) \in \Gamma_1.\ \Gamma_2, \Gamma_2' \vdash \sigma D_1 \# \sigma D_2$, which follows similarly by weakening. □

Lemma A.11 (Lifting Substitution Typings).

(1) $$\dfrac{\vdash \sigma : \Gamma_1 \Rightarrow \Gamma_2 \qquad \Gamma_1 \vdash T : \mathsf{Type} \qquad \Gamma_2 \vdash \sigma T : \mathsf{Type} \qquad x \notin \mathrm{dom}(\Gamma_1) \qquad x \notin \mathrm{dom}(\Gamma_2)}{\vdash (\sigma, x \mapsto x) : (\Gamma_1, x : T) \Rightarrow (\Gamma_2, x : \sigma T)}$$

(2) $$\dfrac{\vdash \sigma : \Gamma_1 \Rightarrow \Gamma_2 \qquad \Gamma_1 \vdash K \qquad \Gamma_2 \vdash \sigma K \qquad \alpha \notin \mathrm{dom}(\Gamma_1) \qquad \alpha \notin \mathrm{dom}(\Gamma_2)}{\vdash (\sigma, \alpha \mapsto \alpha) : (\Gamma_1, \alpha : K) \Rightarrow (\Gamma_2, \alpha : \sigma K)}$$

(3) $$\dfrac{\vdash \sigma : \Gamma_1 \Rightarrow \Gamma_2 \qquad \vdash \Gamma_1, \mathbb{C} \qquad \vdash \Gamma_2, \sigma\mathbb{C}}{\vdash \sigma : (\Gamma_1, \mathbb{C}) \Rightarrow (\Gamma_2, \mathbb{C})}$$

(4) $$\dfrac{\vdash \sigma : \Gamma_1 \Rightarrow \Gamma_2 \qquad \vdash \Gamma_1, \Gamma \qquad \vdash \Gamma_2, \sigma\Gamma}{\vdash (\sigma, id) : (\Gamma_1, \Gamma) \Rightarrow (\Gamma_2, \sigma\Gamma)}$$

(5) $$\dfrac{\vdash \sigma : \Gamma_1 \Rightarrow \Gamma_2 \qquad \vdash \Gamma_1 ,_{\#} \Gamma \qquad \vdash \Gamma_2 ,_{\#} \sigma\Gamma}{\vdash (\sigma, id) : (\Gamma_1 ,_{\#} \Gamma) \Rightarrow (\Gamma_2 ,_{\#} \sigma\Gamma)}$$

Proof.

(1) First, we weaken the substitution typing

$$\dfrac{\vdash \sigma : \Gamma_1 \Rightarrow \Gamma_2 \qquad \dfrac{\vdash \Gamma_2 \qquad \Gamma_2 \vdash \sigma T : \mathsf{Type} \qquad x \notin \mathrm{dom}(\Gamma_2)}{\vdash \Gamma_2, x : \sigma T}\ \textsc{CF-ConsType}}{\vdash \sigma : \Gamma_1 \Rightarrow \Gamma_2, x : \sigma T}\ \textsc{Lemma A.10}$$

Then we extend the weakened substitution typing

$$\dfrac{\vdash \sigma : \Gamma_1 \Rightarrow \Gamma_2, x : \sigma T \qquad \Gamma_1 \vdash T : \mathsf{Type} \qquad \dfrac{}{\Gamma_2, x : \sigma T \vdash x : \sigma T}\ \textsc{T-Var} \qquad x \notin \mathrm{dom}(\Gamma_1)}{\vdash \sigma, x \mapsto x : \Gamma_1, x : T \Rightarrow \Gamma_2, x : \sigma T}\ \textsc{Lemma A.9}$$

(2) Same as (1).

(3) First, we weaken the substitution typing

$$\dfrac{\vdash \sigma : \Gamma_1 \Rightarrow \Gamma_2 \qquad \vdash \Gamma_2, \sigma\mathbb{C}}{\vdash \sigma : \Gamma_1 \Rightarrow \Gamma_2, \sigma\mathbb{C}}\ \textsc{Lemma A.10}$$

Then we extend the weakened substitution typing

$$\dfrac{\vdash \sigma : \Gamma_1 \Rightarrow \Gamma_2, \sigma\mathbb{C} \qquad \vdash \Gamma_1, \mathbb{C} \qquad \Gamma_2, \sigma\mathbb{C} \vdash \sigma\mathbb{C}}{\vdash \sigma, x \mapsto x : \Gamma_1, \mathbb{C} \Rightarrow \Gamma_2, \sigma\mathbb{C}}\ \textsc{Lemma A.9}$$

where $\Gamma_2, \sigma\mathbb{C} \vdash \sigma\mathbb{C}$ follows via repeated applications of CE-Split, CE-Axiom, and CE-Merge.

(4) Follows from (1) to (3) by induction on Γ.

(5) In this case we have

$$\Gamma_1 ,_{\#} \Gamma = \Gamma_1, \Gamma, \mathbb{C}_1, \mathbb{C}_1'$$
$$\Gamma_2 ,_{\#} \sigma\Gamma = \Gamma_2, \sigma\Gamma, \mathbb{C}_2, \mathbb{C}_2'$$

with

$$\begin{aligned}
\mathbb{C}_1 &= \{\alpha_1 \# \alpha_2 \mid \alpha_1, \alpha_2 \in \mathrm{dom}(\lceil \Gamma \rceil), \alpha_1 \neq \alpha_2\} \\
\mathbb{C}_2 &= \{\alpha_1 \# \alpha_2 \mid \alpha_1, \alpha_2 \in \mathrm{dom}(\lceil \sigma\Gamma \rceil), \alpha_1 \neq \alpha_2\} \\
\mathbb{C}_1' &= \{\alpha_1 \# \alpha_2 \mid \alpha_1 \in \mathrm{dom}(\lceil \Gamma_1 \rceil), \alpha_2 \in \mathrm{dom}(\lceil \Gamma \rceil)\} \\
\mathbb{C}_2' &= \{\alpha_1 \# \alpha_2 \mid \alpha_1 \in \mathrm{dom}(\lceil \Gamma_2 \rceil), \alpha_2 \in \mathrm{dom}(\lceil \sigma\Gamma \rceil)\}
\end{aligned}$$

Via (4) follows

$$\vdash \sigma : \Gamma_1, \Gamma \Rightarrow \Gamma_2, \sigma\Gamma$$

We have $\sigma\mathbb{C}_1 = \mathbb{C}_1$, because $\mathbb{C}_1$ contains by definition only variables from Γ, so σ behaves as the identity. Furthermore, we have $\mathbb{C}_1 = \mathbb{C}_2$, because $\mathrm{dom}(\lceil\sigma\Gamma\rceil) = \mathrm{dom}(\lceil\Gamma\rceil)$. Hence, we can apply (3) to the previous result and rewrite $\sigma\mathbb{C}_1$ to $\mathbb{C}_2$, which yields

$$\vdash \sigma : \Gamma_1, \Gamma, \mathbb{C}_1 \Rightarrow \Gamma_2, \sigma\Gamma, \mathbb{C}_2$$

To conclude with

$$\vdash \sigma : \Gamma_1, \Gamma, \mathbb{C}_1, \mathbb{C}_1' \Rightarrow \Gamma_2, \sigma\Gamma, \mathbb{C}_2, \mathbb{C}_2'$$

we need to show that for any $(D_1 \# D_2) \in \Gamma_1, \Gamma, \mathbb{C}_1, \mathbb{C}_1'$ it holds that

$$\Gamma_2, \sigma\Gamma, \mathbb{C}_2, \mathbb{C}_2' \vdash \sigma D_1 \# \sigma D_2$$

If the constraint axiom is in $\Gamma_1, \Gamma, \mathbb{C}_1$, then the result follows by the previous substitution typing and weakening. If the constraint axiom is in $\mathbb{C}_1'$, then σD_1 contains only variables from $\lceil\Gamma_2\rceil$ and σD_2 contains only variables from $\lceil\sigma\Gamma\rceil$, so $(\sigma D_1 \# \sigma D_2)$ is part of $\mathbb{C}_2'$ and can be proved via CE-Axiom. □

Lemma A.12 (Context Restriction preserves Substitution Typing).

$$\frac{\vdash \sigma : \Gamma_1 \Rightarrow \Gamma_2}{\vdash \sigma : \lfloor\Gamma_1\rfloor \Rightarrow \lfloor\Gamma_2\rfloor}$$

Proof.

- Axioms (1) and (2) follow via Lemma A.1.3.
- Axiom (3) and (5) hold trivially, since $\lfloor\cdot\rfloor$ removes all value-level and constraint bindings.
- For Axiom (4), let $(\alpha : K) \in \lfloor\Gamma_1\rfloor$. From $\vdash \sigma : \Gamma_1 \Rightarrow \Gamma_2$ we know $\Gamma_2 \vdash \sigma\alpha : \sigma K$, but we need to prove $\lfloor\Gamma_2\rfloor \vdash \sigma\alpha : \sigma K$. By definition of $\lfloor\cdot\rfloor$, we know that

$$K \in \{\mathsf{Shape}, \mathsf{Session}, \mathsf{Dom}(N) \to \mathsf{Type}, \mathsf{Dom}(N) \to \mathsf{State}\}.$$

Types of those kinds, like $\sigma\alpha$, have free type variables only at positions, which are themselves restricted with $\lfloor\cdot\rfloor$, so $\lfloor\Gamma_2\rfloor \vdash \sigma\alpha : \sigma K$ is a valid strenghtening of $\Gamma_2 \vdash \sigma\alpha : \sigma K$. □

Lemma A.13 (Substitution Preserves Derivations). *If* $\vdash \sigma : \Gamma_1 \Rightarrow \Gamma_2$*, then*

$$(1)\ \frac{\Gamma_1 \vdash \mathbb{C}}{\Gamma_2 \vdash \sigma\mathbb{C}} \quad (2)\ \frac{\vdash \Gamma_1, \Gamma_3}{\vdash \Gamma_2, \sigma\Gamma_3} \quad (3)\ \frac{\vdash \Gamma_1 ,_{\#} \Gamma_3}{\vdash \Gamma_2 ,_{\#} \sigma\Gamma_3} \quad (4)\ \frac{\Gamma_1 \vdash K}{\Gamma_2 \vdash \sigma K} \quad (5)\ \frac{\Gamma_1 \vdash T : K}{\Gamma_2 \vdash \sigma T : \sigma K} \quad (6)\ \frac{\Gamma_1 \vdash v : T}{\Gamma_2 \vdash \sigma v : \sigma T}$$

$$(7)\ \frac{\Gamma_1 \vdash \Sigma_1 : \mathsf{State} \qquad \Gamma_1; \Sigma_1 \vdash e : \exists\Gamma_3.\Sigma_2; T}{\Gamma_2; \sigma\Sigma_1 \vdash \sigma e : \exists\sigma\Gamma_3.\sigma\Sigma_2; \sigma T} \qquad (9)\ \frac{T_1 \equiv T_2}{\sigma T_1 \equiv \sigma T_2}$$

Proof. By mutual induction on the derivations subject to substitution:

(1) By induction on $\Gamma_1 \vdash \mathbb{C}$:
   - *Case* CE-Axiom. Here we have $\Gamma_1 = \Gamma_1', D_1 \# D_2$ and

   $$\Gamma_1', D_1 \# D_2 \vdash D_1 \# D_2$$

   We need to show

   $$\Gamma_2 \vdash \sigma D_1 \# \sigma D_2$$

   which follows immediately from Axiom (5) of the substitution typing:

   $$\forall (D_1 \# D_2) \in \Gamma_1.\ \Gamma_2 \vdash \sigma D_1 \# \sigma D_2$$

- *Case* CE-Sym, CE-Empty, CE-Split, CE-Merge, CE-Empty, CE-Cons. Immediate from the induction hypotheses.

(2) Analogous to the corresponding case in Lemma A.6 (Weakening).

(3) Follows by applying Lemma A.5 to both the premise and conclusion of (2).

(4)
- *Case* KF-Type, KF-Session, KF-State, KF-Shape. Trivial.
- *Case* KF-Dom, KF-Arr. Immediate from the induction hypotheses.

(5)
- *Case* K-Var. Immediate from Axiom (4) of the substitution typing.
- *Case* K-App. Immediate from the induction hypotheses.
- *Case* K-Lam. We first apply the induction hypothesis to the first subderivation

$$\Gamma_1 \vdash N : \mathsf{Shape}$$

which yields

$$\Gamma_2 \vdash \sigma N : \mathsf{Shape}.$$

To be able to apply the induction hypothesis to the second subderivation, we first lift the substitution typing and kindings over the context restriction

$$\dfrac{\vdash \sigma : \Gamma_1 \Rrightarrow \Gamma_2}{\vdash \sigma : \lfloor\Gamma_1\rfloor \Rrightarrow \lfloor\Gamma_2\rfloor}\ \text{Lemma A.12} \qquad \dfrac{\dfrac{\Gamma_1 \vdash N : \mathsf{Shape}}{\Gamma_1 \vdash \mathsf{Dom}(N)}\ \text{KF-Dom}}{\lfloor\Gamma_1\rfloor \vdash \mathsf{Dom}(N)}\ \text{Lemma A.1} \qquad \dfrac{\dfrac{\Gamma_2 \vdash \sigma N : \mathsf{Shape}}{\Gamma_2 \vdash \mathsf{Dom}(\sigma N)}\ \text{KF-Dom}}{\lfloor\Gamma_2\rfloor \vdash \mathsf{Dom}(\sigma N)}\ \text{Lemma A.1}$$

and then lift the substitution typing over the new domain binding

$$\dfrac{\vdash \sigma : \lfloor\Gamma_1\rfloor \Rrightarrow \lfloor\Gamma_2\rfloor \qquad \lfloor\Gamma_1\rfloor \vdash \mathsf{Dom}(N) \qquad \lfloor\Gamma_2\rfloor \vdash \mathsf{Dom}(\sigma N) \qquad \alpha \notin \mathrm{dom}(\Gamma_1), \mathrm{dom}(\Gamma_2)}{\vdash \sigma : \lfloor\Gamma_1\rfloor, \alpha : \mathsf{Dom}(N) \Rrightarrow \lfloor\Gamma_2\rfloor, \alpha : \mathsf{Dom}(\sigma N)}\ \text{Lemma A.11}$$

where the fourth assumption follows via the Barendregt-Convention.
The result then follows by reconstructing the K-Lam rule.
- *Case* K-All. For the first subderivation, we can directly apply the induction hypotheses and obtain $\vdash \Gamma_2, \alpha : \sigma K, \sigma\mathbb{C}$. For the second subderivation, we have to lift the substitution typing

$$\dfrac{\vdash \sigma : \Gamma_1 \Rrightarrow \Gamma_2 \qquad \vdash \Gamma_1, \alpha : K, \mathbb{C} \qquad \vdash \Gamma_2, \alpha : \sigma K, \sigma\mathbb{C} \qquad \alpha \notin \mathrm{dom}(\Gamma_1), \mathrm{dom}(\Gamma_2)}{\vdash \sigma : \Gamma_1, \alpha : K, \mathbb{C} \Rrightarrow \Gamma_2, \alpha : \sigma K, \sigma\mathbb{C}}\ \text{Lemma A.11}$$

where the fourth assumption follows from the Barendregt Convention.
The result then follows by reconstructing the K-All rule.
- *Case* K-Arr. For the premises

$$\Gamma_1 \vdash \Sigma_1 : \mathsf{State} \qquad \Gamma_1 \vdash T_1 : \mathsf{Type} \qquad \vdash \Gamma_1 ,_{\#} \Gamma$$

we directly apply the induction hypothesis and obtain

$$\Gamma_2 \vdash \sigma\Sigma_1 : \mathsf{State} \qquad \Gamma_2 \vdash \sigma T_1 : \mathsf{Type} \qquad \vdash \Gamma_2 ,_{\#} \sigma\Gamma$$

For the premises

$$\Gamma_1 ,_{\#} \Gamma \vdash \Sigma_2 : \mathsf{State} \qquad \Gamma_1 ,_{\#} \Gamma \vdash T_2 : \mathsf{Type}$$

we first lift the substitution typing

$$\dfrac{\vdash \sigma : \Gamma_1 \Rrightarrow \Gamma_2 \qquad \vdash \Gamma_1 ,_{\#} \Gamma \qquad \vdash \Gamma_2 ,_{\#} \sigma\Gamma}{\vdash \sigma : \Gamma_1 ,_{\#} \Gamma \Rrightarrow \Gamma_2 ,_{\#} \sigma\Gamma}\ \text{Lemma A.11.5}$$

and then apply the induction hypothesis to obtain

$$\Gamma_2 ,_{\#} \sigma\Gamma \vdash \sigma\Sigma_2 : \mathsf{State} \qquad \Gamma_2 ,_{\#} \sigma\Gamma \vdash \sigma T_2 : \mathsf{Type}$$

Finally, we reconstruct the K-Arr rule from the above results.

- *Case* K-Send, K-Recv. Same as K-Lam.
- The other cases follow immediately from the induction hypotheses.

(6)
- *Case* T-Var. Immediate from Axiom (3) of the substitution typing.
- The other cases follow immediately from the induction hypotheses using Lemma A.11 and the Barendregt-Convention to lift the substitution typing when going under binders.

(7)
- *Case* T-TApp. In this case we have the premise $\{T'/\alpha\}T \equiv T''$ for which the induction hypothesis yields $\sigma(\{T'/\alpha\}T) \equiv \sigma T''$ which is equivalent to the required conclusion $\{\sigma T'/\alpha\}(\sigma T) \equiv \sigma T''$ due to the Barendregt-Convention.
- *Case* T-Send. Same as T-TApp.
- *Case* T-Case. Here we have the assumption $\Gamma_1 \vdash \Sigma_1, D \mapsto S_1 \,\&\, S_2 : \mathsf{State}$. In order to apply the induction hypothesis to the branch expressions, we need to prove

$$\Gamma_1 \vdash \Sigma_1, D \mapsto S_1 : \mathsf{State} \qquad\qquad \Gamma_1 \vdash \Sigma_1, D \mapsto S_2 : \mathsf{State}$$

  which follow by simple case analysis of the assumption and K-StMerge.
- The other cases follow immediately from the induction hypotheses using Lemma A.11 and the Barendregt-Convention to lift the substitution typing when going under binders.

(8) Straightforward induction due to the Barendregt-Convention. □

Lemma A.14 (Removal of Implied Constraints). *If* $\Gamma \vdash \mathbb{C}$, *then*

$$(1)\ \frac{\Gamma, \mathbb{C} \vdash \mathbb{C}'}{\Gamma \vdash \mathbb{C}'} \qquad (2)\ \frac{\Gamma, \mathbb{C} \vdash K}{\Gamma \vdash K} \qquad (3)\ \frac{\Gamma, \mathbb{C} \vdash T : K}{\Gamma \vdash T : K} \qquad (4)\ \frac{\Gamma, \mathbb{C} \vdash v : T}{\Gamma \vdash v : T} \qquad (5)\ \frac{\Gamma, \mathbb{C}; \Sigma_1 \vdash e : \exists \Gamma_2.\Sigma_2; T}{\Gamma; \Sigma_1 \vdash e : \exists \Gamma_2.\Sigma_2; T}$$

Proof. This is a corollary of Lemma A.13 due to the substitution typing $\vdash \mathrm{id} : \Gamma, \mathbb{C} \Rightarrow \Gamma$ □

Lemma A.15 (Evaluation Context Typings for Expressions).

$$(1)\ \frac{\Gamma_1; \Sigma_1 \vdash \mathcal{E}[e] : \exists \Gamma_3.\Sigma_3; T}{\begin{array}{c}\exists \Sigma_{11}, \Sigma_{12}, \Gamma_{21}, \Gamma_{22}, \Sigma_2, T' \qquad \Sigma_1 = \Sigma_{11}, \Sigma_{12} \qquad \Gamma_3 = \Gamma_{21}, \Gamma_{22} \\ \Gamma_1; \Sigma_{11} \vdash e : \exists \Gamma_{21}.\Sigma_2; T' \qquad \Gamma_1 \,{}_{{}^\circ\!\!{}_\#}\, \Gamma_{21}, x : T'; \Sigma_{12}, \Sigma_2 \vdash \mathcal{E}[x] : \exists \Gamma_{22}.\Sigma_3; T\end{array}}$$

$$(2)\ \frac{\Gamma_1; \Sigma_{11} \vdash e : \exists \Gamma_{21}.\Sigma_2; T' \qquad \Gamma_1 \,{}_{{}^\circ\!\!{}_\#}\, \Gamma_{21}, x : T'; \Sigma_{12}, \Sigma_2 \vdash \mathcal{E}[x] : \exists \Gamma_{22}.\Sigma_3; T}{\Gamma_1; \Sigma_{11}, \Sigma_{12} \vdash \mathcal{E}[e] : \exists \Gamma_{21}, \Gamma_{22}.\Sigma_3; T}$$

Proof.

(1) By induction on the evaluation context $\mathcal{E}$:
- *Case* □. The assumption is

$$\Gamma_1; \Sigma_1 \vdash e : \exists \Gamma_3.\Sigma_3; T$$

  By choosing $\Sigma_{11} = \Sigma_1$, $\Sigma_{12} = \cdot$, $\Gamma_{21} = \Gamma_3$, $\Gamma_{22} = \cdot$, $\Sigma_2 = \Sigma_3$, $T' = T$ the goals become

$$\Sigma_1 = \Sigma_1 \tag{1}$$
$$\Gamma_3 = \Gamma_3 \tag{2}$$
$$\Gamma_1; \Sigma_1 \vdash e : \exists \Gamma_3.\Sigma_3; T \tag{3}$$
$$\Gamma_1 \,{}_{{}^\circ\!\!{}_\#}\, \Gamma_3, x : T; \Sigma_3 \vdash x : \exists \cdot.\Sigma_3; T \tag{4}$$

  (1) and (2) are trivial, (3) follows by assumption, and (4) by T-Var and T-Val.

- *Case* $\mathsf{let}\, x = \mathcal{E}\, \mathsf{in}\, e$. The assumption is

$$\textsc{T-Let}\ \frac{\Gamma_1; \Sigma_{11} \vdash \mathcal{E}[e] : \exists \Gamma_{21}.\Sigma_2; T' \qquad \Gamma_1 \,{}_{\circ\#}\, \Gamma_{21}, x : T'; \Sigma_{12}, \Sigma_2 \vdash e' : \exists \Gamma_{22}.\Sigma_3; T \qquad \Gamma_1 \,{}_{\circ\#}\, \Gamma_{21}, x : T' \vdash \Sigma_{12}, \Sigma_2 : \mathsf{State}}{\Gamma_1; \Sigma_{11}, \Sigma_{12} \vdash \mathsf{let}\, x = \mathcal{E}[e]\, \mathsf{in}\, e' : \exists \Gamma_{21}, \Gamma_{22}.\Sigma_3; T}$$

From the induction hypothesis follows

$$\text{IH}\ \frac{\Gamma_1; \Sigma_{11} \vdash \mathcal{E}[e] : \exists \Gamma_{21}.\Sigma_2; T'}{\begin{array}{c}\exists \Sigma_{111}, \Sigma_{112}, \Gamma_{21}, \Gamma_{22}, \Sigma_2', T'' \qquad \Sigma_{11} = \Sigma_{111}, \Sigma_{112} \qquad \Gamma_{21} = \Gamma_{211}, \Gamma_{212} \\ \Gamma_1; \Sigma_{111} \vdash e : \exists \Gamma_{211}.\Sigma_2'; T'' \qquad \Gamma_1 \,{}_{\circ\#}\, \Gamma_{211}, y : T''; \Sigma_{112}, \Sigma_2' \vdash \mathcal{E}[y] : \exists \Gamma_{212}.\Sigma_2; T'\end{array}}$$

By choosing $\Sigma_{11} = \Sigma_{111}, \Sigma_{12} = \Sigma_{112}, \Sigma_{12}, \Gamma_{21} = \Gamma_{211}, \Gamma_{22} = \Gamma_{212}, \Gamma_{22}, \Sigma_2 = \Sigma_2', T' = T''$ the goals become

$$\Sigma_{11}, \Sigma_{12} = \Sigma_{111}, \Sigma_{112}, \Sigma_{12} \tag{1}$$

$$\Gamma_{21}, \Gamma_{22} = \Gamma_{211}, \Gamma_{212}, \Gamma_{22} \tag{2}$$

$$\Gamma_1; \Sigma_{111} \vdash e : \exists \Gamma_{211}.\Sigma_2'; T'' \tag{3}$$

$$\Gamma_1 \,{}_{\circ\#}\, \Gamma_{211}, y : T'; \Sigma_{112}, \Sigma_{12}, \Sigma_2' \vdash \mathsf{let}\, x = \mathcal{E}[e]\, \mathsf{in}\, y : \exists \Gamma_{212}, \Gamma_{22}.\Sigma_3; T \tag{4}$$

(1), (2) and (3) are direct consequences of the IH; (4) follows via

$$\textsc{T-Let}\ \frac{\Gamma_1 \,{}_{\circ\#}\, \Gamma_{211}, y : T''; \Sigma_{112}, \Sigma_2' \vdash \mathcal{E}[y] : \exists \Gamma_{212}.\Sigma_2; T' \qquad \Gamma_1 \,{}_{\circ\#}\, \Gamma_{21}, y : T'', x : T'; \Sigma_{12}, \Sigma_2 \vdash e' : \exists \Gamma_{22}.\Sigma_3; T \qquad \Gamma_1 \,{}_{\circ\#}\, \Gamma_{21}, y : T'', x : T' \vdash \Sigma_{12}, \Sigma_2 : \mathsf{State}}{\Gamma_1 \,{}_{\circ\#}\, \Gamma_{211}, y : T''; \Sigma_{112}, \Sigma_{12}, \Sigma_2' \vdash \mathsf{let}\, x = \mathcal{E}[e]\, \mathsf{in}\, y : \exists \Gamma_{212}, \Gamma_{22}.\Sigma_3; T}$$

where the typing of e' and the kinding of Σ_{12}, Σ_2 follow by weakening for $y : T''$ via Lemma A.6.

(2) Analogous to (1). □

Lemma A.16 (Evaluation Context Typings for Configurations).

$$\frac{\Gamma; \Sigma \vdash C[C]}{\exists \Gamma', \Sigma' \qquad \Gamma'; \Sigma' \vdash C \qquad \forall C'.\, (\Gamma'; \Sigma' \vdash C') \Rightarrow (\Gamma; \Sigma \vdash C[C'])}$$

Proof. By induction on the evaluation context:

- *Case* $C = \square$. We choose $\Gamma' = \Gamma$ and $\Sigma' = \Sigma$, which reduces our goals to assumptions and tautologies.
- *Case* $C = \nu\alpha, \alpha' \mapsto S.\, C$. Here the assumption has the form

$$\textsc{T-NuChan}\ \frac{\alpha, \alpha' \text{ not free in } \Gamma \qquad \Gamma \vdash S : \mathsf{Session} \qquad \Gamma_\alpha; \Sigma_\alpha \vdash C[C]}{\nu\alpha, \alpha' \mapsto S.\, C[C]}$$

where $\Gamma_\alpha = \Gamma \,{}_{\circ\#}\, \alpha : \mathsf{Dom}(\mathbb{X}) \,{}_{\circ\#}\, \alpha' : \mathsf{Dom}(\mathbb{X})$ and $\Sigma_\alpha = \Sigma, \alpha \mapsto S, \alpha' \mapsto \overline{S}$.
From the induction hypothesis follows

$$\exists \Gamma', \Sigma' \qquad \Gamma'; \Sigma' \vdash C \qquad \forall C'.\, (\Gamma'; \Sigma' \vdash C') \Rightarrow (\Gamma_\alpha; \Sigma_\alpha \vdash C[C'])$$

For the goal we choose the same Γ' and Σ'. Let C' be some configuration such that $\Gamma'; \Sigma' \vdash C'$. From the result of the induction hypothesis follows

$$\Gamma_\alpha; \Sigma_\alpha \vdash C[C']$$

which allows us to reconstruct the T-NuChan rule.
- *Case* $C = \nu x\colon [S].\, C$. Similar as the previous case.

- *Case* $C = C \parallel C$. Similar as the previous case. □

Lemma A.17 (Wellformed Inputs imply Wellformed Outputs).

(1) $$\frac{\vdash \Gamma \qquad \Gamma \vdash T : K}{\Gamma \vdash K}$$

(2) $$\frac{\vdash \Gamma \qquad \Gamma \vdash v : T}{\Gamma \vdash T : \mathsf{Type}}$$

(3) $$\frac{\vdash \Gamma \qquad \Gamma \vdash \Sigma_1 : \mathsf{State} \qquad \Gamma; \Sigma_1 \vdash e : \exists \Gamma'.\Sigma_2; T_2}{\vdash \Gamma \,,_{\#}\, \Gamma' \qquad \Gamma \,,_{\#}\, \Gamma' \vdash \Sigma_2 : \mathsf{State} \qquad \Gamma \,,_{\#}\, \Gamma' \vdash T_2 : \mathsf{Type}}$$

Proof.

(1) By induction on the kinding derivation:
   - *Case* K-Var. Follows from the context formation due to CF-ConsKind.
   - *Case* K-App. The induction hypothesis for $\Gamma \vdash T_1 : K_1 \to K_2$ yields $\Gamma \vdash K_1 \to K_2$ which by case-analysis yields $\Gamma \vdash K_2$.
   - *Case* K-Lam. From $\Gamma \vdash N : \mathsf{Shape}$ follows $\Gamma \vdash \mathsf{Dom}(N)$. From $K \in \{\mathsf{Type}, \mathsf{State}\}$ follows $\Gamma \vdash K$. Via KF-Arr follows $\Gamma \vdash \mathsf{Dom}(N) \to K$.
   - *Case* K-DomMerge. Applying the induction hypothesis to $\Gamma \vdash D_i : \mathsf{Dom}(N_i)$ yields $\Gamma \vdash \mathsf{Dom}(N_i)$, which by case analysis on KF-Dom yields $\Gamma \vdash N_i : \mathsf{Shape}$. The result then follows from the following proof tree:

$$\dfrac{\dfrac{\Gamma \vdash N_1 : \mathsf{Shape} \qquad \Gamma \vdash N_2 : \mathsf{Shape}}{\Gamma \vdash N_1 \fatsemi N_2 : \mathsf{Shape}}\,\text{K-ShapePair}}{\Gamma \vdash \mathsf{Dom}(N_1 \fatsemi N_2)}\,\text{KF-Dom}$$

   - *Case* K-DomProj. By induction hypothesis we know $\Gamma \vdash \mathsf{Dom}(N_1 \fatsemi N_2)$, which by case analysis gives us $\Gamma \vdash N_1 \fatsemi N_2 : \mathsf{Shape}$, which by further case analysis gives us $\Gamma \vdash N_1 : \mathsf{Shape}$ and $\Gamma \vdash N_2 : \mathsf{Shape}$, which via KF-Dom gives us $\Gamma \vdash \mathsf{Dom}(N_\ell)$.
   - All other cases follow immediately via KF-Type, KF-Session, KF-State, or KF-Shape.

(2) By induction on the value typing derivation:
   - *Case* T-Var. Follows from the context formation due to CF-ConsType.
   - *Case* T-Unit. Follows directly from K-Unit.
   - *Case* T-Pair. The induction hypothesis yields $\Gamma \vdash T_1 : \mathsf{Type}$ and $\Gamma \vdash T_2 : \mathsf{Type}$, which via K-Pair yields $\Gamma \vdash T_1 \times T_2 : \mathsf{Type}$.
   - *Case* T-Abs, T-TAbs. Follows directly from the first assumption of their case's rule.
   - *Case* T-Chan. Follows directly from the assumption via K-Chan.

(3) By induction on the expression typing derivation:
   - *Case* T-Val. Follows from (2) and the assumption $\Gamma_1 \vdash \Sigma_1 : \mathsf{State}$.
   - *Case* T-Let. From the assumption $\Gamma_1 \vdash \Sigma_1, \Sigma_2 : \mathsf{State}$ follows by case analysis $\Gamma_1 \vdash \Sigma_1 : \mathsf{State}$ and $\Gamma_1 \vdash \Sigma_2 : \mathsf{State}$.

     We then apply the induction hypothesis on the typing of e_1:

$$\frac{\vdash \Gamma_1 \qquad \Gamma_1 \vdash \Sigma_1 : \mathsf{State} \qquad \Gamma_1; \Sigma_1 \vdash e_1 : \exists \Gamma_2.\Sigma_2'; T_1}{\vdash \Gamma_1 \,,_{\#}\, \Gamma_2 \qquad \Gamma_1 \,,_{\#}\, \Gamma_2 \vdash \Sigma_2' : \mathsf{State} \qquad \Gamma_1 \,,_{\#}\, \Gamma_2 \vdash T_1 : \mathsf{Type}}\,\text{IH}$$

     and extend the context formation as follows:

$$\frac{\vdash \Gamma_1 \,,_{\#}\, \Gamma_2 \qquad \Gamma_1 \,,_{\#}\, \Gamma_2 \vdash T_1 : \mathsf{Type}}{\vdash \Gamma_1 \,,_{\#}\, \Gamma_2, x : T_1}\,\text{CF-ConsType}$$

We then apply the induction hypothesis on the typing of e_2:

$$\frac{\vdash \Gamma_1 \mathbin{,_\#} \Gamma_2, x : T_1 \qquad \Gamma_1 \mathbin{,_\#} \Gamma_2, x : T_1 \vdash \Sigma_2, \Sigma_2' : \mathsf{State} \qquad \Gamma_1 \mathbin{,_\#} \Gamma_2, x : T_1; \Sigma_2, \Sigma_2' \vdash e_2 : \exists \Gamma_3.\Sigma_3; T_2}{\vdash \Gamma_1 \mathbin{,_\#} \Gamma_2, x : T_1 \mathbin{,_\#} \Gamma_3 \qquad \Gamma_1 \mathbin{,_\#} \Gamma_2, x : T_1 \mathbin{,_\#} \Gamma_3 \vdash \Sigma_3 : \mathsf{State} \qquad \Gamma_1 \mathbin{,_\#} \Gamma_2, x : T_1 \mathbin{,_\#} \Gamma_3 \vdash T_2 : \mathsf{Type}} \text{IH}$$

As value-level bindings do neither affect context formation nor kinding relations, we can safely remove the $x : T_1$ binding from the conclusion and obtain

$$\vdash \Gamma_1 \mathbin{,_\#} \Gamma_2 \mathbin{,_\#} \Gamma_3 \qquad \Gamma_1 \mathbin{,_\#} \Gamma_2 \mathbin{,_\#} \Gamma_3 \vdash \Sigma_3 : \mathsf{State} \qquad \Gamma_1 \mathbin{,_\#} \Gamma_2 \mathbin{,_\#} \Gamma_3 \vdash T_2 : \mathsf{Type}$$

Since $(\Gamma_1 \mathbin{,_\#} \Gamma_2) \mathbin{,_\#} \Gamma_3$ is equivalent to $\Gamma_1 \mathbin{,_\#} (\Gamma_2, \Gamma_3)$ up to the order of the constraints (which is irrelevant) the result follows.

- *Case* T-Proj. The first two results follow trivially. The third result follows from the induction hypothesis and subsequent case analysis.
- *Case* T-App. Applying the induction hypothesis on v_1 yields $\Gamma_1 \vdash (\Sigma_1;\, T_1 \to \exists \Gamma_2.\Sigma_2;\, T_2) : \mathsf{Type}$, which by inversion yields the results.
- *Case* T-TApp. From the induction hypothesis on the typing of v and subsequent case analysis follows $\Gamma, \alpha : K, \mathbb{C} \vdash T : \mathsf{Type}$. By Lemma A.13.5 and Lemma A.14.3 then follows $\Gamma \vdash \{T'/\alpha\}T : \mathsf{Type}$. By Lemma A.13.8 for $\{T'/\alpha\}T \equiv T''$ then follows our result $\Gamma \vdash T'' : \mathsf{Type}$.
- *Case* T-Send, T-Select. The first and third results follow trivially. Repeated case analysis on the kinding of the input state yields $\Gamma \vdash D : \mathsf{Dom}(\mathbb{X})$ and $\Gamma \vdash S : \mathsf{Session}$, which allows to construct the second result $\Gamma \vdash D \mapsto S : \mathsf{State}$ via K-StChan.
- *Case* T-Recv. Repeated case analysis on the kinding of the input state yields

  $$\Gamma \vdash D : \mathsf{Dom}(\mathbb{X}) \qquad \Gamma \vdash S : \mathsf{Session} \qquad \Gamma \vdash N : \mathsf{Shape}$$

  $$\lfloor \Gamma \rfloor, \alpha' : \mathsf{Dom}(N) \vdash \Sigma' : \mathsf{State} \qquad \lfloor \Gamma \rfloor, \alpha' : \mathsf{Dom}(N) \vdash T' : \mathsf{Type}$$

  The first result $\vdash \Gamma \mathbin{,_\#} \alpha' : \mathsf{Dom}(N)$ follows via CF-ConsKind and CF-ConsCstr. Applying Lemma A.6 (Weakening) on the kindings of Σ' and T' yields

  $$\Gamma, \alpha' : \mathsf{Dom}(N) \vdash \Sigma' : \mathsf{State} \qquad \Gamma, \alpha' : \mathsf{Dom}(N) \vdash T' : \mathsf{Type}$$

  Further weakening yields

  $$\Gamma \mathbin{,_\#} \alpha' : \mathsf{Dom}(N) \vdash \Sigma' : \mathsf{State} \qquad \Gamma \mathbin{,_\#} \alpha' : \mathsf{Dom}(N) \vdash T' : \mathsf{Type}$$

  from which the second result $\Gamma \mathbin{,_\#} \alpha' : \mathsf{Dom}(N) \vdash \Sigma', D \mapsto S : \mathsf{State}$ and third result $\Gamma \mathbin{,_\#} \alpha' : \mathsf{Dom}(N) \vdash T' : \mathsf{Type}$ can be constructed.
- *Case* T-Case. From $\Gamma \vdash \Sigma, \alpha \mapsto S_1 \,\&\, S_2 : \mathsf{State}$ follows $\Gamma \vdash \Sigma, \alpha \mapsto S_1 : \mathsf{State}$ via case analysis and kinding rules. The results then follow by the induction hypothesis on e_1.
- *Case* T-Fork, T-Close. Follows immediately from K-StEmpty and K-Unit. □

Lemma A.18 (Subject Congruence).

$$\frac{\vdash \Gamma \qquad \Gamma \vdash \Sigma : \mathsf{State} \qquad \Gamma; \Sigma \vdash C \qquad C \equiv C'}{\Gamma; \Sigma \vdash C'}$$

Proof. By induction on the congruence derivation:

- *Case* CC-Lift. Follows from the induction hypothesis in combination with Lemma A.16 to reach into the evaluation context.
- All other cases are straightforward by reordering the derivation trees of the configuration typings.

□

Lemma A.19 (Subject Reduction).

$$(1)\quad \frac{\vdash \Gamma_1 \qquad \Gamma_1 \vdash \Sigma_1 : \mathsf{State} \qquad \Gamma_1; \Sigma_1 \vdash e : \exists \Gamma_2.\Sigma_2; T \qquad e \hookrightarrow_e e'}{\exists T'.\ \Gamma_1; \Sigma_1 \vdash e' : \exists \Gamma_2.\Sigma_2; T' \land T' \equiv T}$$

$$(2)\quad \frac{\vdash \Gamma \qquad \Gamma \vdash \Sigma : \mathsf{State} \qquad \Gamma; \Sigma \vdash C \qquad C \hookrightarrow_C C'}{\exists \Sigma'.\ \Gamma; \Sigma' \vdash C'}$$

Proof.

(1) By induction on the $e \hookrightarrow_e e'$ derivation:

- *Case* ER-BetaFun. The assumptions have the following structure:

$$\textsc{ER-BetaFun}\ \frac{}{(\lambda(\Sigma_1;\, x\colon T).e_1)\ v_2 \hookrightarrow_e \{v_2/x\}e_1}$$

$$\begin{array}{l}\textsc{T-Abs}\\ \textsc{T-App}\end{array}\ \dfrac{\dfrac{\Gamma_1 \vdash (\Sigma_1;\, T_1 \to \exists\Gamma_2.\Sigma_2;\, T_2) : \mathsf{Type} \qquad \Gamma_1, x : T_1; \Sigma_1 \vdash e_1 : \exists\Gamma_2.\Sigma_2; T_2}{\Gamma_1 \vdash \lambda(\Sigma_1;\, x\colon T_1).e_1 : (\Sigma_1;\, T_1 \to \exists\Gamma_2.\Sigma_2;\, T_2)} \qquad \Gamma_1 \vdash v_2 : T_1}{\Gamma_1; \Sigma_1 \vdash (\lambda(\Sigma_1;\, x\colon T_1).e_1)\ v_2 : \exists\Gamma_2.\Sigma_2; T_2}$$

The result follows via Lemma A.13:

$$\begin{array}{l}\textsc{Lemma A.6}\\ \textsc{Lemma A.13}\end{array}\ \dfrac{\dfrac{\Gamma_1 \vdash \Sigma_1 : \mathsf{State}}{\Gamma_1, x : T_1 \vdash \Sigma_1 : \mathsf{State}} \qquad \Gamma_1, x : T_1; \Sigma_1 \vdash e_1 : \exists\Gamma_2.\Sigma_2; T_2 \qquad \vdash \{v_2/x\} : \Gamma_1, x : T_1 \Rightarrow \Gamma_1}{\Gamma_1; \Sigma_1 \vdash \{v_2/x\}e_1 : \exists\Gamma_2.\Sigma_2; T_2}$$

- *Case* ER-BetaAll. The assumptions have the following structure:

$$\textsc{ER-BetaAll}\ \frac{}{(\Lambda(\alpha\colon K).\,\mathbb{C} \Rightarrow v)\,[T'] \hookrightarrow_e \{T'/\alpha\}v}$$

$$\begin{array}{l}\textsc{T-KAll}\\ \textsc{T-TAbs}\\ \textsc{T-TApp}\end{array}\ \dfrac{\dfrac{\dfrac{\vdash \Gamma_1, \alpha : K, \mathbb{C} \qquad \Gamma_1, \alpha : K, \mathbb{C} \vdash T : \mathsf{Type}}{\Gamma_1 \vdash \forall(\alpha\colon K).\,\mathbb{C} \Rightarrow T : \mathsf{Type}} \qquad \Gamma_1, \alpha : K, \mathbb{C} \vdash v : T}{\Gamma_1 \vdash \Lambda(\alpha\colon K).\,\mathbb{C} \Rightarrow v : \forall(\alpha\colon K).\,\mathbb{C} \Rightarrow T} \qquad \Gamma_1 \vdash T' : K \qquad \Gamma_1 \vdash \{T'/\alpha\}\mathbb{C} \qquad \{T'/\alpha\}T \Downarrow T''}{\Gamma_1; \cdot \vdash (\Lambda(\alpha\colon K).\,\mathbb{C} \Rightarrow v)\,[T'] : \exists\cdot.\cdot; T''}$$

The result follows via Lemma A.13 and A.14:

$$\begin{array}{l}\textsc{Lemma A.14}\\ \textsc{T-Val}\end{array}\ \dfrac{\dfrac{\Gamma_1 \vdash \{T'/\alpha\}\mathbb{C} \qquad \textsc{Lemma A.13}\ \dfrac{\Gamma_1, \alpha : K, \mathbb{C} \vdash v : T \qquad \vdash \{T'/\alpha\} : (\Gamma_1, \alpha : K, \mathbb{C}) \Rightarrow (\Gamma_2, \{T'/\alpha\}\mathbb{C})}{\Gamma_1, \{T'/\alpha\}\mathbb{C} \vdash \{T'/\alpha\}v : \{T'/\alpha\}T}}{\Gamma_1 \vdash \{T'/\alpha\}v : \{T'/\alpha\}T}}{\Gamma_1; \cdot \vdash \{T'/\alpha\}v : \exists\cdot.\cdot; \{T'/\alpha\}T}$$

- *Case* ER-BetaLet. The assumptions have the following structure:

$$\begin{array}{l}\textsc{ER-BetaLet}\\ \mathsf{let}\, x = v_1\, \mathsf{in}\, e_2 \hookrightarrow_e \{v_1/x\}e_2\end{array}$$

$$\begin{array}{l}\textsc{T-Val}\\ \textsc{T-Let}\end{array}\ \dfrac{\dfrac{\Gamma_1 \vdash v_1 : T_1}{\Gamma_1; \cdot \vdash v_1 : \exists\cdot.\cdot; T_1} \qquad \Gamma_1, x : T_1; \Sigma_2 \vdash e_2 : \exists\Gamma_3.\Sigma_3; T_2 \qquad \Gamma_1, x : T_1 \vdash \Sigma_2 : \mathsf{State}}{\Gamma_1; \Sigma_2 \vdash \mathsf{let}\, x = v_1\, \mathsf{in}\, e_2 : \exists\Gamma_3.\Sigma_3; T_2}$$

The result follows via:

$$\begin{array}{l}\textsc{Lemma A.6}\\ \textsc{Lemma A.13}\end{array}\ \dfrac{\dfrac{\Gamma_1 \vdash \Sigma_2 : \mathsf{State}}{\Gamma_1, x : T_1 \vdash \Sigma_2 : \mathsf{State}} \qquad \Gamma_1, x : T_1; \Sigma_2 \vdash e_2 : \exists\Gamma_3.\Sigma_3; T_2 \qquad \vdash \{v_1/x\} : \Gamma_1, x : T_1 \Rightarrow \Gamma_1}{\Gamma_1; \Sigma_2 \vdash \{v_1/x\}e_2 : \exists\Gamma_3.\Sigma_3; T_2}$$

- *Case* ER-BetaPair. The assumptions have the following structure:

$$\begin{array}{l}\textsc{ER-BetaPair}\\ \pi_\ell\,(v_1, v_2) \hookrightarrow_e v_\ell\end{array}$$

$$\begin{array}{l}\textsc{T-Pair}\\ \textsc{T-Proj}\end{array}\ \dfrac{\dfrac{\Gamma \vdash v_1 : T_1 \qquad \Gamma \vdash v_2 : T_2}{\Gamma \vdash (v_1, v_2) : T_1 \times T_2}}{\Gamma; \cdot \vdash \pi_\ell\,(v_1, v_2) : \exists\cdot.\cdot; T_\ell}$$

The result follows via

$$\textsc{T-Val}\ \frac{\Gamma \vdash v_\ell : T_\ell}{\Gamma; \cdot \vdash v_\ell : \exists \cdot.\cdot; T_\ell}$$

- *Case* ER-Lift. The assumptions have the following structure:

$$\textsc{ER-Lift}\ \frac{e_1 \hookrightarrow_e e_2}{\mathsf{let}\ x = e_1\ \mathsf{in}\ e \hookrightarrow_e \mathsf{let}\ x = e_2\ \mathsf{in}\ e}$$

$$\Gamma_1; \Sigma_1 \vdash \mathcal{E}[e_1] : \exists \Gamma_3.\Sigma_3; T$$

We first extract the typing of e_1 from the evaluation context:

$$\textsc{Lemma A.15.1}\ \frac{\Gamma_1; \Sigma_1 \vdash \mathcal{E}[e_1] : \exists \Gamma_3.\Sigma_3; T}{\begin{array}{c}\exists \Sigma_{11}, \Sigma_{12}, \Gamma_{21}, \Gamma_{22}, \Sigma_2, T' \qquad \Sigma_1 = \Sigma_{11}, \Sigma_{12} \qquad \Gamma_3 = \Gamma_{21}, \Gamma_{22} \\ \Gamma_1; \Sigma_{11} \vdash e_1 : \exists \Gamma_{21}.\Sigma_2; T' \qquad \Gamma_1 ,_{\#} \Gamma_{21}, x : T'; \Sigma_{12}, \Sigma_2 \vdash \mathcal{E}[x] : \exists \Gamma_{22}.\Sigma_3; T\end{array}}$$

From $\Sigma_1 = \Sigma_{11}, \Sigma_{12}$ and $\Gamma_1 \vdash \Sigma_1 : \mathsf{State}$ follows via inversion

$$\Gamma_1 \vdash \Sigma_{11} : \mathsf{State} \qquad\qquad \Gamma_1 \vdash \Sigma_{12} : \mathsf{State}$$

Then we apply the induction hypothesis:

$$\textsc{IH}\ \frac{\vdash \Gamma_1 \qquad \Gamma_1 \vdash \Sigma_{11} : \mathsf{State} \qquad \Gamma_1; \Sigma_{11} \vdash e_1 : \exists \Gamma_{21}.\Sigma_2; T' \qquad e_1 \hookrightarrow_e e_2}{\Gamma_1; \Sigma_{11} \vdash e_2 : \exists \Gamma_{21}.\Sigma_2; T'}$$

Then we plug the typing of e_2 back into the evaluation context:

$$\textsc{Lemma A.15.2}\ \frac{\Gamma_1; \Sigma_{11} \vdash e_2 : \exists \Gamma_{21}.\Sigma_2; T' \qquad \Gamma_1 ,_{\#} \Gamma_{21}, x : T'; \Sigma_{12}, \Sigma_2 \vdash \mathcal{E}[x] : \exists \Gamma_{22}.\Sigma_3; T}{\Gamma_1; \Sigma_1 \vdash \mathcal{E}[e_2] : \exists \Gamma_3.\Sigma_3; T_2}$$

(2) By induction on the $C \hookrightarrow_C C'$ derivation. For the sake of readability, we apply Lemma A.15 and A.16 informally to talk about the typings inside evaluation contexts.

- *Case* CR-Expr. Immediate from (1).
- *Case* CR-Fork. The assumptions have the following structure:

$$\textsc{CR-Fork}\ \frac{}{C[\mathcal{E}[\mathsf{fork}\ v]] \hookrightarrow_C C[(v\ \mathsf{unit}) \parallel \mathcal{E}[\mathsf{unit}]]}$$

$$\cfrac{\cfrac{\cfrac{\cfrac{\Gamma \vdash v : (\Sigma_1; \mathsf{Unit} \to \cdot; \mathsf{Unit})}{\Gamma; \Sigma_1 \vdash \mathsf{fork}\ v : \exists \cdot.\cdot; T}\ \textsc{T-Fork}}{\Gamma; \Sigma_1, \Sigma_2 \vdash \mathcal{E}[\mathsf{fork}\ v] : \exists \Gamma'.\cdot; T}\ \textsc{Lemma A.15}}{\Gamma; \Sigma_1, \Sigma_2 \vdash \mathcal{E}[\mathsf{fork}\ v]}\ \textsc{T-Exp}}{\Gamma_0; \Sigma_0 \vdash C[\mathcal{E}[\mathsf{fork}\ v]]}\ \textsc{Lemma A.16}$$

The result follows via

$$\textsc{Lemma A.16}\ \cfrac{\textsc{T-Par}\ \cfrac{\textsc{T-Exp}\ \cfrac{\textsc{T-App}\ \cfrac{\Gamma \vdash v : (\Sigma_1; \mathsf{Unit} \to \cdot; \mathsf{Unit}) \qquad \cfrac{}{\Gamma \vdash \mathsf{unit} : \mathsf{Unit}}\ \textsc{T-Unit}}{\Gamma; \Sigma_1 \vdash v\ \mathsf{unit} : \exists \cdot.\cdot; \mathsf{Unit}}}{\Gamma; \Sigma_1 \vdash v\ \mathsf{unit}} \qquad \cfrac{}{\Gamma; \Sigma_2 \vdash \mathcal{E}[x]}\ \textsc{Lemma A.15}}{\Gamma; \Sigma_1, \Sigma_2 \vdash v\ \mathsf{unit} \parallel \mathcal{E}[x]}}{\Gamma_0; \Sigma_0 \vdash C[v\ \mathsf{unit} \parallel \mathcal{E}[x]]}$$

- *Case* CR-New. The assumptions have the following structure:

$$\textsc{CR-New}\ \frac{x\ \text{fresh}}{C[\mathcal{E}[\mathsf{new}\ S]] \hookrightarrow_C C[\nu x {:}\ [S].\ \mathcal{E}[x]]}$$

$$\textsc{Lemma A.16}\ \cfrac{\textsc{T-Exp}\ \cfrac{\textsc{Lemma A.15}\ \cfrac{\textsc{T-New}\ \cfrac{\Gamma \vdash S : \mathsf{Session}}{\Gamma; \Sigma_1 \vdash \mathsf{new}\ S : \exists \cdot.\cdot; [S]}}{\Gamma; \Sigma \vdash \mathcal{E}[\mathsf{new}\ S] : \exists \Gamma'.\cdot; T}}{\Gamma; \Sigma \vdash \mathcal{E}[\mathsf{new}\ S]}}{\Gamma_0; \Sigma_0 \vdash C[\mathcal{E}[\mathsf{new}\ S]]}$$

The result follows via

$$\textsc{Lemma A.16}\ \dfrac{\textsc{T-NuAccess}\ \dfrac{x\text{fresh} \qquad \Gamma \vdash S : \mathsf{Session} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, x : [S] \vdash x : [S]}\ \textsc{T-Var}}{\Gamma, x : [S]; \cdot \vdash x : \exists \cdot . \cdot ; [S]}\ \textsc{T-Val}}{\Gamma, x : [S]; \Sigma \vdash \mathcal{E}[x] : \exists \Gamma' . \cdot ; T}\ \textsc{Lemma A.15}}{\Gamma, x : [S]; \Sigma \vdash \mathcal{E}[x]}\ \textsc{T-Exp}}{\Gamma; \Sigma \vdash \nu x \colon [S].\ \mathcal{E}[x]}}{\Gamma_0; \Sigma_0 \vdash C[\nu x \colon [S].\ \mathcal{E}[x]]}$$

- *Case* CR-RequestAccept. The assumptions have the following structure:

$$\textsc{CR-RequestAccept}\ \dfrac{\alpha, \alpha' \text{ fresh} \qquad C \equiv C[\nu x \colon [S].\ (\mathcal{E}_1[\mathsf{request}\,x] \parallel \mathcal{E}_2[\mathsf{accept}\,x] \parallel C')]}{C \hookrightarrow_C C[\nu x \colon [S].\ \nu \alpha, \alpha' \mapsto S.\ (\mathcal{E}_1[\mathsf{chan}\,\alpha] \parallel \mathcal{E}_2[\mathsf{chan}\,\alpha'] \parallel C')]} \qquad \dfrac{\Gamma; \Sigma \vdash \nu x \colon [S].\ C}{\Gamma_0; \Sigma_0 \vdash C[\nu x \colon [S].\ C]}\ \textsc{Lemma A.16}$$

Applying Lemma A.18 to the configuration typing and the congruency yields a configuration typing, which by inversion has the the following structure, where $\Sigma = \Sigma_1, \Sigma_2, \Sigma_3$ are the channels used by $\mathcal{E}_1$, $\mathcal{E}_2$ and C', respectively:

$$\dfrac{x \text{ not free in } \Gamma \qquad \Gamma \vdash S : \mathsf{Session} \qquad \dfrac{\textsc{T-Par}\ \dfrac{(1) \qquad (2)}{\Gamma, x : [S]; \Sigma_1, \Sigma_2 \vdash (\mathcal{E}_1[\mathsf{request}\,x] \parallel \mathcal{E}_2[\mathsf{accept}\,x])} \qquad \Gamma, x : [S]; \Sigma_3 \vdash C'}{\Gamma, x : [S]; \Sigma_1, \Sigma_2, \Sigma_3 \vdash (\mathcal{E}_1[\mathsf{request}\,x] \parallel \mathcal{E}_2[\mathsf{accept}\,x] \parallel C')}\ \textsc{T-Par}}{\Gamma; \Sigma_1, \Sigma_2, \Sigma_3 \vdash \nu x \colon [S].\ (\mathcal{E}_1[\mathsf{request}\,x] \parallel \mathcal{E}_2[\mathsf{accept}\,x] \parallel C')}\ \textsc{T-NuAccess}$$

where

$$(1)\quad \dfrac{\dfrac{\dfrac{\Gamma, x : [S] \vdash x : [S]}{\Gamma, x : [S]; \cdot \vdash \mathsf{request}\,x : \exists \alpha : \mathsf{Dom}(\mathbb{X}) . \alpha \mapsto S; \mathsf{Chan}\,\alpha}\ \textsc{T-Request}}{\Gamma, x : [S]; \Sigma_1 \vdash \mathcal{E}_1[\mathsf{request}\,x] : \exists \Gamma_1', \alpha : \mathsf{Dom}(\mathbb{X}) . \cdot ; T}\ \textsc{Lemma A.15}}{\Gamma, x : [S]; \Sigma_1 \vdash \mathcal{E}_1[\mathsf{request}\,x]}\ \textsc{T-Exp}$$

$$(2)\quad \dfrac{\dfrac{\dfrac{\Gamma, x : [S] \vdash x : [S]}{\Gamma, x : [S]; \cdot \vdash \mathsf{accept}\,x : \exists \alpha : \mathsf{Dom}(\mathbb{X}) . \alpha \mapsto \overline{S}; \mathsf{Chan}\,\alpha}\ \textsc{T-Accept}}{\Gamma, x : [S]; \Sigma_1 \vdash \mathcal{E}_2[\mathsf{accept}\,x] : \exists \Gamma_2', \alpha : \mathsf{Dom}(\mathbb{X}) . \cdot ; T}\ \textsc{Lemma A.15}}{\Gamma, x : [S]; \Sigma_1 \vdash \mathcal{E}_2[\mathsf{accept}\,x]}\ \textsc{T-Exp}$$

The result follows via

$$\dfrac{\dfrac{x \text{ not free in } \Gamma \qquad \Gamma \vdash S : \mathsf{Session} \qquad \dfrac{\alpha, \alpha' \text{ fresh} \qquad \Gamma \vdash S : \mathsf{Session} \qquad (3)}{\Gamma, x : [S]; \Sigma_1, \Sigma_2, \Sigma_3 \vdash \nu \alpha, \alpha' \mapsto S.\ (\mathcal{E}_1[\mathsf{chan}\,\alpha] \parallel \mathcal{E}_2[\mathsf{chan}\,\alpha'] \parallel C')}\ \textsc{T-NuChan}}{\Gamma; \Sigma_1, \Sigma_2, \Sigma_3 \vdash \nu x \colon [S].\ \nu \alpha, \alpha' \mapsto S.\ (\mathcal{E}_1[\mathsf{chan}\,\alpha] \parallel \mathcal{E}_2[\mathsf{chan}\,\alpha'] \parallel C')}\ \textsc{T-NuAccess}}{\Gamma_0; \Sigma_0 \vdash C[\nu x \colon [S].\ \nu \alpha, \alpha' \mapsto S.\ (\mathcal{E}_1[\mathsf{chan}\,\alpha] \parallel \mathcal{E}_2[\mathsf{chan}\,\alpha'] \parallel C')]}\ \textsc{Lemma A.16}$$

where

$$(3)\quad \textsc{T-Par}\ \dfrac{\textsc{T-Par}\ \dfrac{(4) \qquad (5)}{\Gamma'; \Sigma_1, \Sigma_2, \alpha \mapsto S, \alpha' \mapsto \overline{S} \vdash (\mathcal{E}_1[\mathsf{chan}\,\alpha] \parallel \mathcal{E}_2[\mathsf{chan}\,\alpha'])} \qquad \Gamma'; \Sigma_3 \vdash C'}{\Gamma'; \Sigma_1, \Sigma_2, \Sigma_3, \alpha \mapsto S, \alpha' \mapsto \overline{S} \vdash (\mathcal{E}_1[\mathsf{chan}\,\alpha] \parallel \mathcal{E}_2[\mathsf{chan}\,\alpha'] \parallel C')}$$

for $\Gamma' = \Gamma, x : [S] \mathbin{,_{\#}} \alpha : \mathsf{Dom}(\mathbb{X}) \mathbin{,_{\#}} \alpha' : \mathsf{Dom}(\mathbb{X})$

$$(4)\quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma' \vdash \alpha : \mathsf{Dom}(\mathbb{X})}\ \textsc{T-TVar}}{\Gamma' \vdash \mathsf{chan}\,\alpha : \mathsf{Chan}\,\alpha}\ \textsc{T-Chan}}{\Gamma'; \cdot \vdash \mathsf{chan}\,\alpha : \exists \cdot . \cdot ; \mathsf{Chan}\,\alpha}\ \textsc{T-Val}}{\Gamma'; \Sigma_1, \alpha \mapsto S \vdash \mathcal{E}_1[\mathsf{chan}\,\alpha] : \exists \Gamma_1' . \cdot ; T}\ \textsc{Lemma A.15}}{\Gamma'; \Sigma_1, \alpha \mapsto S \vdash \mathcal{E}_1[\mathsf{chan}\,\alpha]}\ \textsc{T-Exp}$$

(5) Similar to (4).

Note that the channels, which in the pre-reduction tree are introduced existentially by the request · and accept · operations, are in the post-reduction tree provided from the outside via the ν-Binder. Lemma A.15 is strong enough to support this.

- *Case* CR-SendRecv. The assumptions have the following structure:

$$\textsc{CR-SendRecv}\ \frac{C \equiv (\mathcal{E}_1[\mathsf{send}\ v\ \mathsf{on\ chan}\ \alpha] \parallel \mathcal{E}_2[\mathsf{receive\ chan}\ \alpha'] \parallel C')}{\nu\alpha,\alpha' \mapsto S'.\, C \hookrightarrow_C \nu\alpha,\alpha' \mapsto S.\, (\mathcal{E}_1[\mathsf{unit}] \parallel \mathcal{E}_2[v] \parallel C')}$$

$$\frac{\dfrac{\alpha,\alpha' \text{ not free in } \Gamma \qquad \Gamma \vdash S' : \mathsf{Session} \qquad \Gamma';\Sigma,\alpha\mapsto S',\alpha'\mapsto\overline{S'} \vdash C}{\Gamma;\Sigma \vdash \nu\alpha,\alpha'\mapsto S'.\,C}\ \textsc{T-NuChan}}{\Gamma_0;\Sigma_0 \vdash C[\nu\alpha,\alpha'\mapsto S'.\,C]}\ \textsc{Lemma A.16}$$

where $\Gamma' = \Gamma \,,_{\#}\, \alpha : \mathsf{Dom}(\mathbb{X}) \,,_{\#}\, \alpha' : \mathsf{Dom}(\mathbb{X})$ and $S' = !(\exists\alpha'' : \mathsf{Dom}(N).\Sigma'; T').S$. Applying Lemma A.18 to the configuration typing of C and the congruency yields a configuration typing, which by inversion reveals the following structure with $\Sigma = \Sigma_{!}, \Sigma_1, \Sigma_2, \Sigma_3$, where $\Sigma_{!}$ are the channels that are sent and received, and Σ_1, Σ_2, and Σ_3 are the channels used in $\mathcal{E}_1$, $\mathcal{E}_2$, and C', respectively:

$$\frac{\dfrac{(1) \qquad (2)}{\Gamma';\Sigma_{!},\Sigma_1,\Sigma_2,\alpha\mapsto S',\alpha'\mapsto\overline{S'} \vdash (\mathcal{E}_1[\mathsf{send}\ v\ \mathsf{on\ chan}\ \alpha] \parallel \mathcal{E}_2[\mathsf{receive\ chan}\ \alpha'])}\ \textsc{T-Par} \qquad \Gamma';\Sigma_3 \vdash C'}{\Gamma';\Sigma_{!},\Sigma_1,\Sigma_2,\Sigma_3,\alpha\mapsto S',\alpha'\mapsto\overline{S'} \vdash (\mathcal{E}_1[\mathsf{send}\ v\ \mathsf{on\ chan}\ \alpha] \parallel \mathcal{E}_2[\mathsf{receive\ chan}\ \alpha'] \parallel C')}\ \textsc{T-Par}$$

where

$$(1)\ \frac{\dfrac{\dfrac{\Gamma \vdash D : \mathsf{Dom}(N) \qquad \{D/\alpha''\}\Sigma' \equiv \Sigma_{!} \qquad \{D/\alpha''\}T' \equiv T'' \qquad \Gamma \vdash v : T'' \qquad \Gamma \vdash \mathsf{chan}\ \alpha : \mathsf{Chan}\ \alpha}{\Gamma';\Sigma_{!},\alpha\mapsto S' \vdash \mathsf{send}\ v\ \mathsf{on\ chan}\ \alpha : \exists\cdot.\alpha\mapsto S;\mathsf{Unit}}\ \textsc{T-Send}}{\Gamma';\Sigma_{!},\Sigma_1,\alpha\mapsto S' \vdash \mathcal{E}_1[\mathsf{send}\ v\ \mathsf{on\ chan}\ \alpha] : \exists\Gamma_1'.\cdot;T_1}\ \textsc{Lemma A.15}}{\Gamma';\Sigma_{!},\Sigma_1,\alpha\mapsto S' \vdash \mathcal{E}_1[\mathsf{send}\ v\ \mathsf{on\ chan}\ \alpha]}\ \textsc{T-Exp}$$

$$(2)\ \frac{\dfrac{\dfrac{\Gamma \vdash \alpha' : \mathsf{Dom}(\mathbb{X}) \qquad \Gamma \vdash \mathsf{chan}\ \alpha' : \mathsf{Chan}\ \alpha'}{\Gamma';\alpha'\mapsto\overline{S'} \vdash \mathsf{receive\ chan}\ \alpha' : \exists\alpha'' : \mathsf{Dom}(N).\Sigma',\alpha'\mapsto\overline{S};T'}\ \textsc{T-Recv}}{\Gamma';\Sigma_2,\alpha'\mapsto\overline{S'} \vdash \mathcal{E}_2[\mathsf{receive\ chan}\ \alpha'] : \exists\Gamma_2',\alpha'' : \mathsf{Dom}(N).\cdot;T_2}\ \textsc{Lemma A.15}}{\Gamma';\Sigma_2,\alpha'\mapsto\overline{S'} \vdash \mathcal{E}_2[\mathsf{receive\ chan}\ \alpha']}\ \textsc{T-Exp}$$

The result follows via

$$\frac{\dfrac{\alpha,\alpha' \text{ not free in } \Gamma \qquad \Gamma \vdash S : \mathsf{Session} \qquad \dfrac{\dfrac{(3) \qquad (4)}{\Gamma';\Sigma_{!},\Sigma_1,\Sigma_2,\alpha\mapsto S,\alpha'\mapsto\overline{S} \vdash \mathcal{E}_1[\mathsf{unit}] \parallel \mathcal{E}_2[v]}\ \textsc{T-Par} \qquad \Gamma';\Sigma_3 \vdash C'}{\Gamma';\Sigma_{!},\Sigma_1,\Sigma_2,\Sigma_3,\alpha\mapsto S,\alpha'\mapsto\overline{S} \vdash \mathcal{E}_1[\mathsf{unit}] \parallel \mathcal{E}_2[v] \parallel C'}\ \textsc{T-Par}}{\Gamma;\Sigma_{!},\Sigma_1,\Sigma_2,\Sigma_3 \vdash \nu\alpha,\alpha'\mapsto S.\,(\mathcal{E}_1[\mathsf{unit}] \parallel \mathcal{E}_2[v] \parallel C')}\ \textsc{T-NuChan}}{\Gamma_0;\Sigma_0 \vdash C[\nu\alpha,\alpha'\mapsto S.\,(\mathcal{E}_1[\mathsf{unit}] \parallel \mathcal{E}_2[v] \parallel C')]}\ \textsc{Lemma A.16}$$

where

$$(3)\ \frac{\dfrac{\dfrac{\dfrac{}{\Gamma' \vdash \mathsf{unit} : \mathsf{Unit}}\ \textsc{T-Unit}}{\Gamma';\cdot \vdash \mathsf{unit} : \exists\cdot.\cdot;\mathsf{Unit}}\ \textsc{T-Val}}{\Gamma';\Sigma_1,\alpha\mapsto S \vdash \mathcal{E}_1[\mathsf{unit}] : \exists\Gamma_1'.\cdot;T_1}\ \textsc{Lemma A.15}}{\Gamma';\Sigma_1,\alpha\mapsto S \vdash \mathcal{E}_1[\mathsf{unit}]}\ \textsc{T-Exp}$$

$$(4)\ \frac{\dfrac{\dfrac{\Gamma' \vdash v : T'}{\Gamma';\cdot \vdash v : \exists\cdot.\cdot;T'}\ \textsc{T-Val}}{\Gamma';\Sigma_{!},\Sigma_2,\alpha'\mapsto\overline{S} \vdash \mathcal{E}_2[v] : \exists\Gamma_2'.\cdot;T_2}\ \textsc{Lemma A.15}}{\Gamma';\Sigma_{!},\Sigma_2,\alpha'\mapsto\overline{S} \vdash \mathcal{E}_2[v]}\ \textsc{T-Exp}$$

- *Case* CR-SelectCase, CR-Close. Similar to the previous cases.

□

Lemma A.20 (Context inversion).

$$\frac{\vdash \Gamma \qquad \mathsf{Outer}\,\Gamma \qquad \Gamma \vdash \Sigma : \mathsf{State} \qquad \Gamma; \Sigma \vdash C[e]}{\exists \Gamma', \Sigma', T.\ \vdash \Gamma' \qquad \mathsf{Outer}\,\Gamma' \qquad \Gamma' \vdash \Sigma' : \mathsf{State} \qquad \Gamma'; \Sigma' \vdash e : \exists \cdot . \cdot ; T}$$

Proof. By induction on $\Gamma; \Sigma \vdash C[e]$. □

Lemma A.21 (Canonical forms). *Suppose that* $\Gamma \vdash v : T$ *and* $\mathsf{Outer}\,\Gamma$.

- *If* T *is* $(\Sigma;\ T \to \exists \Gamma'.\Sigma';\ T')$, *then* v *is* $\lambda(\Sigma;\ x\colon T).e$, *for some* e.
- *If* T *is* $T_1 \times T_2$, *then* v *is* (v_1, v_2), *for some* v_1 *and* v_2.
- *If* T *is* $\forall(\alpha\colon K).\ \mathbb{C} \Rightarrow T$, *then* v *is* $\Lambda(\alpha\colon K).\ \mathbb{C} \Rightarrow v_1$, *for some* v_1.
- *If* T *is* Unit, *then* v *is* unit.
- *If* T *is* $\mathsf{Chan}\,D$, *then* v *is* $\mathsf{chan}\,D$, *for some* D.
- *If* T *is* $[S]$, *then* v *is* x, *for some* $x \in \mathrm{dom}(\Gamma)$.

Proof. By inversion of the value typing judgment $\Gamma \vdash v : T$. □

Lemma 4.5 (Progress for expressions).

$$\frac{\vdash \Gamma \qquad \mathsf{Outer}\,\Gamma \qquad \Gamma \vdash \Sigma : \mathsf{State} \qquad \Gamma; \Sigma \vdash e : \exists \Gamma'.\Sigma'; T'}{\mathsf{Value}\, e \vee \mathsf{Comm}\, e \vee \exists e'.\ e \hookrightarrow_e e'}$$

Proof. The proof is by induction on the expression e.
Case v: Value v holds.
Case $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$: By the IH for e_1 we have three cases:

- if Value e_1, then $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \hookrightarrow_e \{e_1/x\}e_2$ by ER-BetaLet;
- if Comm e_1, then Comm ($\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$);
- if $e_1 \hookrightarrow_e e_1'$, then the let reduces, too.

Case $v_1\ v_2$: Inversion of $\Gamma; \Sigma \vdash v_1\ v_2 : \exists \Gamma'.\Sigma'; T'$ yields

$$\Gamma \vdash v_1 : (\Sigma;\ T \to \exists \Gamma'.\Sigma';\ T') \tag{12}$$

$$\Gamma \vdash v_2 : T \tag{13}$$

By Lemma A.21, $v_1 = \lambda(\Sigma;\ x\colon T).e_1$, for some e_1. Hence, $v_1\ v_2 \hookrightarrow_e$ by ER-BetaFun.
Case $\pi_\ell\, v$: Inversion of $\Gamma; \Sigma \vdash \pi_\ell\, v : \exists \Gamma'.\Sigma'; T'$ yields

$$\Gamma \vdash v : T_1 \times T_2 \tag{14}$$

By Lemma A.21, $v = (v_1, v_2)$, for some v_1 and v_2. Hence, $\pi_\ell\, v \hookrightarrow_e$ by ER-BetaPair.
Case $v[T'']$: Inversion of $\Gamma; \Sigma \vdash v[T''] : \exists \Gamma'.\Sigma'; T'$ yields

$$\Gamma \vdash v : \forall(\alpha\colon K).\ \mathbb{C} \Rightarrow T \tag{15}$$

$$\Gamma \vdash T'' : K \tag{16}$$

$$\Gamma \vdash \{T''/\alpha\}\mathbb{C} \tag{17}$$

$$\{T''/\alpha\}T \equiv T' \tag{18}$$

By Lemma A.21, v is $\Lambda(\alpha\colon K).\ \mathbb{C} \Rightarrow v_1$, for some v_1. Hence, $v[T''] \hookrightarrow_e$ by ER-BetaAll.
Case $\mathsf{fork}\, v$: Inversion of $\Gamma; \Sigma \vdash \mathsf{fork}\, v : \exists \Gamma'.\Sigma'; T'$ yields $\Gamma' = \cdot$, $\Sigma' = \cdot$, $T' = \mathsf{Unit}$, and

$$\Gamma \vdash v : (\Sigma;\ \mathsf{Unit} \to \cdot;\ \mathsf{Unit}) \tag{19}$$

By Lemma A.21, v is $\lambda(\Sigma;\ x\colon \mathsf{Unit}).e_1$, hence $\mathsf{Comm}\,(\mathsf{fork}\,v)$.

Case $\mathsf{new}\,S$: Inversion of $\Gamma; \Sigma \vdash \mathsf{new}\,S : \exists\Gamma'.\Sigma'; T'$ yields $\Sigma = \cdot$, $\Gamma' = \cdot$, $\Sigma' = \cdot$, and $T' = [S]$. Hence $\mathsf{Comm}\,(\mathsf{new}\,S)$.

Case $\mathsf{accept}\,v$: Inversion of $\Gamma; \Sigma \vdash \mathsf{accept}\,v : \exists\Gamma'.\Sigma'; T'$ yields

$$\Gamma \vdash v : [S] \tag{20}$$

By Lemma A.21, v is x, hence $\mathsf{Comm}\,(\mathsf{accept}\,v)$.

Case $\mathsf{request}\,v$: by similar reasoning, $v = x$ and $\mathsf{Comm}\,(\mathsf{request}\,v)$.

Case $\mathsf{send}\,v_1\,\mathsf{on}\,v_2$: Inversion of $\Gamma; \Sigma \vdash \mathsf{send}\,v_1\,\mathsf{on}\,v_2 : \exists\Gamma'.\Sigma'; T'$ yields

- $\Sigma = \Sigma_1, D \mapsto\, !(\exists\alpha'\colon \mathsf{Dom}(N).\Sigma';\ T'').S$,
- $\Gamma' = \cdot$,
- $\Sigma' = D \mapsto S$,
- $T' = \mathsf{Unit}$, and

$$\Gamma \vdash D' : \mathsf{Dom}(N) \tag{21}$$
$$\{D'/\alpha'\}\Sigma' \equiv \Sigma_1 \tag{22}$$
$$\{D'/\alpha'\}T'' \equiv T \tag{23}$$
$$\Gamma \vdash D : \mathsf{Dom}(\mathbb{X}) \tag{24}$$
$$\Gamma \vdash v_1 : T \tag{25}$$
$$\Gamma \vdash v_2 : \mathsf{Chan}\,D \tag{26}$$

By Lemma A.21, v_2 is $\mathsf{chan}\,D$, hence $\mathsf{Comm}\,(\mathsf{send}\,v_1\,\mathsf{on}\,v_2)$.

Case $\mathsf{receive}\,v$: by similar reasoning as in the previous case, $v = \mathsf{chan}\,D$ and $\mathsf{Comm}\,(\mathsf{receive}\,v)$.

Case $\mathsf{select}\,\ell\,\mathsf{on}\,v$: by similar reasoning, $v = \mathsf{chan}\,D$ and $\mathsf{Comm}\,(\mathsf{select}\,\ell\,\mathsf{on}\,v)$.

Case $\mathsf{case}\,v\,\mathsf{of}\{e_1;\ e_2\}$: by similar reasoning, $v = \mathsf{chan}\,D$ and $\mathsf{Comm}\,(\mathsf{case}\,v\,\mathsf{of}\{e_1;\ e_2\})$.

Case $\mathsf{close}\,v$: by similar reasoning, $v = \mathsf{chan}\,D$ and $\mathsf{Comm}\,(\mathsf{close}\,v)$. □

Lemma 4.8 (Progress for configurations).

$$\frac{\vdash \Gamma \qquad \mathsf{Outer}\,\Gamma \qquad \Gamma \vdash \Sigma : \mathsf{State} \qquad \Gamma; \Sigma \vdash C}{\mathsf{Final}\,C \lor \mathsf{Deadlock}\,C \lor \exists C'.\ C \hookrightarrow_C C'}$$

Proof. Suppose that $\neg\,\mathsf{Final}\,C$ and $\neg\,\mathsf{Deadlock}\,C$. Hence, one of the three items in Definition 4.7 must be violated and we show that C reduces in each case.

Suppose item 1 is violated. Hence, there is some $\mathcal{C}$ such that $C = \mathcal{C}[e]$ and $\neg\,\mathsf{Value}\,e$ and $\neg\,\mathsf{Comm}\,e$ or $e = \mathcal{E}[\mathsf{fork}\,v]$ or $e = \mathcal{E}[\mathsf{new}\,S]$, for some $\mathcal{E}$, v, S.

If $e = \mathcal{E}[\mathsf{fork}\,v]$, then $v = \lambda(\Sigma;\ x\colon \mathsf{Unit}).e_1$ and C reduces as follows

$$\mathcal{C}[\mathcal{E}[\mathsf{fork}\,\lambda(\Sigma;\ x\colon \mathsf{Unit}).e_1]] \hookrightarrow_C \mathcal{C}[\mathcal{E}[\mathsf{unit}] \parallel (\lambda(\Sigma;\ x\colon \mathsf{Unit}).e_1)\ \mathsf{unit}]$$

If $e = \mathcal{E}[\mathsf{new}\,S]$, then C reduces by CR-New.

Otherwise, by Lemma 4.5 (which is applicable because of context inversion, Lemma A.20), there exists some e' such that $e \hookrightarrow_e e'$. Hence, $\mathcal{C}[e] \hookrightarrow_C \mathcal{C}[e']$.

Suppose item 2 is violated. That is, there are configuration and evaluation contexts $\mathcal{C}$, $\mathcal{C}_1$, $\mathcal{C}_2$, $\mathcal{E}_1$, and $\mathcal{E}_2$ such that $C = \mathcal{C}[\nu x\colon [S].\ C']$ and $C' = \mathcal{C}_1[\mathcal{E}_1[\mathsf{request}\,x]]$ and $C' = \mathcal{C}_2[\mathcal{E}_2[\mathsf{accept}\,x]]$. Exploiting congruence we can find a configuration context $\mathcal{C}'$ and process C'' such that $C \equiv \mathcal{C}'[\nu x\colon [S].\ \mathcal{E}_1[\mathsf{request}\,x] \parallel \mathcal{E}_2[\mathsf{accept}\,x] \parallel C'']$, which reduces by CR-RequestAccept.

Suppose item 3 is violated. Consider the case for send _ on _ and receive. That is, there are configuration and evaluation contexts $\mathcal{C}$, $\mathcal{C}_1$, $\mathcal{C}_2$, $\mathcal{E}_1$, and $\mathcal{E}_2$ such that $C = \mathcal{C}[\nu\alpha_1, \alpha_2 \mapsto S.\ C']$ and

$C' = C_1[\mathcal{E}_1[\textsf{send}\, v \,\textsf{on chan}\, \alpha_\ell]]$ and $C' = C_2[\mathcal{E}_2[\textsf{receive chan}\, \alpha_{3-\ell}]]$. Exploiting congruence we can find a configuration context C' and process C'' such that

$$C \equiv C'[\nu\alpha_1, \alpha_2 \mapsto S.\, \mathcal{E}_1[\textsf{send}\, v \,\textsf{on chan}\, \alpha_\ell] \parallel \mathcal{E}_2[\textsf{receive chan}\, \alpha_{3-\ell}] \parallel C'']$$

which reduces by CR-SendRecv.

The remaining cases are similar.

□