

Polymorphic Typestate for Session Types

Hannes Saffrich
University of Freiburg
Germany

saffrich@informatik.uni-freiburg.de

Peter Thiemann
University of Freiburg
Germany

thiemann@informatik.uni-freiburg.de

ABSTRACT

Session types provide a principled approach to typed communication protocols that guarantee type safety and protocol fidelity. Formalizations of session-typed communication are typically based on process calculi, concurrent lambda calculi, or linear logic. An alternative model based on context-sensitive typing and typestate has not received much attention due to its apparent restrictions. However, this model is attractive because it does not force programmers into particular patterns like continuation-passing style or channel-passing style, but rather enables them to treat communication channels like mutable variables.

Polymorphic typestate is the key that enables a full treatment of session-typed communication. Previous work in this direction was hampered by its setting in a simply-typed lambda calculus. We show that higher-order polymorphism and existential types enable us to lift the restrictions imposed by the previous work, thus bringing the expressivity of the typestate-based approach on par with the competition. On this basis, we define PolyVGR, the system of polymorphic typestate for session types, establish its basic metatheory, type preservation and progress, and present a prototype implementation.

KEYWORDS

binary session types, typestate, polymorphism, existential types

1 INTRODUCTION

When Honda and others [18, 35] 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, heterogeneously typed communication channels between two processes in pi-calculus. It also contains facilities for offering and accepting choices in the protocol.

Subsequent work added 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 [15, 19, 22, 24, 32]. These embeddings impose particular programming styles, following the structure of session types. For example, embeddings in linear functional languages [15, 22] impose writing code in what we call *channel-passing style* as demonstrated in Listing 1.

```
let (x, c2) = receive c1 in
let (y, c3) = receive c2 in
let c4 = send (x + y, c3) in ...
```

Listing 1: Channel-passing style

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

Listing 3: Example server

Listing 4: Example server with capture

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 $?T.S \rightarrow (T \times S)$. When it consumes $c1$, it returns $c2$ of type $?Int. !Int. s0$, which is further transformed to $c3$ of type $!Int. s0$ by the next **receive**, and finally to $c4 : s0$ by the **send** operation of type $(T \times !T.S) \rightarrow S$.

Writing a program in this style is cumbersome as programmers have to thread the channel explicitly through the program. This style is not safe for embedding session types in general programming languages because most languages do not enforce the linearity needed to avoid aliasing of channel ends at compile time (some implementations check linear use at run time [24, 32]). Wrapping the channel passing in a parameterized monad [4] would accommodate the typing requirements and ensure linearity by encapsulation, 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 [26] developed a Haskell implementation of session types in this style. In object-oriented languages, fluent interfaces enable the correct chaining of method calls according to a session type [20], but have similar issues as channel-passing style when scaling to multiple channels and new issues with receiving values which seems to require mutable references as shown in Listing 2.

```
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

An alternative approach is inspired by systems with typestate [34]. Vasconcelos et al. [37] proposed a multithreaded functional language on this basis. Their language, which we call VGR, enables programming in direct style; it does not require linear handling of variables; and it scales to multiple channels. Listing 3 contains a program fragment in VGR equivalent to the code in Listing 1. The parameter u of the server function is a *channel reference* of type **Chan** α , where α is a variable representing a *channel identity*. The

operation **receive** takes a channel reference associated with session type $!Int.S$ and returns an integer. The association is maintained at compile time in a typestate $\Sigma = \{\alpha \mapsto !Int.S\}$ that maps channel identities to session types. As a compile-time side effect, **receive** changes the typestate to $\Sigma' = \{\alpha \mapsto S\}$. Thus, we can describe the action of **receive** by the typing:

$$\text{receive} : \{\alpha \mapsto !Int.S\}; \text{Chan } \alpha \rightarrow (\text{Chan } \alpha \times Int); \{\alpha \mapsto S\}$$

The general shape of a function type in VGR is thus: $\Sigma_1; T_1 \rightarrow T_2; \Sigma_2$. Here, T_1 and T_2 are argument and return type of the function. The typestate environments Σ_1 and Σ_2 map channel identities to session types. They reflect the state (session type) of the channels before (Σ_1) and after (Σ_2) calling the function. Channels in T_1 refer to entries in Σ_1 and channels in T_2 refer to entries in Σ_2 .

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 . Putting these typings together, we obtain the VGR type of the server function in Listing 3:

$$\{\alpha : ?Int.?Int.!Int.S\}; \text{Chan } \alpha \rightarrow \text{Unit}; \{\alpha : S\}, \quad (1)$$

for some channel name α and session type S . Listing 3 also 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 the typestate Σ , which is threaded linearly through the typing rules, at compile time.

As VGR is based on simple types, the typing (1) is severely restricted.

- (1) The function server is tied to a single continuation session type S , a restriction shared with many functional systems [12, 15].
- (2) The function server can only be called on the single channel identified by α .

The language PolyVGR that we propose here fixes all those drawbacks, and more. The PolyVGR type for server abstracts over continuation sessions and channel identities:¹

$$\forall(\sigma : \text{Session}). \forall(\alpha : \text{Dom}(\mathbb{X})). \{\alpha : ?Int.?Int.!Int.\sigma\}; \text{Chan } \alpha \rightarrow \exists \cdot \{\alpha : \sigma\}; \text{Unit} \quad (2)$$

Quantification over session types, as in $\forall(\sigma : \text{Session})$, has been considered and analyzed in other recent work [1, 22].

The quantification of α is novel to PolyVGR. Its kind, $\text{Dom}(\mathbb{X})$, indicates that α ranges over all channel identities. We call \mathbb{X} a *shape*. Shapes allow us to talk about and quantify over the (channel) resources embedded in a value in PolyVGR. For example, $\mathbb{X} \# \mathbb{X}$ is the shape to describe a value with two embedded channels. This facility enables PolyVGR to provide a single typing rule for the operations **receive** and **send_on_**: In VGR, there are two separate typing rules, one to transmit data values and another to transmit one single channel. Shapes also facilitate an extension of PolyVGR with algebraic datatypes like lists, which was not considered in previous work.

¹Boxes with a frame highlight types and expressions of PolyVGR.

A final ingredient of the function type in PolyVGR is the innocuous existential right of the function arrow in (1). The existential addresses another shortcoming exhibited by this VGR type:

$$\{\}; \text{Unit} \rightarrow \text{Chan } \alpha; \{\alpha : S\} \quad (3)$$

A function of type (3) must create a new channel of session type S . But lacking polymorphism, each invocation of this function has to create a channel with the same identity α . To avoid this limitation, PolyVGR handles newly created channels (and other resources) using existential quantification:

$$\{\}; \text{Unit} \rightarrow \exists(\alpha : \text{Dom}(\mathbb{X})). \{\alpha : S\}; \text{Chan } \alpha \quad (4)$$

Every use of a function of this type gives rise to a new channel identity. Thanks to the existential, this identity is renamed as needed to avoid clashes with any existing channel identity in the context.

Coming back to the examples, let us have a look at function server' in Listing 4. This function contains a free variable u with a channel reference of type $\text{Chan } \alpha$. It can only be used in a context that provides the same channel α , which is somewhat hidden in the VGR type of server':

$$\{\alpha : ?Int.?Int.!Int.S\}; \text{Unit} \rightarrow \text{Unit}; \{\alpha : S\}, \quad (5)$$

but which becomes very clear in its PolyVGR type:

$$\forall(\sigma : \text{Session}). \{\alpha : ?Int.?Int.!Int.\sigma\}; \text{Unit} \rightarrow \exists \cdot \{\alpha : \sigma\}; \text{Unit}. \quad (6)$$

The lack of quantification over α indicates that it is not 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'. In any case, we can invoke a function of type (5) or of type (1) any time the channel α is in a state matching the “before” session type of the function.

Contributions

- We define PolyVGR, a novel session type system based on polymorphic typestate 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 on the basis of a standard synchronous semantics for session types (see Section 4).
- Type checking for PolyVGR is decidable and implemented (see Section 5). We plan to submit the implementation for artifact evaluation.
- We informally sketch an extension of PolyVGR for sum types that may contain channels (see Section 6).

Proofs and some extra examples may be found in the supplement.

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 and existential quantification gives us fine grained control over channel identity management, channel passing between processes as well as channel creation.

The next few subsections systematically explain the innovations of PolyVGR compared to VGR. Types and code fragments for the new calculus PolyVGR appear in boxes with a frame. 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 (1) and (5); Section 2.2).
- A function can abstract over the creation of an arbitrary number of channels because the names of newly created channels are existentially quantified (see (7) and (12); Section 2.1).
- Arbitrary data structures can be transmitted. Ownership of all channels contained in the data structure is transferred to the receiver (see (10); Section 2.3).
- Abstraction over transmission operations is possible. In particular, a type can be given to a fully flexible send or receive operation (see (10)).

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 \bar{S} (the dual type of S) on the client.

$$\begin{array}{c}
 \text{C-ACCEPT} \\
 \hline
 \Gamma; v \mapsto [S] \quad \text{fresh } c \\
 \hline
 \Gamma; \Sigma; \text{accept } v \mapsto \Sigma; \mathbf{Chan } c; \{c : S\} \\
 \\
 \text{C-REQUEST} \\
 \hline
 \Gamma; v \mapsto [S] \quad \text{fresh } c \\
 \hline
 \Gamma; \Sigma; \text{request } v \mapsto \Sigma; \mathbf{Chan } c; \{c : \bar{S}\}
 \end{array}$$

In the VGR typing rules for these operations, new channels just show up with a fresh name in the outgoing state of the expression typing. Similarly, if a function of type $\Sigma_1; T_1 \rightarrow T_2; \Sigma_2$ 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 using existential quantification. As an example, consider

abstracting over the accept operation:

$$\begin{array}{c}
 acc = \Lambda(\sigma : \text{Session}). \lambda(\cdot; x : [\sigma]). \text{accept } x \\
 : \forall(\sigma : \text{Session}). (\cdot; [\sigma] \rightarrow \exists \gamma : \text{Dom}(\mathbb{X}). \gamma \mapsto \sigma; \text{Chan } \gamma)
 \end{array} \quad (7)$$

The core of this type still has a shape like the VGR type $\Sigma_1; T_1 \rightarrow T_2; \Sigma_2$, 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:

$$(\Sigma_1; T_1 \rightarrow \exists \alpha : \text{Dom}(n). \Sigma_2; T_2). \quad (8)$$

The incoming state Σ_1 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. On return, a function can provide new entries in the state, which are disjointly added to the calling state, by way of the existential $\exists \alpha : \text{Dom}(n)$.

The type of *acc* in (7) is universally quantified over a session type, $\sigma : \text{Session}$, to work with arbitrary access points. Left of the arrow, the required incoming state is empty \cdot and argument of type $[\sigma]$ is an access point for σ . Right of the arrow, the existential quantification $\exists \gamma : \text{Dom}(\mathbb{X})$ abstracts over the created channel. The kind $\text{Dom}(\mathbb{X})$ indicates abstraction over exactly one channel name.² Hence, the variable γ can be used like a channel name in constructing a state. The returned value is a channel reference for γ . The existential serves as a modular alternate of the fresh c constraint. So we can invoke *acc* multiple times and obtain different channels from every invocation.

2.2 Channel Abstraction

The discussion of VGR's function type $\Sigma_1; T_1 \rightarrow T_2; \Sigma_2$ 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 identities as outlined in the introduction. Thus, the type of *server* generalizes as shown in (1) so that it can be applied to any channel of type **Chan** α regardless of the name α and the type of *server*, which captures a channel, is shown in (5).

2.3 Data Transmission vs. Channel Transmission

VGR can 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' . Here is the VGR typing rule for the underlying operation:

$$\begin{array}{c}
 \text{C-SENDS} \\
 \hline
 \Gamma; v \mapsto \mathbf{Chan } \beta \quad \Gamma; v' \mapsto \mathbf{Chan } \alpha \\
 \hline
 \Gamma; \Sigma, \alpha : !S'.S, \beta : S'; \text{send } v \text{ on } v' \mapsto \Sigma; \mathbf{Unit}; \alpha : S
 \end{array}$$

The premises are *value typings* that indicate that v and v' are references to fixed channels β and α under variable environment Γ . The

²We defer further discussion of other shapes n and the meaning of $\text{Dom}(n)$ to Sections 2.3.3 and 2.3.4.

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 executing 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-SEND passes ownership of channel β to the receiver.

In addition, VGR implicitly transmits a channel reference which is captured in a closure. To study this phenomenon, we look at VGR's rules for sending and receiving data of type D .

$$\begin{array}{c} \text{C-SEND} \\ \hline \Gamma; v \mapsto D \quad \Gamma; v' \mapsto \mathbf{Chan} \alpha \\ \hline \Gamma; \Sigma, \alpha : !D.S; \text{send } v \text{ on } v' \mapsto \Sigma; \mathbf{Unit}; \alpha : S \\ \\ \text{C-RECEIVED} \\ \hline \Gamma; v \mapsto \mathbf{Chan} \alpha \\ \hline \Gamma; \Sigma, \alpha : ?D.S; \text{receive } v \mapsto \Sigma; D; \alpha : S \end{array}$$

One possibility for type D is a function type like

$$D_1 = \{\beta : S'\}; \mathbf{Unit} \rightarrow \mathbf{Unit}; \{\beta : S''\}.$$

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 :

$$\begin{array}{c} \text{C-RECEIVES} \\ \hline \Gamma; v \mapsto \mathbf{Chan} \alpha \quad \text{fresh } d \\ \hline \Gamma; \Sigma, \alpha : ?S'.S; \text{receive } v \mapsto \Sigma; \mathbf{Chan} d; d : S', \alpha : S \end{array}$$

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 abstracts over states in session types and lifts all restrictions on the type of transmitted values (aka the *payload type*), so that a channel and a function that refers to it can be transmitted at the same time. Here is the revised grammar of session types:

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

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.

The “size” of Σ is gauged with α which determines its domain as indicated in its kind $\text{Dom}(N)$ where N is the shape of the domain. Shapes range over \mathbb{I} (the empty shape), \mathbb{X} (the shape with one binding), and $N_1 \mathbin{\dot{\vee}} N_2$ 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 . In the general pattern $!(\exists \alpha : \text{Dom}(N). \Sigma; T).S$ we find that

- $T = \text{Int}$;
- $\Sigma = \cdot$, 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 $\text{Dom}(\mathbb{X})$ indicates that it is a single channel):

$$\begin{array}{l} \text{send0} = \Lambda(c : \text{Dom}(\mathbb{X})). \Lambda(\sigma : \text{Session}). \\ \quad \lambda(\cdot; x : \text{Int}). \lambda(c \mapsto !(\exists \alpha : \text{Dom}(\mathbb{I}). \cdot; \text{Int}). \sigma; y : \text{Chan } c). \\ \quad \text{send } x \text{ on } y \\ \quad : \forall(c : \text{Dom}(\mathbb{X})). \forall(\sigma : \text{Session}). \\ \quad (\cdot; \text{Int} \rightarrow \cdot; \\ \quad (c \mapsto !(\exists \alpha : \text{Dom}(\mathbb{I}). \cdot; \text{Int}). \sigma; \text{Chan } c \rightarrow c \mapsto \sigma; \text{Unit})) \end{array} \quad (9)$$

2.3.2 One channel. We instantiate the general pattern

$$!(\exists \alpha : \text{Dom}(N). \Sigma; T).S$$

as follows to send a channel of type S' .

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

We omit the term, which is similar to the one in (9), 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: we declare that the channels α and c are different, so that they can be used as keys in the state. The *disjointness constraint* $(\alpha \# c)$ specifies that names in α are disjoint from names in c .

$$\begin{array}{l} \text{send1} : \forall(\alpha : \text{Dom}(\mathbb{X})). \forall(c : \text{Dom}(\mathbb{X})). (\alpha \# c) \Rightarrow \forall(\sigma : \text{Session}). \\ \quad (\cdot; \text{Chan } \alpha \rightarrow \cdot; \\ \quad (\alpha \mapsto S', c \mapsto !(\exists \alpha : \text{Dom}(\mathbb{X}). \alpha \mapsto S'; \text{Chan } \alpha). \sigma; \text{Chan } c \rightarrow \\ \quad c \mapsto \sigma; \text{Unit})) \end{array}$$

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 $!(\exists \alpha : \text{Dom}(N). \Sigma; T).S$ 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., $\text{Dom}(\mathbb{X} \mathbin{\dot{\vee}} \mathbb{X})$;

- 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 $\Sigma = \pi_1 \alpha \mapsto S', \pi_2 \alpha \mapsto S''$;
- to send a pair of channels: $T = \text{Chan}(\pi_1 \alpha) \times \text{Chan}(\pi_2 \alpha)$.

let $S(\sigma) = !(\exists \alpha: \text{Dom}(\mathbb{X} \mathbin{\text{\textcircled{X}}} \mathbb{X}). \pi_1 \alpha \mapsto S', \pi_2 \alpha \mapsto S'';$
 $\text{Chan}(\pi_1 \alpha) \times \text{Chan}(\pi_2 \alpha)). \sigma$ in
 $\forall(\alpha: \text{Dom}(\mathbb{X})). \forall(\beta: \text{Dom}(\mathbb{X})). (\alpha \neq \beta) \Rightarrow$
 $\forall(c: \text{Dom}(\mathbb{X})). (\alpha \neq c, \beta \neq c) \Rightarrow \forall(\sigma: \text{Session}).$
 $(\cdot; \text{Chan } \alpha \times \text{Chan } \beta \rightarrow \cdot;$
 $(\alpha \mapsto S', \beta \mapsto S'', c \mapsto S(\sigma); \text{Chan } c \rightarrow c \mapsto \sigma; \text{Unit}))$

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 $\alpha: \text{Dom}(\mathbb{X})$ and $\beta: \text{Dom}(\mathbb{X})$ and the “uncurried” kind $\text{Dom}(\mathbb{X} \mathbin{\text{\textcircled{X}}} \mathbb{X})$ expected by the existential. To rectify this discrepancy, the term pairs the two domains to obtain some $\gamma = (\alpha, \beta)$ with $\gamma: \text{Dom}(\mathbb{X} \mathbin{\text{\textcircled{X}}} \mathbb{X})$ as needed for the existential. This definition of γ implies that $\alpha = \pi_1 \gamma$ and $\beta = \pi_2 \gamma$ which are needed to obtain the correct state and type for the body of the existential.

2.3.4 The general case. In general a value can refer to an arbitrary number of channels, which should not be fixed a priori. We exhibit and discuss the type of a general send function *gsend* and show how to obtain the previous examples by instantiation.

gsend: $\forall(n: \text{Shape}). \forall(\alpha: \text{Dom}(n)).$
 $\forall(\hat{\Sigma}: \text{Dom}(n) \rightarrow \text{State}). \forall(\hat{T}: \text{Dom}(n) \rightarrow \text{Type}).$
 $\forall(c: \text{Dom}(\mathbb{X})). (\alpha \neq c) \Rightarrow \forall(\sigma: \text{Session}).$
 $(\cdot; \hat{T} \alpha \rightarrow \cdot;$
 $(\hat{\Sigma} \alpha, c \mapsto !(\exists \alpha: \text{Dom}(n). \hat{\Sigma} \alpha; \hat{T} \alpha). \sigma; \text{Chan } c \rightarrow c \mapsto \sigma;$
 $\text{Unit}))$ (10)

We abstract over the shape, n , 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*:

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

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

Figure 1: Syntax of PolyVGR

3 FORMAL SYNTAX AND SEMANTICS OF POLYVGR

3.1 Syntax

Figure 1 defines the syntax of PolyVGR starting with kinds and types. Different metavariables for types indicate 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 ($\text{Dom}(N)$ ranged over by D), and arrow kinds. The kind for domains depends on 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 \bar{S} to indicate the dual of a session type (which flips sending and receiving operations as well as choice and branch). Shapes comprise the empty shape \mathbb{I} , the single-channel shape \mathbb{X} , and the combination of two shapes $_ \# _$. 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. Constraints \mathbb{C} are type environments restricted to bindings of disjointness constraints.

Following VGR [37], the expression language is presented in A-normal form [11], 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, the unit value, pairs of values, lambda abstractions, and type abstractions with constraints — their body is restricted to a syntactic value to avoid unsoundness in the presence of effects.

Configurations C describe 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 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. We start with

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

All judgments depend on context formation, which 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 $[\Gamma]$ and $[\Gamma]$,
- 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

CF-EMPTY $\vdash \cdot$	CF-CONSKIND $\frac{\vdash \Gamma \quad \Gamma \vdash K \quad \alpha \notin \text{dom}(\Gamma)}{\vdash \Gamma, \alpha : K}$
CF-CONSTYPE $\frac{\vdash \Gamma \quad \Gamma \vdash T : \text{Type} \quad x \notin \text{dom}(\Gamma)}{\vdash \Gamma, x : T}$	
CF-CONSCSTR $\frac{\vdash \Gamma \quad \Gamma \vdash D_1 : \text{Dom}(N_1) \quad \Gamma \vdash D_2 : \text{Dom}(N_2)}{\vdash \Gamma, D_1 \# D_2}$	

Figure 2: Context formation ($\vdash \Gamma$)

KF-TYPE $\Gamma \vdash \text{Type}$	KF-SESSION $\Gamma \vdash \text{Session}$	KF-STATE $\Gamma \vdash \text{State}$	KF-SHAPE $\Gamma \vdash \text{Shape}$
KF-DOM $\frac{\Gamma \vdash N : \text{Shape}}{\Gamma \vdash \text{Dom}(N)}$	KF-ARR $\frac{\Gamma \vdash K_1 \quad \Gamma \vdash K_2}{\Gamma \vdash K_1 \rightarrow K_2}$		

Figure 3: Kind formation ($\Gamma \vdash K$)

is wellformed with respect to the current context Γ , which may be needed to construct the shape and then the domain.

Kind formation is in Figure 3. 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 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 8). 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

K-VAR $\Gamma, \alpha : K \vdash \alpha : K$	K-APP $\frac{\Gamma \vdash T_1 : K_1 \rightarrow K_2 \quad \Gamma \vdash T_2 : K_1}{\Gamma \vdash T_1 T_2 : K_2}$	
K-LAM $\frac{\Gamma \vdash N : \text{Shape} \quad [\Gamma], \alpha : \text{Dom}(N) \vdash T : K \quad K \in \{\text{Type}, \text{State}\}}{\Gamma \vdash \lambda(\alpha : \text{Dom}(N)).T : \text{Dom}(N) \rightarrow K}$		
K-ALL $\frac{\vdash \Gamma, \alpha : K, \mathbb{C} \quad \Gamma, \alpha : K, \mathbb{C} \vdash T : \text{Type}}{\Gamma \vdash \forall(\alpha : K). \mathbb{C} \Rightarrow T : \text{Type}}$	K-CHAN $\frac{\Gamma \vdash D : \text{Dom}(\mathbb{X})}{\Gamma \vdash \text{Chan } D : \text{Type}}$	
K-ARR $\frac{\begin{array}{cc} \Gamma_1 \vdash \Sigma_1 : \text{State} & \Gamma_1 \vdash T_1 : \text{Type} \\ \Gamma_1, \# \Gamma_2 \vdash \Sigma_2 : \text{State} & \Gamma_1, \# \Gamma_2 \vdash T_2 : \text{Type} \\ \vdash \Gamma_1, \# \Gamma_2 & \Gamma_2 = [\Gamma_2] \end{array}}{\Gamma_1 \vdash (\Sigma_1; T_1 \rightarrow \exists \Gamma_2. \Sigma_2; T_2) : \text{Type}}$		
		K-ACCESSPOINT $\frac{\Gamma \vdash S : \text{Session}}{\Gamma \vdash [S] : \text{Type}}$
K-UNIT $\Gamma \vdash \text{Unit} : \text{Type}$	K-PAIR $\frac{\Gamma \vdash T_1 : \text{Type} \quad \Gamma \vdash T_2 : \text{Type}}{\Gamma \vdash T_1 \times T_2 : \text{Type}}$	
K-SEND $\frac{\begin{array}{cc} \Gamma \vdash N : \text{Shape} & [\Gamma], \alpha : \text{Dom}(N) \vdash \Sigma : \text{State} \\ [\Gamma], \alpha : \text{Dom}(N) \vdash T : \text{Type} & \Gamma \vdash S : \text{Session} \end{array}}{\Gamma \vdash !(\exists \alpha : \text{Dom}(N). \Sigma; T). S : \text{Session}}$		
K-RECV $\frac{\begin{array}{cc} \Gamma \vdash N : \text{Shape} & [\Gamma], \alpha : \text{Dom}(N) \vdash \Sigma : \text{State} \\ [\Gamma], \alpha : \text{Dom}(N) \vdash T : \text{Type} & \Gamma \vdash S : \text{Session} \end{array}}{\Gamma \vdash ?(\exists \alpha : \text{Dom}(N). \Sigma; T). S : \text{Session}}$		
K-BRANCH $\frac{\Gamma \vdash S_1 : \text{Session} \quad \Gamma \vdash S_2 : \text{Session}}{\Gamma \vdash S_1 \& S_2 : \text{Session}}$	K-DUAL $\frac{\Gamma \vdash S : \text{Session}}{\Gamma \vdash \bar{S} : \text{Session}}$	
K-CHOICE $\frac{\Gamma \vdash S_1 : \text{Session} \quad \Gamma \vdash S_2 : \text{Session}}{\Gamma \vdash S_1 \oplus S_2 : \text{Session}}$	K-END $\Gamma \vdash \text{End} : \text{Session}$	

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

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, and 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

K-SHAPEEMPTY $\Gamma \vdash \mathbb{I} : \text{Shape}$		K-SHAPECHAN $\Gamma \vdash \mathbb{X} : \text{Shape}$
K-SHAPEPAIR $\frac{\Gamma \vdash N_1 : \text{Shape} \quad \Gamma \vdash N_2 : \text{Shape}}{\Gamma \vdash N_1 \circ N_2 : \text{Shape}}$		K-DOMEMPTY $\Gamma \vdash * : \text{Dom}(\mathbb{I})$
K-DOMMERGE $\frac{\Gamma \vdash D_1 : \text{Dom}(N_1) \quad \Gamma \vdash D_2 : \text{Dom}(N_2) \quad \Gamma \vdash D_1 \# D_2}{\Gamma \vdash D_1, D_2 : \text{Dom}(N_1 \circ N_2)}$		
K-DOMPROJ $\frac{\Gamma \vdash D : \text{Dom}(N_1 \circ N_2)}{\Gamma \vdash \pi_\ell D : \text{Dom}(N_\ell)}$		K-STEMPTY $\Gamma \vdash \cdot : \text{State}$
K-STCHAN $\frac{\Gamma \vdash D : \text{Dom}(\mathbb{X}) \quad \Gamma \vdash S : \text{Session}}{\Gamma \vdash D \mapsto S : \text{State}}$		
K-STMERGE $\frac{\Gamma \vdash \Sigma_1 : \text{State} \quad \Gamma \vdash \text{dom}(\Sigma_1) \# \text{dom}(\Sigma_2) \quad \Gamma \vdash \Sigma_2 : \text{State}}{\Gamma \vdash \Sigma_1, \Sigma_2 : \text{State}}$		

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

TC-TAPP $(\lambda(\alpha : \text{Dom}(N)). T_1) T_2 \equiv \{T_2/\alpha\} T_1$		TC-PROJ $\pi_\ell(D_1, D_2) \equiv D_\ell$
TC-DUALEND $\text{End} \equiv \text{End}$		TC-DUALVAR $\bar{\bar{\alpha}} \equiv \alpha$
TC-DUALSEND $!(\exists \alpha : \text{Dom}(N). \Sigma; T). S \equiv ?(\exists \alpha : \text{Dom}(N). \Sigma; T). \bar{S}$		
TC-DUALRECV $?(\exists \alpha : \text{Dom}(N). \Sigma; T). S \equiv !(\exists \alpha : \text{Dom}(N). \Sigma; T). \bar{S}$		
TC-DUALCHOICE $\bar{S}_1 \oplus \bar{S}_2 \equiv \bar{S}_1 \& \bar{S}_2$		TC-DUALBRANCH $\bar{S}_1 \& \bar{S}_2 \equiv \bar{S}_1 \oplus \bar{S}_2$

Figure 6: Type conversion ($T \equiv T$)

the operator cannot be fully eliminated as in $\bar{\alpha}$. 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 send \cdot on \cdot operation and T-TAPP for type application (see Figure 10).

Constraint entailment is defined 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.

$$\begin{array}{c}
\text{CE-AXIOM} \\
\Gamma, D_1 \# D_2 \vdash D_1 \# D_2 \\
\\
\text{CE-SYM} \\
\frac{\Gamma \vdash D_2 \# D_1}{\Gamma \vdash D_1 \# D_2} \\
\\
\text{CE-EMPTY} \\
\Gamma \vdash D \# * \\
\\
\text{CE-SPLIT} \\
\frac{\Gamma \vdash D \# (D_1, D_2)}{\Gamma \vdash D \# D_1 \quad \Gamma \vdash D \# D_2} \\
\\
\text{CE-MERGE} \\
\frac{\Gamma \vdash D \# D_1 \quad \Gamma \vdash D \# D_2}{\Gamma \vdash D \# (D_1, D_2)} \\
\\
\text{CE-PROJMERGE} \\
\frac{\Gamma \vdash D_1 \# \pi_1 D_2 \quad \Gamma \vdash D_1 \# \pi_2 D_2}{\Gamma \vdash D_1 \# D_2} \\
\\
\text{CE-PROJSPLIT} \\
\frac{\Gamma \vdash D_1 \# D_2}{\Gamma \vdash D_1 \# \pi_\ell D_2} \\
\\
\text{CE-EMPTY} \\
\Gamma \vdash \cdot \\
\\
\text{CE-CONS} \\
\frac{\Gamma \vdash \mathbb{C} \quad \Gamma \vdash D_1 \# D_2}{\Gamma \vdash \mathbb{C}, D_1 \# D_2}
\end{array}$$

Figure 7: Constraint entailment ($\Gamma \vdash \mathbb{C}$)

$$\begin{aligned}
&\Gamma_1 \# \Gamma_2 = \Gamma_1, \Gamma_2, \mathbb{C}_2, \mathbb{C}_{12} \text{ where} \\
&\mathbb{C}_2 = \{\alpha_1 \# \alpha_2 \mid \alpha_1, \alpha_2 \in \text{dom}(\lceil \Gamma_2 \rceil), \alpha_1 \neq \alpha_2\} \\
&\mathbb{C}_{12} = \{\alpha_1 \# \alpha_2 \mid \alpha_1 \in \text{dom}(\lceil \Gamma_1 \rceil), \alpha_2 \in \text{dom}(\lceil \Gamma_2 \rceil)\}
\end{aligned}$$

Figure 8: Disjoint context extension ($\Gamma \# \Gamma$)

$$\begin{array}{c}
\text{T-VAR} \quad \Gamma, x : T \vdash x : T \quad \text{T-UNIT} \quad \Gamma \vdash \text{unit} : \text{Unit} \quad \text{T-PAIR} \quad \frac{\Gamma \vdash v_1 : T_1 \quad \Gamma \vdash v_2 : T_2}{\Gamma \vdash (v_1, v_2) : T_1 \times T_2} \\
\\
\text{T-TABS} \quad \frac{\Gamma \vdash \forall(\alpha : K). \mathbb{C} \Rightarrow T : \text{Type} \quad \Gamma, \alpha : K, \mathbb{C} \vdash v : T}{\Gamma \vdash \Lambda(\alpha : K). \mathbb{C} \Rightarrow v : \forall(\alpha : K). \mathbb{C} \Rightarrow T} \\
\\
\text{T-CHAN} \quad \frac{\Gamma \vdash D : \text{Dom}(\mathbb{X})}{\Gamma \vdash \text{chan } D : \text{Chan } D} \quad \text{T-ABS} \quad \frac{\Gamma_1 \vdash (\Sigma_1; T_1 \rightarrow \exists \Gamma_2. \Sigma_2; T_2) : \text{Type} \quad \Gamma_1, x : T_1; \Sigma_1 \vdash e : \exists \Gamma_2. \Sigma_2; T_2}{\Gamma_1 \vdash \lambda(\Sigma_1; x : T_1). e : (\Sigma_1; T_1 \rightarrow \exists \Gamma_2. \Sigma_2; T_2)}
\end{array}$$

Figure 9: Value typing ($\Gamma \vdash v : T$)

The context restriction operators, $\lceil \Gamma \rceil$ and $\lceil \Gamma \rceil$, are a technical device. Both operators keep only bindings of type variables. One removes all domain bindings and the other removes all non-domain bindings.

Figure 8 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

As the syntax of expressions obeys A-normal form, there are three main judgments

- value typing $\Gamma \vdash v : T$,

- expression typing $\Gamma; \Sigma \vdash e : \exists \Gamma. \Sigma; T$, and
- configuration typing $\Gamma; \Sigma \vdash C$.

The rules in Figure 9 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 and invokes expression typing to obtain the return state and type.

Figure 10 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 let $x = e_1$ 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 $\text{Dom}(\mathbb{X})$ (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

$$\begin{array}{c}
\text{T-LET} \\
\frac{\Gamma_1; \Sigma_1 \vdash e_1 : \exists \Gamma_2. \Sigma'_2; T_1 \quad \Gamma_1, \# \Gamma_2, x : T_1; \Sigma_2, \Sigma'_2 \vdash e_2 : \exists \Gamma_3. \Sigma_3; T_2}{\Gamma_1, \# \Gamma_2, x : T_1 \vdash \Sigma_2, \Sigma'_2 : \text{State}} \\
\hline
\Gamma_1; \Sigma_1, \Sigma_2 \vdash \text{let } x = e_1 \text{ in } e_2 : \exists \Gamma_2, \Gamma_3. \Sigma_3; T_2
\\[10pt]
\begin{array}{ccc}
\text{T-VAL} & \text{T-PROJ} & \text{T-NEW} \\
\frac{\Gamma \vdash v : T}{\Gamma; \Sigma \vdash v : \exists \cdot. \Sigma; T} & \frac{\Gamma \vdash v : T_1 \times T_2}{\Gamma; \cdot \vdash \pi_\ell v : \exists \cdot. \cdot; T_\ell} & \frac{\Gamma \vdash S : \text{Session}}{\Gamma; \cdot \vdash \text{new } S : \exists \cdot. \cdot; [S]}
\\[10pt]
\text{T-APP} \\
\frac{\Gamma_1 \vdash v_1 : (\Sigma_1; T_1 \rightarrow \exists \Gamma_2. \Sigma_2; T_2) \quad \Gamma_1 \vdash v_2 : T_1}{\Gamma_1; \Sigma_1 \vdash v_1 v_2 : \exists \Gamma_2. \Sigma_2; T_2}
\\[10pt]
\text{T-TAPP} \\
\frac{\Gamma \vdash v : \forall (\alpha : K). \mathbb{C} \Rightarrow T \quad \Gamma \vdash T' : K \quad \Gamma \vdash \{T'/\alpha\} \mathbb{C} \quad \{T'/\alpha\} T \equiv T''}{\Gamma; \cdot \vdash v[T'] : \exists \cdot. \cdot; T''}
\\[10pt]
\text{T-REQUEST} \\
\frac{\Gamma \vdash v : [S]}{\Gamma; \cdot \vdash \text{request } v : \exists \alpha : \text{Dom}(\mathbb{X}). \alpha \mapsto S; \text{Chan } \alpha}
\\[10pt]
\text{T-ACCEPT} \\
\frac{\Gamma \vdash v : [S]}{\Gamma; \cdot \vdash \text{accept } v : \exists \alpha : \text{Dom}(\mathbb{X}). \alpha \mapsto \bar{S}; \text{Chan } \alpha}
\\[10pt]
\text{T-SEND} \\
\frac{\Gamma \vdash D' : \text{Dom}(N) \quad \{D'/\alpha'\} \Sigma' \equiv \Sigma \quad \{D'/\alpha'\} T' \equiv T}{\Gamma \vdash D : \text{Dom}(\mathbb{X}) \quad \Gamma \vdash v_1 : T \quad \Gamma \vdash v_2 : \text{Chan } D} \\
\hline
\Gamma; \Sigma, D \mapsto !(\exists \alpha' : \text{Dom}(N). \Sigma'; T'). S \vdash \text{send } v_1 \text{ on } v_2 : \exists \cdot. D \mapsto S; \text{Unit}
\\[10pt]
\text{T-RECV} \\
\frac{\Gamma \vdash D : \text{Dom}(\mathbb{X}) \quad \Gamma \vdash v : \text{Chan } D}{\Gamma; D \mapsto ?(\exists \alpha' : \text{Dom}(N). \Sigma'; T'). S \vdash \text{receive } v : \exists (\alpha' : \text{Dom}(N)). \Sigma', D \mapsto S; T'}
\\[10pt]
\begin{array}{cc}
\text{T-FORK} & \text{T-CLOSE} \\
\frac{\Gamma \vdash v : (\Sigma; \text{Unit} \rightarrow \cdot; \text{Unit})}{\Gamma; \Sigma \vdash \text{fork } v : \exists \cdot. \cdot; \text{Unit}} & \frac{\Gamma \vdash v : \text{Chan } D}{\Gamma; D \mapsto \text{End} \vdash \text{close } v : \exists \cdot. \cdot; \text{Unit}}
\\[10pt]
\text{T-SELECT} \\
\frac{\Gamma \vdash v : \text{Chan } D}{\Gamma; D \mapsto S_1 \oplus S_2 \vdash \text{select } \ell \text{ on } v : \exists \cdot. D \mapsto S_\ell; \text{Unit}}
\\[10pt]
\text{T-CASE} \\
\frac{\Gamma_1 \vdash v : \text{Chan } D \quad (\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 \text{case } v \text{ of } \{e_1; e_2\} : \exists \Gamma_2. \Sigma_2; T}
\end{array}
\end{array}$$

Figure 10: Expression typing ($\Gamma; \Sigma \vdash e : \exists \Gamma. \Sigma; T$)

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 .

$$\begin{array}{c}
\text{T-EXP} \quad \frac{\Gamma; \Sigma \vdash e : \exists \Gamma'. \cdot; T}{\Gamma; \Sigma \vdash e} \quad \text{T-PAR} \quad \frac{\Gamma; \Sigma_1 \vdash C_1 \quad \Gamma; \Sigma_2 \vdash C_2}{\Gamma; \Sigma_1, \Sigma_2 \vdash C_1 \parallel C_2}
\\[10pt]
\text{T-NUCHAN} \\
\frac{\alpha, \alpha' \text{ not free in } \Gamma \quad \Gamma \vdash S : \text{Session} \quad \Gamma, \# \alpha : \text{Dom}(\mathbb{X}), \# \alpha' : \text{Dom}(\mathbb{X}); \Sigma, \alpha \mapsto S, \alpha' \mapsto \bar{S} \vdash C}{\Gamma; \Sigma \vdash v \alpha, \alpha' \mapsto S. C}
\\[10pt]
\text{T-NUCHANCLOSED} \\
\frac{\alpha, \alpha' \text{ not free in } \Gamma \quad \Gamma, \# \alpha : \text{Dom}(\mathbb{X}), \# \alpha' : \text{Dom}(\mathbb{X}); \Sigma \vdash C}{\Gamma; \Sigma \vdash v \alpha, \alpha' \mapsto \text{End}. C}
\\[10pt]
\text{T-NUACCESS} \\
\frac{x \text{ not free in } \Gamma \quad \Gamma \vdash S : \text{Session} \quad \Gamma, x : [S]; \Sigma \vdash C}{\Gamma; \Sigma \vdash v x : [S]. C}
\end{array}$$

Figure 11: Configuration typing ($\Gamma; \Sigma \vdash C$)

$$\begin{array}{c}
\text{ER-BETAFUN} \quad (\lambda(\Sigma; x : T). e_1) v_2 \hookrightarrow_e \{v_2/x\} e_1 \quad \text{ER-BETAPAIR} \quad \pi_\ell(v_1, v_2) \hookrightarrow_e v_\ell
\\[10pt]
\text{ER-BETALL} \quad (\Lambda(\alpha : K). \mathbb{C} \Rightarrow v)[T] \hookrightarrow_e \{T/\alpha\} v
\\[10pt]
\text{ER-BETALET} \quad \text{let } x = v_1 \text{ in } e_2 \hookrightarrow_e \{v_1/x\} e_2 \quad \text{ER-LIFT} \quad \frac{e_1 \hookrightarrow_e e_2}{\text{let } x = e_1 \text{ in } e \hookrightarrow_e \text{let } x = e_2 \text{ in } e}
\end{array}$$

Figure 12: Expression reduction ($e \hookrightarrow_e e$)

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

Forking (rule T-FORK) starts a new process from a $\text{Unit} \rightarrow \text{Unit}$ 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 11 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.

$$\begin{array}{c}
\text{CC-NULL} \quad C \parallel \text{unit} \equiv C \quad \text{CC-COMM} \quad C_1 \parallel C_2 \equiv C_2 \parallel C_1 \quad \text{CC-LIFT} \quad \frac{C_1 \equiv C_2}{C[C_1] \equiv C[C_2]} \\
\\
\text{CC-ASSOC} \quad C_1 \parallel (C_2 \parallel C_3) \equiv (C_1 \parallel C_2) \parallel C_3 \quad \text{CC-SWAP} \quad v\alpha, \alpha' \mapsto S. C \equiv v\alpha', \alpha \mapsto \bar{S}. C \\
\\
\text{CC-SCOPE-CHAN} \quad \frac{\alpha, \alpha' \text{ not free in } C_1}{C_1 \parallel (v\alpha, \alpha' \mapsto S. C_2) \equiv v\alpha, \alpha' \mapsto S. (C_1 \parallel C_2)} \\
\\
\text{CC-SCOPE-ACCESS} \quad \frac{x \text{ not free in } C_1}{C_1 \parallel (vx: [S]. C_2) \equiv vx: [S]. (C_1 \parallel C_2)}
\end{array}$$

Figure 13: Configuration congruence ($C \equiv C$)

$$\begin{array}{c}
\text{CR-FORK} \quad C[\mathcal{E}[\text{fork } v]] \hookrightarrow_C C[(v \text{ unit}) \parallel \mathcal{E}[\text{unit}]] \\
\\
\text{CR-NEW} \quad \frac{x \text{ fresh}}{C[\mathcal{E}[\text{new } S]] \hookrightarrow_C C[vx: [S]. \mathcal{E}[x]]} \quad \text{CR-EXPR} \quad \frac{e_1 \hookrightarrow_e e_2}{C[e_1] \hookrightarrow_C C[e_2]} \\
\\
\text{CR-REQUESTACCEPT} \quad \frac{\alpha, \alpha' \text{ fresh} \quad C \equiv C[vx: [S]. (\mathcal{E}_1[\text{request } x] \parallel \mathcal{E}_2[\text{accept } x] \parallel C')]}{C \hookrightarrow_C C[vx: [S]. v\alpha, \alpha' \mapsto S. (\mathcal{E}_1[\text{chan } \alpha] \parallel \mathcal{E}_2[\text{chan } \alpha'] \parallel C')]} \\
\\
\text{CR-SENDRECV} \quad \frac{C \equiv C[v\alpha, \alpha' \mapsto !(\exists \alpha: \text{Dom}(N). \Sigma; T). S. (\mathcal{E}_1[\text{send } v \text{ on chan } \alpha] \parallel \mathcal{E}_2[\text{receive chan } \alpha'] \parallel C')]}{C \hookrightarrow_C C[v\alpha, \alpha' \mapsto S. (\mathcal{E}_1[\text{unit}] \parallel \mathcal{E}_2[v] \parallel C')]} \\
\\
\text{CR-SELECTCASE} \quad \frac{C \equiv C[v\alpha, \alpha' \mapsto S_1 \oplus S_2. (\mathcal{E}_1[\text{select } \ell \text{ on chan } \alpha] \parallel \mathcal{E}_2[\text{case chan } \alpha' \text{ of } \{e_1; e_2\}] \parallel C')]}{C \hookrightarrow_C C[v\alpha, \alpha' \mapsto S_\ell. (\mathcal{E}_1[\text{unit}] \parallel \mathcal{E}_2[e_\ell] \parallel C')]} \\
\\
\text{CR-CLOSE} \quad \frac{C \equiv C[v\alpha, \alpha' \mapsto \text{End}. (\mathcal{E}_1[\text{close chan } \alpha] \parallel \mathcal{E}_2[\text{close chan } \alpha'] \parallel C')]}{C \hookrightarrow_C C[v\alpha, \alpha' \mapsto \text{End}. (\mathcal{E}_1[\text{unit}] \parallel \mathcal{E}_2[\text{unit}] \parallel C')]}
\end{array}$$

Figure 14: Configuration reduction ($C \hookrightarrow_C C$)

3.4 Dynamics

Figure 12 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 13 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 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 14 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 communication between two processes. Our rules 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).

$$\begin{array}{l}
(1) \quad \frac{\vdash \Gamma_1 \quad \Gamma_1 \vdash \Sigma_1 : \text{State} \quad \Gamma_1; \Sigma_1 \vdash e : \exists \Gamma_2. \Sigma_2; T \quad e \hookrightarrow_e e'}{\exists T'. \Gamma_1; \Sigma_1 \vdash e' : \exists \Gamma_2. \Sigma_2; T' \wedge T' \equiv T} \\
(2) \quad \frac{\vdash \Gamma \quad \Gamma \vdash \Sigma : \text{State} \quad \Gamma; \Sigma \vdash C \quad C \hookrightarrow_C C'}{\exists \Sigma'. \Gamma; \Sigma' \vdash C'}
\end{array}$$

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

LEMMA 4.2 (SUBJECT CONGRUENCE).

$$\frac{\vdash \Gamma \quad \Gamma \vdash \Sigma : \text{State} \quad \Gamma; \Sigma \vdash C \quad C \equiv C'}{\Gamma; \Sigma \vdash C'}$$

It is 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 at 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
 - $e = \text{fork } \lambda(\Sigma; x: T).e_1$,
 - $e = \text{new } S$,
 - $e = \text{accept } v$,
 - $e = \text{request } v$,
 - $e = \text{send } v \text{ on chan } D$,
 - $e = \text{receive chan } D$,
 - $e = \text{select } \ell \text{ on chan } D$,
 - $e = \text{case chan } D \text{ of } \{e_1; e_2\}$,
 - $e = \text{close chan } D$, or
 - $e = \text{let } x = e_1 \text{ in } e_2$ where 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 Outer Γ is defined by

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

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).

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

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 Final C is defined inductively by the following cases:

- Final v (an expression process reduced to a value),
- Final $(C_1 \parallel C_2)$ if Final C_1 and Final C_2 ,
- Final $(\nu x : [S]. C_1)$ if Final C_1 , or
- Final $(\nu \alpha, \alpha' \mapsto \text{End}. C_1)$ if 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 Deadlock C holds for a configuration C iff:

- (1) For all configuration contexts C , if $C = C[e]$, then either Value e or Comm e and $e \neq \text{fork } v$ and $e \neq \text{new } S$.
- (2) For all configuration contexts C , if $C = C[\nu x : [S]. C']$, then
 - if $C' = C_1[\mathcal{E}_1[\text{request } x]]$, then there is no C_2, \mathcal{E}_2 such that $C' = C_2[\mathcal{E}_2[\text{accept } x]]$,
 - if $C' = C_1[\mathcal{E}_1[\text{accept } x]]$, then there is no C_2, \mathcal{E}_2 such that $C' = C_2[\mathcal{E}_2[\text{request } x]]$.
- (3) For all configuration contexts C , if $C = C[\nu \alpha_1, \alpha_2 \mapsto S. C']$, then
 - if $C' = C_1[\mathcal{E}_1[\text{send } v \text{ on chan } \alpha_\ell]]$, then there is no C_2, \mathcal{E}_2 such that $C' = C_2[\mathcal{E}_2[\text{receive chan } \alpha_{3-\ell}]]$,
 - if $C' = C_1[\mathcal{E}_1[\text{receive chan } \alpha_\ell]]$, then there is no C_2, \mathcal{E}_2 such that $C' = C_2[\mathcal{E}_2[\text{send } v \text{ on chan } \alpha_{3-\ell}]]$,

- if $C' = C_1[\mathcal{E}_1[\text{select } \ell' \text{ on chan } \alpha_\ell]]$, then there is no C_2, \mathcal{E}_2 such that $C' = C_2[\mathcal{E}_2[\text{case chan } \alpha_{3-\ell} \text{ of } \{e_1; e_2\}]]$,
- if $C' = C_1[\mathcal{E}_1[\text{case chan } \alpha_\ell \text{ of } \{e_1; e_2\}]]$, then there is no C_2, \mathcal{E}_2 such that $C' = C_2[\mathcal{E}_2[\text{select } \ell' \text{ on chan } \alpha_{3-\ell}]]$,
- if $C' = C_1[\mathcal{E}_1[\text{close chan } \alpha_\ell]]$, then there is no C_2, \mathcal{E}_2 such that $C' = C_2[\mathcal{E}_2[\text{close chan } \alpha_{3-\ell}]]$.

LEMMA 4.8 (PROGRESS FOR CONFIGURATIONS).

$$\frac{\vdash \Gamma \quad \text{Outer } \Gamma \quad \Gamma \vdash \Sigma : \text{State} \quad \Gamma; \Sigma \vdash C}{\text{Final } C \vee \text{Deadlock } C \vee \exists C'. C \hookrightarrow_C C'}$$

5 IMPLEMENTATION

We have implemented a type checker and an interpreter for PolyVGR in Haskell. The syntax accepted by the implementation is exactly as presented in this paper, i.e., type annotations are required at lambda abstractions for input type and input state.

The implementation of the type checker requires an algorithmic formulation of the typing. We briefly sketch how to make the declarative typing presented in this paper algorithmic.

- The rules for type conversion give rise to a type normalization function. Type conversion can then be decided by checking alpha-equivalence of normalized types.
- Constraint solving $\Gamma \vdash \mathbb{C}$ is decidable by normalizing and decomposing Γ and \mathbb{C} into closed sets of atomic constraints A_Γ and $A_\mathbb{C}$ and checking $A_\Gamma \supseteq A_\mathbb{C}$. Decomposition is done according to CE-SPLIT and CE-SYM, yielding constraints of form $d_1 \# d_2$ where $d_i = \pi_{\ell_1} \dots \pi_{\ell_{n_i}} \alpha$. Then the closure is taken with respect to CE-PROJMERGE, CE-PROJSPLIT, and CE-SYM.
- The T-LET and T-PAR rules non-deterministically split the input state Σ_1, Σ_2 between the subterms. The implementation threads the entire input state through the first subterm and uses the resulting output state as the input for the second subterm.
- In T-CASE we have to check if the existential parts $\exists \Gamma_i. \Sigma_i; T_i$ of both branch typings are equal. This equality should be up to alpha-renaming and reordering of the variables bound in Γ_1 and Γ_2 and up the reordering of the bindings in Σ_1 and Σ_2 . This equality can be decided by computing a renaming ρ on type variables such that $\rho T_1 = T_2$. If all variables in the domain of ρ are bound in Γ_1 and $\rho \Gamma_1 = \Gamma_2$ and $\rho \Sigma_1 = \Sigma_2$ up to reordering, then both existential packages are equal. A similar approach is used in T-SEND, where the existential package of the session type needs to be matched against the current context and state.

6 EXTENSIONS

Typestate is notoriously difficult to scale up to sum types or, more generally, to algebraic datatypes. In this section, we sketch our approach to add sum types to PolyVGR and offer some insights into the additional problems involved in handling recursive datatypes like lists.

To understand the issues arising with sum types, consider the type **Chan** $\alpha + \text{Chan } \beta$ in the context of state Σ . The situation is clear at run time: we either have a channel described by α or one described by β . But which channel should be described in the state

Σ ? Clearly, $\Sigma = \alpha \mapsto S_1; \beta \mapsto S_2$ does not work because it does not express the mutual exclusiveness of the presence of α and β . In fact, if we matched against a value of type **Chan** $\alpha + \text{Chan } \beta$, we would only consume one of α or β and leave the other channel identity dangling in the outgoing state.

Instead, we propose to add new shapes and domains to the type system along with the sum type. As we will see, the remaining features needed to deal with sum types are already provided for.

Types $T, N, D ::= \dots \mid T + T \mid N + N \mid D + D \mid \varphi_\ell D$
 Expressions $e ::= \dots \mid \text{inj}_\ell v \mid \text{match } v \text{ of } \{x : e; x : e\}$

Sum types come with the usual introduction and elimination forms, a sum shape $N + N$, the domain of a sum shape, and two sum extractors $\varphi_1 D$ and $\varphi_2 D$ pronounced “from”. The domain of a sum shape is a pair of the domains of the two alternatives of the sum. The extractors are only applicable to domains of sum shape and behave like projections as becomes clear from the formation rules (extending Figure 4):

$$\begin{array}{c} \text{K-SUM} \\ \frac{\Gamma \vdash T_1 : \text{Type} \quad \Gamma \vdash T_2 : \text{Type}}{\Gamma \vdash T_1 + T_2 : \text{Type}} \end{array} \quad \begin{array}{c} \text{K-DOMFROM} \\ \frac{\Gamma \vdash D : \text{Dom}(N_1 + N_2)}{\Gamma \vdash \varphi_\ell D : \text{Dom}(N_\ell)} \end{array}$$

$$\begin{array}{c} \text{K-SHAPESUM} \\ \frac{\Gamma \vdash N_1 : \text{Shape} \quad \Gamma \vdash N_2 : \text{Shape}}{\Gamma \vdash N_1 + N_2 : \text{Shape}} \end{array} \quad \begin{array}{c} \text{K-DOMSUm} \\ \frac{(\forall \ell) \Gamma \vdash D_\ell : \text{Dom}(N_\ell)}{\Gamma \vdash D_1 + D_2 : \text{Dom}(N_1 + N_2)} \end{array}$$

The typing of sum introduction and elimination needs to be adapted to account for shapes.

$$\begin{array}{c} \text{T-INJ1} \\ \frac{(\forall \ell) \Gamma \vdash \hat{\Sigma}_\ell : \text{Dom}(N_\ell) \rightarrow \text{State} \quad (\forall \ell) \Gamma \vdash \hat{T}_\ell : \text{Dom}(N_\ell) \rightarrow \text{Type} \quad \Gamma \vdash v : T \quad \Gamma \vdash D : \text{Dom}(N_1) \quad \hat{\Sigma}_1 D \equiv \Sigma \quad \hat{T}_1 D \equiv T}{\Gamma; \Sigma \vdash \text{inj}_1 v : \exists \beta : \text{Dom}(N_2). \hat{\Sigma}_1 D, \hat{\Sigma}_2 \beta; \hat{T}_1 D + \hat{T}_2 \beta} \end{array}$$

In rule T-INJ1, we are given a value $v : T$ along with some Σ that describes the channels contained in v . We assume that the shape of Σ is described by N_1 and corresponding domain D . We further assume that the alternatives of the sum are described by type functions $\hat{\Sigma}_1, \hat{T}_1$ and $\hat{\Sigma}_2, \hat{T}_2$. The point is that the pair labeled 1 describes the real resources in v represented by Σ and the pair labeled 2 describes virtual resources that serve as placeholders to describe the (non-existent) other alternative of the sum. The two conversions determine the connection to the real resources.

Injecting the value into the sum type creates a virtual resource for the non-existing alternative, which is represented by domain β . The real part—labeled 1—continues to refer to the same resources D , so that the sharing semantics of further channel references for those resources is preserved. The virtual part—labeled 2—is never exercised because the run-time value has the form $\text{inj}_1 v$.

The alert reader might wonder why we do not treat $\text{inj}_1 v$ as a value. Indeed, $\text{inj}_1 v$ comes with a reduction to create the virtual resource β , which returns a syntactic value $\text{vinj}_1 v$. We elide the corresponding value typing rule, which is obtained from T-INJ1 by stripping the Σ components and assuming the presence of both

domains in Γ .

T-MATCH

$$\begin{array}{c} \frac{\Gamma \vdash D : \text{Dom}(N_1 + N_2) \quad (\forall \ell) \Gamma \vdash \hat{\Sigma}_\ell : \text{Dom}(N_\ell) \rightarrow \text{State} \quad (\forall \ell) \Gamma \vdash \hat{T}_\ell : \text{Dom}(N_\ell) \rightarrow \text{Type} \quad (\forall \ell) \hat{T}_\ell(\varphi_\ell D) \equiv T_\ell \quad \Gamma \vdash v : T_1 + T_2 \quad (\forall \ell) \Gamma, x_\ell : \hat{T}_\ell(\varphi_\ell D); \Sigma, \hat{\Sigma}_\ell(\varphi_\ell D) \vdash e_\ell : \exists \Gamma'. \Sigma'; T}{\Gamma; \Sigma, \hat{\Sigma}_1(\varphi_1 D), \hat{\Sigma}_2(\varphi_2 D) \vdash \text{match } v \text{ of } \{x_1 : e_1; x_2 : e_2\} : \exists \Gamma'. \Sigma'; T} \end{array}$$

To match on a value v of sum type the elimination rule T-MATCH requires a corresponding domain D of sum shape and we must be able to partition the incoming state according to its two alternatives. (If one of the alternatives carries no channels, then its shape is \mathbb{I} and the corresponding state is empty.) As in the introduction rule, the type and state functions \hat{T}_ℓ and $\hat{\Sigma}_\ell$ describe the partitioning. The match keeps the selected part of the state, which corresponds to the real resources, and drops the other part, which corresponds to the virtual resources.

The same general approach would also work for lists. However, due to the recursion in the list type, we cannot allow sharing between values in the list and outside of it. Essentially, a channel value that is incorporated in a list has to give up its identity, but at the same time the identity has to be remembered so that the channel can be reconnected when extracted from the list.

7 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 [3, 6, 13, 21]. 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.

7.1 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 [14].

Wadler [38] 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 [36]. Caires et al. [7], Pérez et al. [25] consider impredicative quantifiers with session types using the same interpretation.

Dardha et al. [9] extend an encoding of session types into π -types with parametric and bounded polymorphism. Lindley and Morris [22] 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 $K = \text{Type}$, $Y = \circ$, and $Z = \pi$ indicates a variable ranging over session types; choosing $K = \text{Row}$ yields a row variable. Almeida et al. [1] 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.

All practically oriented works [1, 22] rely on an elaborate kind system to distinguish 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.

7.2 Polymorphism and Disjointness

Alias types [33] 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 [27] 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 [28] 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 [8] 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 [2] lifts this idea to a polymorphic calculus where type variables are introduced with disjointness constraints that rule out overlapping instantiations. Xie et al. [39] show that calculi for disjoint polymorphic intersection types are closely related to polymorphic record calculi with symmetric concatenation [16].

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 [23] 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.

7.3 Diverse Topics

Pucella and Tov [26] give an embedding of a session type calculus in Haskell. Their embedding is based on Atkey’s parameterized monads [4], 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 [29] 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 [30, 31], which is closely related to our investigation. 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 [15], 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. [17] 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 [10] 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.

Balzer and Pfenning [5] considers a notion of manifest sharing in session types. Their notion is substantially different from our work. PolyVGR facilitates (local) variables, not constrained by linearity, bound to channel references. Thanks to tpestate, the same reference can refer to a channel in different states at different points in a program. In manifest sharing, there are globally shared channels which always offer the same state. Processes can pick up a shared channel, run an unshared protocol on it, and return it in the same shared state as before.

8 CONCLUSION

We started this work on two premises:

- We believe it is important to map the unexplored part of the design space of session type systems based on tpestate.
- We believe that there are practical advantages 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 may be burdensome for programmers. We are just starting to gather practical experience with our implementation of PolyVGR, so we cannot offer a final verdict at this point.

REFERENCES

- [1] Bernardo Almeida, Andreia Mordido, Peter Thiemann, and Vasco T. Vasconcelos. 2022. Polymorphic lambda calculus with context-free session types. *Information and Computation* 289, Part (2022), 104948. <https://doi.org/10.1016/j.ic.2022.104948>
- [2] 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 (Lecture Notes in Computer Science, Vol. 10201)*, Hongseok Yang (Ed.). Springer, Uppsala, Sweden, 1–28. https://doi.org/10.1007/978-3-662-54434-1_1
- [3] Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniérou, 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/10.1561/25000000031>
- [4] Robert Atkey. 2009. Parameterised Notions of Computation. *J. Funct. Program.* 19, 3-4 (2009), 335–376. <https://doi.org/10.1017/S095679680900728X>
- [5] Stephanie Balzer and Frank Pfenning. 2017. Manifest sharing with session types. *Proc. ACM Program. Lang.* 1, ICFP (2017), 37:1–37:29. <https://doi.org/10.1145/3110281>
- [6] Massimo Bartoletti, Ilaria Castellani, Pierre-Malo Deniérou, 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>
- [7] 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
- [8] 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>
- [9] Ornella 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>
- [10] 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](https://doi.org/10.2168/LMCS-7(2:16)2011)
- [11] 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>
- [12] 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>
- [13] Simon Gay and António Ravara (Eds.). 2017. *Behavioural Types: from Theory to Tools*. River Publishers. <https://doi.org/10.13052/rp-9788793519817>
- [14] Simon J. Gay. 2008. Bounded Polymorphism in Session Types. *Math. Struct. Comput. Sci.* 18, 5 (2008), 895–930. <https://doi.org/10.1017/S0960129508006944>
- [15] 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>
- [16] 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>
- [17] Jonas Kastberg Hinrichsen, Daniël Louwink, 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>
- [18] 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
- [19] 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
- [20] 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
- [21] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniérou, 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>
- [22] 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>
- [23] 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>
- [24] Luca Padovani. 2017. A Simple Library Implementation of Binary Sessions. *J. Funct. Program.* 27 (2017), e4. <https://doi.org/10.1017/S0956796816000289>
- [25] 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>
- [26] 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>
- [27] 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>
- [28] 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>
- [29] 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>
- [30] 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 Ornella Dardha (Eds.). Springer, 61–79. https://doi.org/10.1007/978-3-030-78142-2_4
- [31] Hannes Saffrich and Peter Thiemann. 2022. Relating Functional and Imperative Session Types. *Log. Methods Comput. Sci.* 18, 3 (2022). [https://doi.org/10.46298/lmcs-18\(3:33\)2022](https://doi.org/10.46298/lmcs-18(3:33)2022)
- [32] 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>
- [33] 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
- [34] 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>
- [35] 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
- [36] David N. Turner. 1996. *The polymorphic Pi-calculus: theory and implementation*. Ph. D. Dissertation. University of Edinburgh, UK. <http://hdl.handle.net/1842/395>
- [37] 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>
- [38] Philip Wadler. 2014. Propositions as Sessions. *JFP* 24, 2-3 (2014), 384–418. <http://dx.doi.org/10.1017/S095679681400001X>
- [39] 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 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.

$$\text{sendSend} : \Sigma_1; \text{Chan } u \rightarrow (\Sigma_1; \text{Chan } v \rightarrow \text{Unit}; \Sigma_2); \Sigma_1 \quad (11)$$

with $\Sigma_1 = \{u : ! \text{Int}.S_u, v : ! \text{Int}.S_v\}$ 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 $! \text{Int}.! \text{Int}.$ **End** for u and v . However, `sendSend w w` does not type check with the type in (11) 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:

$$\begin{aligned} & \forall (\alpha : \text{Dom}(\mathbb{X})). \forall (\beta : \text{Dom}(\mathbb{X})). (\alpha \neq \beta) \Rightarrow \forall (\sigma_1 : \text{Session}). \forall (\sigma_2 : \text{Session}). \\ & (\cdot; \text{Chan } \alpha \rightarrow \cdot; \left(\begin{array}{l} \alpha \mapsto !(\exists \alpha : \text{Dom}(\mathbb{I}). \cdot; \text{Int}).\sigma_1, \\ \beta \mapsto !(\exists \alpha : \text{Dom}(\mathbb{I}). \cdot; \text{Int}).\sigma_2 \end{array} \right); \text{Chan } \beta \rightarrow \left\{ \begin{array}{l} \alpha \mapsto \sigma_1, \\ \beta \mapsto \sigma_2 \end{array} \right\}; \text{Unit})) \end{aligned}$$

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 $\text{sendSend} : \Sigma_1; \text{Chan } w \rightarrow (\Sigma_1; \text{Chan } w \rightarrow \text{Unit}; \Sigma_2); \Sigma_1$ with $\Sigma_1 = \{w : ! \text{Int}.! \text{Int}.S_w\}$ 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:

$$\begin{aligned} & \forall (\alpha : \text{Dom}(\mathbb{X})). \forall (\sigma : \text{Session}). \\ & (\cdot; \text{Chan } \alpha \rightarrow \cdot; (\alpha \mapsto !(\exists \alpha : \text{Dom}(\mathbb{I}). \cdot; \text{Int}).!(\exists \alpha : \text{Dom}(\mathbb{I}). \cdot; \text{Int}).\sigma; \text{Chan } \alpha \rightarrow \alpha \mapsto \sigma; \text{Unit})) \end{aligned}$$

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

A.2 Higher-Order Functions

Section 2 has 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.

$$\begin{aligned} & \forall (n : \text{Shape}). \forall (\hat{\Sigma} : \text{Dom}(n) \rightarrow \text{State}). \forall (\hat{T} : \text{Dom}(n) \rightarrow \text{Type}). \\ & \forall (\gamma : \text{Dom}(\mathbb{X})). \forall (\sigma : \text{Session}). \forall (\sigma' : \text{Session}). \\ & (\cdot; (\gamma \mapsto \sigma; \text{Chan } \gamma \rightarrow \exists \alpha : \text{Dom}(n). \hat{\Sigma} \alpha, \gamma \mapsto \sigma'; \hat{T} \alpha) \rightarrow \\ & \quad \cdot; (\gamma \mapsto !(\exists \alpha : \text{Dom}(\mathbb{I}). \cdot; \text{Int}).\sigma; \text{Chan } \gamma \rightarrow \exists \alpha : \text{Dom}(n). \hat{\Sigma} \alpha, \gamma \mapsto \sigma'; \hat{T} \alpha)) \end{aligned} \quad (12)$$

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 $\forall (\hat{T} : \text{Dom}(\mathbb{I}) \rightarrow \text{Type}). \dots$ 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 (12) 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.

Removing bindings, which might contain free domain variables

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

Removing non-domain bindings

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

Figure 15: Context restriction ($[\Gamma]$ and $[\Gamma']$)

$$\begin{aligned} & \forall (n' : \text{Shape}). \forall (\hat{\Sigma}' : \text{Dom}(n') \rightarrow \text{State}). \\ & \forall (n'' : \text{Shape}). \forall (\hat{\Sigma}'' : \text{Dom}(n'') \rightarrow \text{State}). \forall (\hat{T}'' : \text{Dom}(n'') \rightarrow \text{Type}). \\ & \forall (\alpha' : \text{Dom}(n')). \forall (\alpha'' : \text{Dom}(n'')). \forall (\gamma : \text{Dom}(\mathbb{X})). (\alpha' \# \gamma, \alpha'' \# \gamma) \Rightarrow \\ & \forall (\sigma : \text{Session}). \forall (\sigma' : \text{Session}). \\ & (\cdot ; (\hat{\Sigma}' \alpha', \gamma \mapsto \sigma; \text{Chan } \gamma \rightarrow \hat{\Sigma}'' \alpha'', \gamma \mapsto \sigma'; \hat{T}'' \alpha'') \rightarrow \\ & \quad \cdot ; (\hat{\Sigma}' \alpha', \gamma \mapsto !(\exists \alpha : \text{Dom}(\mathbb{I}). \cdot ; \text{Int}). \sigma; \text{Chan } \gamma \rightarrow \hat{\Sigma}'' \alpha'', \gamma \mapsto \sigma'; \hat{T}'' \alpha'')) \end{aligned} \quad (13)$$

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.

$$\begin{aligned} & \forall (n' : \text{Shape}). \forall (\hat{\Sigma}' : \text{Dom}(n') \rightarrow \text{State}). \\ & \forall (n'' : \text{Shape}). \\ & \forall (n : \text{Shape}). \forall (\hat{\Sigma}'' : \text{Dom}(n \# n'') \rightarrow \text{State}). \forall (\hat{T}'' : \text{Dom}(n \# n'') \rightarrow \text{Type}). \\ & \forall (\alpha' : \text{Dom}(n')). \forall (\alpha'' : \text{Dom}(n'')). \forall (\gamma : \text{Dom}(\mathbb{X})). (\alpha' \# \gamma, \alpha'' \# \gamma) \Rightarrow \\ & \forall (\sigma : \text{Session}). \forall (\sigma' : \text{Session}). \\ & (\cdot ; (\hat{\Sigma}' \alpha', \gamma \mapsto \sigma; \text{Chan } \gamma \rightarrow \exists \alpha : \text{Dom}(n). \hat{\Sigma}'' (\alpha, \alpha''), \gamma \mapsto \sigma'; \hat{T}'' (\alpha, \alpha'')) \rightarrow \\ & \quad \cdot ; (\hat{\Sigma}' \alpha', \gamma \mapsto !(\exists \alpha : \text{Dom}(\mathbb{I}). \cdot ; \text{Int}). \sigma; \text{Chan } \gamma \rightarrow \exists \alpha : \text{Dom}(n). \hat{\Sigma}'' (\alpha, \alpha''), \gamma \mapsto \sigma'; \hat{T}'' (\alpha, \alpha''))) \end{aligned} \quad (14)$$

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.

A.3 Context Restriction Operators

Figure 15 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.

A.4 Metatheory

In this formalization we use inference rule notation to state lemmas and use proved lemmas in proof trees. While this notation is unconventional, we found that it significantly improves readability.

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

$$\begin{array}{lll} (1) \frac{\Gamma \vdash T : \text{Shape}}{[\Gamma] \vdash T : \text{Shape}} & (2) \frac{\Gamma \vdash K}{[\Gamma] \vdash K} & (3) \frac{\vdash \Gamma}{\vdash [\Gamma]} \end{array}$$

PROOF.

- (1) Straightforward induction on the derivations with kind Shape. The case of type application is not possible, since type lambdas cannot have codomain Shape.
- (2) Straightforward induction on the kind formation using (1) in the case KF-DOM.
- (3) Straightforward induction on $\vdash \Gamma$. 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). \square

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

$$\frac{\Gamma_1 \xRightarrow{\text{OPE}} \Gamma_2}{\lfloor \Gamma_1 \rfloor \xRightarrow{\text{OPE}} \lfloor \Gamma_2 \rfloor}$$

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 $\lfloor \Gamma_1 \rfloor$. \square

LEMMA A.4 (CONTEXT EXTENSION PRESERVES OPE).

$$(1) \frac{\Gamma_1 \xRightarrow{\text{OPE}} \Gamma_2 \quad \vdash \Gamma_1, \Gamma_3 \quad \vdash \Gamma_2, \Gamma_3}{(\Gamma_1, \Gamma_3) \xRightarrow{\text{OPE}} (\Gamma_2, \Gamma_3)} \quad (2) \frac{\Gamma_1 \xRightarrow{\text{OPE}} \Gamma_2 \quad \vdash \Gamma_1, \# \Gamma_3 \quad \vdash \Gamma_2, \# \Gamma_3}{(\Gamma_1, \# \Gamma_3) \xRightarrow{\text{OPE}} (\Gamma_2, \# \Gamma_3)}$$

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 8. Any binding from Γ_1 is also contained in Γ_2 , due to Axiom (3) of $\Gamma_1 \xRightarrow{\text{OPE}} \Gamma_2$. Hence, it also holds that $\text{dom}(\Gamma_1) \subseteq \text{dom}(\Gamma_2)$, which implies $\mathbb{C}_{13} \subseteq \mathbb{C}_{23}$. \square

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 $\text{dom}(\Gamma_1) \cup \text{dom}(\Gamma_2)$, and are hence well-kinded. \square

LEMMA A.6 (WEAKENING). *If $\Gamma_1 \xRightarrow{\text{OPE}} \Gamma_2$ then*

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

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

PROOF. By mutual induction on the derivation to be weakened:

- (1) • *Case CE-AXIOM.* Direct consequence of Axiom (3) of $\Gamma_1 \xRightarrow{\text{OPE}} \Gamma_2$.
• All other cases are immediate by the induction hypotheses.
- (2) By induction on Γ_3 :
• *Case $\Gamma_3 = \cdot$.* Immediate from Axiom (2) of $\Gamma_1 \xRightarrow{\text{OPE}} \Gamma_2$.

- Case $\Gamma_3 = \Gamma'_3, \alpha : K$. In this case we have

$$\frac{\vdash \Gamma_1, \Gamma'_3 \quad \Gamma_1, \Gamma'_3 \vdash \alpha : K}{\vdash \Gamma_1, \Gamma'_3, \alpha : K} \text{CF-CONSKIND}$$

and need to show

$$\frac{\text{(A)} \frac{\vdash \Gamma_2, \Gamma'_3}{\vdash \Gamma_2, \Gamma'_3} \quad \text{(B)} \frac{\Gamma_2, \Gamma'_3 \vdash \alpha : K}{\vdash \Gamma_2, \Gamma'_3, \alpha : K}}{\vdash \Gamma_2, \Gamma'_3, \alpha : K} \text{CF-CONSKIND}$$

(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 $\Gamma_1 \Rightarrow \Gamma_2$.
 • *Case K-LAM, K-SEND, K-RECV*. Here we have assumptions using context restriction, like $[\Gamma_1], \alpha : \text{Dom}(N) \vdash \Sigma : \text{State}$. In order to apply the induction hypothesis on those assumptions, we rely on Lemma A.3, which given $\Gamma_1 \xRightarrow{\text{OPE}} \Gamma_2$ yields $[\Gamma_1] \xRightarrow{\text{OPE}} [\Gamma_2]$.
 • *Case K-ARR*. Here we have assumptions using disjoint context concatenation, like $\Gamma_1, \# \Gamma'_2 \vdash \Sigma_2 : \text{State}$. In order to apply the induction hypothesis on those assumptions, we rely on Lemma A.4.(2), which given $\Gamma_1 \xRightarrow{\text{OPE}} \Gamma_2$ yields $(\Gamma_1, \# \Gamma'_2) \xRightarrow{\text{OPE}} (\Gamma_2, \# \Gamma'_2)$.
 • 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. \square

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 $\vdash \text{id} : \Gamma \Rightarrow \Gamma$.*

PROOF. (1) and (2) follow by assumption.

(3), (4), and (5) follow via T-VAR, K-VAR, and CE-AXIOM, respectively. \square

LEMMA A.9 (EXTENDING SUBSTITUTION TYPINGS).

- (1)
$$\frac{\vdash \sigma : \Gamma_1 \Rightarrow \Gamma_2 \quad \Gamma_1 \vdash T : \text{Type} \quad \Gamma_2 \vdash v : \sigma T \quad x \notin \text{dom}(\Gamma_1)}{\vdash (\sigma, x \mapsto v) : (\Gamma_1, x : T) \Rightarrow \Gamma_2}$$
- (2)
$$\frac{\vdash \sigma : \Gamma_1 \Rightarrow \Gamma_2 \quad \Gamma_1 \vdash K \quad \Gamma_2 \vdash T : \sigma K \quad \alpha \notin \text{dom}(\Gamma_1)}{\vdash (\sigma, \alpha \mapsto T) : (\Gamma_1, \alpha : K) \Rightarrow \Gamma_2}$$
- (3)
$$\frac{\vdash \sigma : \Gamma_1 \Rightarrow \Gamma_2 \quad \vdash \Gamma_1, \mathbb{C} \quad \Gamma_2 \vdash \sigma \mathbb{C}}{\vdash \sigma : (\Gamma_1, \mathbb{C}) \Rightarrow \Gamma_2}$$

PROOF. Straightforward case analysis and rule applications. \square

LEMMA A.10 (WEAKENING SUBSTITUTION TYPINGS).

$$\frac{\vdash \sigma : \Gamma_1 \Rightarrow \Gamma_2 \quad \vdash \Gamma_2, \Gamma'_2}{\vdash \sigma : \Gamma_1 \Rightarrow \Gamma_2, \Gamma'_2}$$

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. \square

LEMMA A.11 (LIFTING SUBSTITUTION TYPINGS).

- (1)
$$\frac{\vdash \sigma : \Gamma_1 \Rightarrow \Gamma_2 \quad \Gamma_1 \vdash T : \text{Type} \quad \Gamma_2 \vdash \sigma T : \text{Type} \quad x \notin \text{dom}(\Gamma_1) \quad x \notin \text{dom}(\Gamma_2)}{\vdash (\sigma, x \mapsto x) : (\Gamma_1, x : T) \Rightarrow (\Gamma_2, x : \sigma T)}$$
- (2)
$$\frac{\vdash \sigma : \Gamma_1 \Rightarrow \Gamma_2 \quad \Gamma_1 \vdash K \quad \Gamma_2 \vdash \sigma K \quad \alpha \notin \text{dom}(\Gamma_1) \quad \alpha \notin \text{dom}(\Gamma_2)}{\vdash (\sigma, \alpha \mapsto \alpha) : (\Gamma_1, \alpha : K) \Rightarrow (\Gamma_2, \alpha : \sigma K)}$$
- (3)
$$\frac{\vdash \sigma : \Gamma_1 \Rightarrow \Gamma_2 \quad \vdash \Gamma_1, \mathbb{C} \quad \vdash \Gamma_2, \sigma \mathbb{C}}{\vdash \sigma : (\Gamma_1, \mathbb{C}) \Rightarrow (\Gamma_2, \mathbb{C})}$$
- (4)
$$\frac{\vdash \sigma : \Gamma_1 \Rightarrow \Gamma_2 \quad \vdash \Gamma_1, \Gamma \quad \vdash \Gamma_2, \sigma \Gamma}{\vdash (\sigma, id) : (\Gamma_1, \Gamma) \Rightarrow (\Gamma_2, \sigma \Gamma)}$$
- (5)
$$\frac{\vdash \sigma : \Gamma_1 \Rightarrow \Gamma_2 \quad \vdash \Gamma_1, \# \Gamma \quad \vdash \Gamma_2, \# \sigma \Gamma}{\vdash (\sigma, id) : (\Gamma_1, \# \Gamma) \Rightarrow (\Gamma_2, \# \sigma \Gamma)}$$

PROOF.

- (1) First, we weaken the substitution typing

$$\frac{\vdash \sigma : \Gamma_1 \Rightarrow \Gamma_2 \quad \frac{\vdash \Gamma_2 \quad \Gamma_2 \vdash \sigma T : \text{Type} \quad x \notin \text{dom}(\Gamma_2)}{\vdash \Gamma_2, x : \sigma T} \text{CF-CONSTYPE}}{\vdash \sigma : \Gamma_1 \Rightarrow \Gamma_2, x : \sigma T} \text{LEMMA A.10}$$

Then we extend the weakened substitution typing

$$\frac{\vdash \sigma : \Gamma_1 \Rightarrow \Gamma_2, x : \sigma T \quad \Gamma_1 \vdash T : \text{Type} \quad \frac{\Gamma_2, x : \sigma T \vdash x : \sigma T}{x \notin \text{dom}(\Gamma_1)} \text{T-VAR}}{\vdash \sigma, x \mapsto x : \Gamma_1, x : T \Rightarrow \Gamma_2, x : \sigma T} \text{LEMMA A.9}$$

- (2) Same as (1).
- (3) First, we weaken the substitution typing

$$\frac{\vdash \sigma : \Gamma_1 \Rightarrow \Gamma_2 \quad \vdash \Gamma_2, \sigma \mathbb{C}}{\vdash \sigma : \Gamma_1 \Rightarrow \Gamma_2, \sigma \mathbb{C}} \text{LEMMA A.10}$$

Then we extend the weakened substitution typing

$$\frac{\vdash \sigma : \Gamma_1 \Rightarrow \Gamma_2, \sigma\mathbb{C} \quad \vdash \Gamma_1, \mathbb{C} \quad \Gamma_2, \sigma\mathbb{C} \vdash \sigma\mathbb{C}}{\vdash \sigma, x \mapsto x : \Gamma_1, \mathbb{C} \Rightarrow \Gamma_2, \sigma\mathbb{C}} \text{ LEMMA A.9}$$

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

$$\begin{aligned} \Gamma_1, \# \Gamma &= \Gamma_1, \Gamma, \mathbb{C}_1, \mathbb{C}'_1 \\ \Gamma_2, \# \sigma\Gamma &= \Gamma_2, \sigma\Gamma, \mathbb{C}_2, \mathbb{C}'_2 \end{aligned}$$

with

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

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 $\text{dom}(\lceil \sigma\Gamma \rceil) = \text{dom}(\lceil \Gamma \rceil)$. 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. \square

LEMMA A.12 (CONTEXT RESTRICTION PRESERVES SUBSTITUTION TYPING).

$$\frac{\vdash \sigma : \Gamma_1 \Rightarrow \Gamma_2}{\vdash \sigma : \lceil \Gamma_1 \rceil \Rightarrow \lceil \Gamma_2 \rceil}$$

PROOF.

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

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

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

LEMMA A.13 (SUBSTITUTION PRESERVES DERIVATIONS). *If $\vdash \sigma : \Gamma_1 \Rightarrow \Gamma_2$, then*

$$\begin{aligned} (1) \quad & \frac{\Gamma_1 \vdash \mathbb{C}}{\Gamma_2 \vdash \sigma\mathbb{C}} & (2) \quad & \frac{\vdash \Gamma_1, \Gamma_3}{\vdash \Gamma_2, \sigma\Gamma_3} & (3) \quad & \frac{\vdash \Gamma_1, \# \Gamma_3}{\vdash \Gamma_2, \# \sigma\Gamma_3} & (4) \quad & \frac{\Gamma_1 \vdash K}{\Gamma_2 \vdash \sigma K} & (5) \quad & \frac{\Gamma_1 \vdash T : K}{\Gamma_2 \vdash \sigma T : \sigma K} \\ (6) \quad & \frac{\Gamma_1 \vdash v : T}{\Gamma_2 \vdash \sigma v : \sigma T} & (7) \quad & \frac{\Gamma_1 \vdash \Sigma_1 : \text{State} \quad \Gamma_1; \Sigma_1 \vdash e : \exists \Gamma_3. \Sigma_2; T}{\Gamma_2; \sigma\Sigma_1 \vdash \sigma e : \exists \sigma\Gamma_3. \sigma\Sigma_2; \sigma T} & (8) \quad & \frac{T_1 \equiv T_2}{\sigma T_1 \equiv \sigma T_2} \end{aligned}$$

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

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

which yields

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

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

$$\frac{\vdash \sigma : \Gamma_1 \Rightarrow \Gamma_2}{\vdash \sigma : [\Gamma_1] \Rightarrow [\Gamma_2]} \text{LEMMA A.12} \quad \frac{\frac{\Gamma_1 \vdash N : \text{Shape}}{\Gamma_1 \vdash \text{Dom}(N)} \text{KF-DOM}}{[\Gamma_1] \vdash \text{Dom}(N)} \text{LEMMA A.1} \quad \frac{\frac{\Gamma_2 \vdash \sigma N : \text{Shape}}{\Gamma_2 \vdash \text{Dom}(\sigma N)} \text{KF-DOM}}{[\Gamma_2] \vdash \text{Dom}(\sigma N)} \text{LEMMA A.1}$$

and then lift the substitution typing over the new domain binding

$$\frac{\vdash \sigma : [\Gamma_1] \Rightarrow [\Gamma_2] \quad [\Gamma_1] \vdash \text{Dom}(N) \quad [\Gamma_2] \vdash \text{Dom}(\sigma N) \quad \alpha \notin \text{dom}(\Gamma_1), \text{dom}(\Gamma_2)}{\vdash \sigma : [\Gamma_1], \alpha : \text{Dom}(N) \Rightarrow [\Gamma_2], \alpha : \text{Dom}(\sigma N)} \text{LEMMA A.11}$$

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

$$\frac{\vdash \sigma : \Gamma_1 \Rightarrow \Gamma_2 \quad \vdash \Gamma_1, \alpha : K, \mathbb{C} \quad \vdash \Gamma_2, \alpha : \sigma K, \sigma \mathbb{C} \quad \alpha \notin \text{dom}(\Gamma_1), \text{dom}(\Gamma_2)}{\vdash \sigma : \Gamma_1, \alpha : K, \mathbb{C} \Rightarrow \Gamma_2, \alpha : \sigma K, \sigma \mathbb{C}} \text{LEMMA A.11}$$

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

$$\Gamma_1 \vdash \Sigma_1 : \text{State} \quad \Gamma_1 \vdash T_1 : \text{Type} \quad \vdash \Gamma_1, \# \Gamma$$

we directly apply the induction hypothesis and obtain

$$\Gamma_2 \vdash \sigma \Sigma_1 : \text{State} \quad \Gamma_2 \vdash \sigma T_1 : \text{Type} \quad \vdash \Gamma_2, \# \sigma \Gamma$$

For the premises

$$\Gamma_1, \# \Gamma \vdash \Sigma_2 : \text{State} \quad \Gamma_1, \# \Gamma \vdash T_2 : \text{Type}$$

we first lift the substitution typing

$$\frac{\vdash \sigma : \Gamma_1 \Rightarrow \Gamma_2 \quad \vdash \Gamma_1, \# \Gamma \quad \vdash \Gamma_2, \# \sigma \Gamma}{\vdash \sigma : \Gamma_1, \# \Gamma \Rightarrow \Gamma_2, \# \sigma \Gamma} \text{LEMMA A.11.5}$$

and then apply the induction hypothesis to obtain

$$\Gamma_2, \# \sigma \Gamma \vdash \sigma \Sigma_2 : \text{State} \quad \Gamma_2, \# \sigma \Gamma \vdash \sigma T_2 : \text{Type}$$

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 $\Gamma_1 \vdash \Sigma_1, D \mapsto S_1 \ \& \ S_2 : \text{State}$. In order to apply the induction hypothesis to the branch expressions, we need to prove

$$\Gamma_1 \vdash \Sigma_1, D \mapsto S_1 : \text{State}$$

$$\Gamma_1 \vdash \Sigma_1, D \mapsto S_2 : \text{State}$$

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*

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

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

LEMMA A.15 (EVALUATION CONTEXT TYPINGS FOR EXPRESSIONS).

$$(1) \frac{\Gamma_1; \Sigma_1 \vdash \mathcal{E}[e] : \exists \Gamma_3. \Sigma_3; T}{\frac{\exists \Sigma_{11}, \Sigma_{12}, \Gamma_{21}, \Gamma_{22}, \Sigma_2, T' \quad \Sigma_1 = \Sigma_{11}, \Sigma_{12} \quad \Gamma_3 = \Gamma_{21}, \Gamma_{22}}{\Gamma_1; \Sigma_{11} \vdash e : \exists \Gamma_{21}. \Sigma_2; T'} \quad \Gamma_1, \# \Gamma_{21}, x : T'; \Sigma_{12}, \Sigma_2 \vdash \mathcal{E}[x] : \exists \Gamma_{22}. \Sigma_3; T}}{\Gamma_1; \Sigma_{11} \vdash e : \exists \Gamma_{21}. \Sigma_2; T'} \quad \Gamma_1, \# \Gamma_{21}, x : T'; \Sigma_{12}, \Sigma_2 \vdash \mathcal{E}[x] : \exists \Gamma_{22}. \Sigma_3; T$$

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

PROOF.

- (1) By induction on the evaluation context \mathcal{E} :

- *Case \square* . 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

$$\Sigma_1 = \Sigma_1 \tag{1}$$

$$\Gamma_3 = \Gamma_3 \tag{2}$$

$$\Gamma_1; \Sigma_1 \vdash e : \exists \Gamma_3. \Sigma_3; T \tag{3}$$

$$\Gamma_1, \# \Gamma_3, x : T; \Sigma_3 \vdash x : \exists \cdot. \Sigma_3; T \tag{4}$$

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

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

$$\frac{\text{T-LET} \quad \Gamma_1; \Sigma_{11} \vdash \mathcal{E}[e] : \exists \Gamma_{21}. \Sigma_2; T' \quad \Gamma_1, \# \Gamma_{21}, x : T'; \Sigma_{12}, \Sigma_2 \vdash e' : \exists \Gamma_{22}. \Sigma_3; T \quad \Gamma_1, \# \Gamma_{21}, x : T' \vdash \Sigma_{12}, \Sigma_2 : \text{State}}{\Gamma_1; \Sigma_{11}, \Sigma_{12} \vdash \text{let } x = \mathcal{E}[e] \text{ in } e' : \exists \Gamma_{21}, \Gamma_{22}. \Sigma_3; T}$$

From the induction hypothesis follows

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

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

$$\Sigma_{11}, \Sigma_{12} = \Sigma_{111}, \Sigma_{112}, \Sigma_{12} \quad (1)$$

$$\Gamma_{21}, \Gamma_{22} = \Gamma_{211}, \Gamma_{212}, \Gamma_{22} \quad (2)$$

$$\Gamma_1; \Sigma_{111} \vdash e : \exists \Gamma_{211}. \Sigma'_2; T'' \quad (3)$$

$$\Gamma_1, \# \Gamma_{211}, y : T'; \Sigma_{112}, \Sigma_{12}, \Sigma'_2 \vdash \text{let } x = \mathcal{E}[e] \text{ in } y : \exists \Gamma_{212}, \Gamma_{22}. \Sigma_3; T \quad (4)$$

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

$$\begin{array}{c} \Gamma_1, \# \Gamma_{211}, y : T''; \Sigma_{112}, \Sigma'_2 \vdash \mathcal{E}[y] : \exists \Gamma_{212}. \Sigma_2; T' \quad \Gamma_1, \# \Gamma_{211}, y : T'', x : T'; \Sigma_{12}, \Sigma_2 \vdash e' : \exists \Gamma_{22}. \Sigma_3; T \\ \Gamma_1, \# \Gamma_{211}, y : T'', x : T' \vdash \Sigma_{12}, \Sigma_2 : \text{State} \\ \hline \text{T-LET} \quad \Gamma_1, \# \Gamma_{211}, y : T''; \Sigma_{112}, \Sigma_{12}, \Sigma'_2 \vdash \text{let } x = \mathcal{E}[e] \text{ in } y : \exists \Gamma_{212}, \Gamma_{22}. \Sigma_3; T \end{array}$$

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). \square

LEMMA A.16 (EVALUATION CONTEXT TYPINGS FOR CONFIGURATIONS).

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

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

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

where $\Gamma_\alpha = \Gamma, \# \alpha : \text{Dom}(\mathbb{X}), \# \alpha' : \text{Dom}(\mathbb{X})$ and $\Sigma_\alpha = \Sigma, \alpha \mapsto S, \alpha' \mapsto \bar{S}$.

From the induction hypothesis follows

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

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* $C = \nu x : [S]. C$. Similar as the previous case.
- *Case* $C = C \parallel C$. Similar as the previous case. \square

LEMMA A.17 (WELLFORMED INPUTS IMPLY WELLFORMED OUTPUTS).

$$\begin{array}{l} (1) \quad \frac{\vdash \Gamma \quad \Gamma \vdash T : K}{\Gamma \vdash K} \\ (2) \quad \frac{\vdash \Gamma \quad \Gamma \vdash v : T}{\Gamma \vdash T : \text{Type}} \\ (3) \quad \frac{\vdash \Gamma \quad \Gamma \vdash \Sigma_1 : \text{State} \quad \Gamma; \Sigma_1 \vdash e : \exists \Gamma'. \Sigma_2; T_2}{\vdash \Gamma, \# \Gamma' \quad \Gamma, \# \Gamma' \vdash \Sigma_2 : \text{State} \quad \Gamma, \# \Gamma' \vdash T_2 : \text{Type}} \end{array}$$

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 $\Gamma \vdash T_1 : K_1 \rightarrow K_2$ yields $\Gamma \vdash K_1 \rightarrow K_2$ which by case-analysis yields $\Gamma \vdash K_2$.
- *Case* K-LAM. From $\Gamma \vdash N : \text{Shape}$ follows $\Gamma \vdash \text{Dom}(N)$. From $K \in \{\text{Type}, \text{State}\}$ follows $\Gamma \vdash K$. Via KF-ARR follows $\Gamma \vdash \text{Dom}(N) \rightarrow K$.
- *Case* K-DOMMERGE. Applying the induction hypothesis to $\Gamma \vdash D_i : \text{Dom}(N_i)$ yields $\Gamma \vdash \text{Dom}(N_i)$, which by case analysis on KF-DOM yields $\Gamma \vdash N_i : \text{Shape}$. The result then follows from the following proof tree:

$$\begin{array}{c} \Gamma \vdash N_1 : \text{Shape} \quad \Gamma \vdash N_2 : \text{Shape} \\ \hline \Gamma \vdash N_1 \# N_2 : \text{Shape} \quad \text{K-SHAPEPAIR} \\ \hline \Gamma \vdash \text{Dom}(N_1 \# N_2) \quad \text{KF-DOM} \end{array}$$

- *Case K-DOMPROJ.* By induction hypothesis we know $\Gamma \vdash \text{Dom}(N_1 \circ N_2)$, which by case analysis gives us $\Gamma \vdash N_1 \circ N_2 : \text{Shape}$, which by further case analysis gives us $\Gamma \vdash N_1 : \text{Shape}$ and $\Gamma \vdash N_2 : \text{Shape}$, which via *KF-DOM* gives us $\Gamma \vdash \text{Dom}(N_\ell)$.
 - 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 $\Gamma \vdash T_1 : \text{Type}$ and $\Gamma \vdash T_2 : \text{Type}$, which via *K-PAIR* yields $\Gamma \vdash T_1 \times T_2 : \text{Type}$.
 - *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 $\Gamma_1 \vdash \Sigma_1 : \text{State}$.
 - *Case T-LET.* From the assumption $\Gamma_1 \vdash \Sigma_1, \Sigma_2 : \text{State}$ follows by case analysis $\Gamma_1 \vdash \Sigma_1 : \text{State}$ and $\Gamma_1 \vdash \Sigma_2 : \text{State}$. We then apply the induction hypothesis on the typing of e_1 :

$$\frac{\frac{\vdash \Gamma_1 \quad \Gamma_1 \vdash \Sigma_1 : \text{State} \quad \Gamma_1; \Sigma_1 \vdash e_1 : \exists \Gamma_2. \Sigma'_2; T_1}{\vdash \Gamma_1, \# \Gamma_2 \quad \Gamma_1, \# \Gamma_2 \vdash \Sigma'_2 : \text{State} \quad \Gamma_1, \# \Gamma_2 \vdash T_1 : \text{Type}} \text{IH}}{\vdash \Gamma_1, \# \Gamma_2 \quad \Gamma_1, \# \Gamma_2 \vdash T_1 : \text{Type}} \text{IH}$$

and extend the context formation as follows:

$$\frac{\vdash \Gamma_1, \# \Gamma_2 \quad \Gamma_1, \# \Gamma_2 \vdash T_1 : \text{Type}}{\vdash \Gamma_1, \# \Gamma_2, x : T_1} \text{CF-CONSTYPE}$$

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

$$\frac{\frac{\vdash \Gamma_1, \# \Gamma_2, x : T_1 \quad \Gamma_1, \# \Gamma_2, x : T_1 \vdash \Sigma_2, \Sigma'_2 : \text{State} \quad \Gamma_1, \# \Gamma_2, x : T_1; \Sigma_2, \Sigma'_2 \vdash e_2 : \exists \Gamma_3. \Sigma_3; T_2}{\vdash \Gamma_1, \# \Gamma_2, x : T_1, \# \Gamma_3 \quad \Gamma_1, \# \Gamma_2, x : T_1, \# \Gamma_3 \vdash \Sigma_3 : \text{State} \quad \Gamma_1, \# \Gamma_2, x : T_1, \# \Gamma_3 \vdash T_2 : \text{Type}} \text{IH}}{\vdash \Gamma_1, \# \Gamma_2, x : T_1, \# \Gamma_3 \quad \Gamma_1, \# \Gamma_2, x : T_1, \# \Gamma_3 \vdash T_2 : \text{Type}} \text{IH}$$

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

$$\vdash \Gamma_1, \# \Gamma_2, \# \Gamma_3 \quad \Gamma_1, \# \Gamma_2, \# \Gamma_3 \vdash \Sigma_3 : \text{State} \quad \Gamma_1, \# \Gamma_2, \# \Gamma_3 \vdash T_2 : \text{Type}$$

Since $(\Gamma_1, \# \Gamma_2), \# \Gamma_3$ is equivalent to $\Gamma_1, \# (\Gamma_2, \Gamma_3)$ 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 $\Gamma_1 \vdash (\Sigma_1; T_1 \rightarrow \exists \Gamma_2. \Sigma_2; T_2) : \text{Type}$, which by inversion yields the results.
- *Case T-TAPP.* From the induction hypothesis on the typing of v and subsequent case analysis follows $\Gamma, \alpha : K, \mathbb{C} \vdash T : \text{Type}$. By Lemma A.13.5 and Lemma A.14.3 then follows $\Gamma \vdash \{T'/\alpha\}T : \text{Type}$. By Lemma A.13.8 for $\{T'/\alpha\}T \equiv T''$ then follows our result $\Gamma \vdash T'' : \text{Type}$.
- *Case T-SEND, T-SELECT.* The first and third results follow trivially. Repeated case analysis on the kinding of the input state yields $\Gamma \vdash D : \text{Dom}(\mathbb{X})$ and $\Gamma \vdash S : \text{Session}$, which allows to construct the second result $\Gamma \vdash D \mapsto S : \text{State}$ via *K-STCHAN*.
- *Case T-RECV.* Repeated case analysis on the kinding of the input state yields

$$\Gamma \vdash D : \text{Dom}(\mathbb{X}) \quad \Gamma \vdash S : \text{Session} \quad \Gamma \vdash N : \text{Shape} \quad [\Gamma], \alpha' : \text{Dom}(N) \vdash \Sigma' : \text{State} \quad [\Gamma], \alpha' : \text{Dom}(N) \vdash T' : \text{Type}$$

The first result $\vdash \Gamma, \# \alpha' : \text{Dom}(N)$ follows via *CF-CONSKIND* and *CF-CONSCSTR*.

Applying Lemma A.6 (Weakening) on the kindings of Σ' and T' yields

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

Further weakening yields

$$\Gamma, \# \alpha' : \text{Dom}(N) \vdash \Sigma' : \text{State} \quad \Gamma, \# \alpha' : \text{Dom}(N) \vdash T' : \text{Type}$$

from which the second result $\Gamma, \# \alpha' : \text{Dom}(N) \vdash \Sigma', D \mapsto S : \text{State}$ and third result $\Gamma, \# \alpha' : \text{Dom}(N) \vdash T' : \text{Type}$ can be constructed.

- *Case T-CASE.* From $\Gamma \vdash \Sigma, \alpha \mapsto S_1 \& S_2 : \text{State}$ follows $\Gamma \vdash \Sigma, \alpha \mapsto S_1 : \text{State}$ 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).

$$\frac{\vdash \Gamma \quad \Gamma \vdash \Sigma : \text{State} \quad \Gamma; \Sigma \vdash C \quad C \equiv C'}{\Gamma; \Sigma \vdash C'}$$

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).

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

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

PROOF.

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

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

$$\text{ER-BETAFUN} \frac{\text{T-ABS} \frac{\Gamma_1 \vdash (\Sigma_1; T_1 \rightarrow \exists \Gamma_2. \Sigma_2; T_2) : \text{Type} \quad \Gamma_1, x : T_1; \Sigma_1 \vdash e_1 : \exists \Gamma_2. \Sigma_2; T_2}{\Gamma_1 \vdash \lambda(\Sigma_1; x : T_1). e_1 : (\Sigma_1; T_1 \rightarrow \exists \Gamma_2. \Sigma_2; T_2)} \quad \Gamma_1 \vdash v_2 : T_1}{(\lambda(\Sigma_1; x : T_1). e_1) v_2 \hookrightarrow_e \{v_2/x\} e_1} \quad \text{T-APP} \frac{\Gamma_1 \vdash \lambda(\Sigma_1; x : T_1). e_1 : (\Sigma_1; T_1 \rightarrow \exists \Gamma_2. \Sigma_2; T_2) \quad \Gamma_1 \vdash v_2 : T_1}{\Gamma_1; \Sigma_1 \vdash (\lambda(\Sigma_1; x : T_1). e_1) v_2 : \exists \Gamma_2. \Sigma_2; T_2}$$

The result follows via Lemma A.13:

$$\frac{\text{LEMMA A.6} \frac{\Gamma_1 \vdash \Sigma_1 : \text{State}}{\Gamma_1, x : T_1 \vdash \Sigma_1 : \text{State}} \quad \Gamma_1, x : T_1; \Sigma_1 \vdash e_1 : \exists \Gamma_2. \Sigma_2; T_2 \quad \vdash \{v_2/x\} : \Gamma_1, x : T_1 \Rightarrow \Gamma_1}{\text{LEMMA A.13} \frac{\Gamma_1; \Sigma_1 \vdash \{v_2/x\} e_1 : \exists \Gamma_2. \Sigma_2; T_2}}{\Gamma_1; \Sigma_1 \vdash \{v_2/x\} e_1 : \exists \Gamma_2. \Sigma_2; T_2}$$

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

$$\text{ER-BETAALL} \frac{\text{T-KALL} \frac{\vdash \Gamma_1, \alpha : K, \mathbb{C} \quad \Gamma_1, \alpha : K, \mathbb{C} \vdash T : \text{Type}}{\Gamma_1 \vdash \forall(\alpha : K). \mathbb{C} \Rightarrow T : \text{Type}} \quad \Gamma_1, \alpha : K, \mathbb{C} \vdash v : T}{\text{T-TABS} \frac{\Gamma_1 \vdash \forall(\alpha : K). \mathbb{C} \Rightarrow v : \forall(\alpha : K). \mathbb{C} \Rightarrow T \quad \Gamma_1 \vdash T' : K \quad \Gamma_1 \vdash \{T'/\alpha\} \mathbb{C} \quad \{T'/\alpha\} T \Downarrow T''}{\Gamma_1; \cdot \vdash (\Lambda(\alpha : K). \mathbb{C} \Rightarrow v)[T'] \hookrightarrow_e \{T'/\alpha\} v}}$$

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

$$\text{LEMMA A.14} \frac{\Gamma_1 \vdash \{T'/\alpha\} \mathbb{C} \quad \text{LEMMA A.13} \frac{\Gamma_1, \alpha : K, \mathbb{C} \vdash v : T \quad \vdash \{T'/\alpha\} : (\Gamma_1, \alpha : K, \mathbb{C}) \Rightarrow (\Gamma_2, \{T'/\alpha\} \mathbb{C})}{\Gamma_1, \{T'/\alpha\} \mathbb{C} \vdash \{T'/\alpha\} v : \{T'/\alpha\} T}}{\text{T-VAL} \frac{\Gamma_1 \vdash \{T'/\alpha\} v : \{T'/\alpha\} T}{\Gamma_1; \cdot \vdash \{T'/\alpha\} v : \exists \cdot. \{T'/\alpha\} T}}$$

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

$$\text{ER-BETALET} \frac{\text{T-VAL} \frac{\Gamma_1 \vdash v_1 : T_1}{\Gamma_1; \cdot \vdash v_1 : \exists \cdot. T_1} \quad \Gamma_1, x : T_1; \Sigma_2 \vdash e_2 : \exists \Gamma_3. \Sigma_3; T_2 \quad \Gamma_1, x : T_1 \vdash \Sigma_2 : \text{State}}{\text{T-LET} \frac{\Gamma_1; \Sigma_2 \vdash \text{let } x = v_1 \text{ in } e_2 : \exists \Gamma_3. \Sigma_3; T_2}}$$

The result follows via:

$$\frac{\text{LEMMA A.6} \frac{\Gamma_1 \vdash \Sigma_2 : \text{State}}{\Gamma_1, x : T_1 \vdash \Sigma_2 : \text{State}} \quad \Gamma_1, x : T_1; \Sigma_2 \vdash e_2 : \exists \Gamma_3. \Sigma_3; T_2 \quad \vdash \{v_1/x\} : \Gamma_1, x : T_1 \Rightarrow \Gamma_1}{\text{LEMMA A.13} \frac{\Gamma_1; \Sigma_2 \vdash \{v_1/x\} e_2 : \exists \Gamma_3. \Sigma_3; T_2}}$$

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

$$\text{ER-BETAPAIR} \frac{\text{T-PAIR} \frac{\Gamma \vdash v_1 : T_1 \quad \Gamma \vdash v_2 : T_2}{\Gamma \vdash (v_1, v_2) : T_1 \times T_2}}{\text{T-PROJ} \frac{\Gamma \vdash (v_1, v_2) : T_1 \times T_2}{\Gamma; \cdot \vdash \pi_\ell(v_1, v_2) : \exists \cdot. T_\ell}}$$

The result follows via

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

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

$$\frac{\text{ER-LIFT} \quad e_1 \hookrightarrow_e e_2}{\text{let } x = e_1 \text{ in } e \hookrightarrow_e \text{let } x = e_2 \text{ in } e} \quad \Gamma_1; \Sigma_1 \vdash \mathcal{E}[e_1] : \exists \Gamma_3. \Sigma_3; T$$

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

$$\text{LEMMA A.15.1} \quad \frac{\Gamma_1; \Sigma_1 \vdash \mathcal{E}[e_1] : \exists \Gamma_3. \Sigma_3; T}{\exists \Sigma_{11}, \Sigma_{12}, \Gamma_{21}, \Gamma_{22}, \Sigma_2, T' \quad \Sigma_1 = \Sigma_{11}, \Sigma_{12} \quad \Gamma_3 = \Gamma_{21}, \Gamma_{22} \quad \Gamma_1; \Sigma_{11} \vdash e_1 : \exists \Gamma_{21}. \Sigma_2; T' \quad \Gamma_1 \rightsquigarrow \Gamma_{21}, x : T'; \Sigma_{12}, \Sigma_2 \vdash \mathcal{E}[x] : \exists \Gamma_{22}. \Sigma_3; T}$$

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

$$\Gamma_1 \vdash \Sigma_{11} : \text{State} \quad \Gamma_1 \vdash \Sigma_{12} : \text{State}$$

Then we apply the induction hypothesis:

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

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

$$\text{LEMMA A.15.2} \quad \frac{\Gamma_1; \Sigma_{11} \vdash e_2 : \exists \Gamma_{21}. \Sigma_2; T' \quad \Gamma_1 \rightsquigarrow \Gamma_{21}, x : T'; \Sigma_{12}, \Sigma_2 \vdash \mathcal{E}[x] : \exists \Gamma_{22}. \Sigma_3; T}{\Gamma_1; \Sigma_1 \vdash \mathcal{E}[e_2] : \exists \Gamma_3. \Sigma_3; T_2}$$

- (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:

$$\begin{array}{c} \text{CR-FORK} \\ C[\mathcal{E}[\text{fork } v]] \hookrightarrow_C C[(v \text{ unit}) \parallel \mathcal{E}[\text{unit}]] \end{array} \quad \begin{array}{c} \frac{\Gamma \vdash v : (\Sigma_1; \text{Unit} \rightarrow \cdot; \text{Unit})}{\Gamma; \Sigma_1 \vdash \text{fork } v : \exists \cdot; T} \text{T-FORK} \\ \frac{\Gamma; \Sigma_1 \vdash \text{fork } v : \exists \cdot; T}{\Gamma; \Sigma_1, \Sigma_2 \vdash \mathcal{E}[\text{fork } v] : \exists \Gamma'; \cdot; T} \text{LEMMA A.15} \\ \frac{\Gamma; \Sigma_1, \Sigma_2 \vdash \mathcal{E}[\text{fork } v] : \exists \Gamma'; \cdot; T}{\Gamma; \Sigma_1, \Sigma_2 \vdash \mathcal{E}[\text{fork } v]} \text{T-EXP} \\ \frac{\Gamma; \Sigma_1, \Sigma_2 \vdash \mathcal{E}[\text{fork } v]}{\Gamma_0; \Sigma_0 \vdash C[\mathcal{E}[\text{fork } v]]} \text{LEMMA A.16} \end{array}$$

The result follows via

$$\begin{array}{c} \text{T-APP} \quad \frac{\Gamma \vdash v : (\Sigma_1; \text{Unit} \rightarrow \cdot; \text{Unit}) \quad \Gamma \vdash \text{unit} : \text{Unit}}{\Gamma; \Sigma_1 \vdash v \text{ unit} : \exists \cdot; \text{Unit}} \text{T-UNIT} \\ \text{T-EXP} \quad \frac{\Gamma; \Sigma_1 \vdash v \text{ unit} : \exists \cdot; \text{Unit}}{\Gamma; \Sigma_1 \vdash v \text{ unit}} \\ \text{T-PAR} \quad \frac{\Gamma; \Sigma_1 \vdash v \text{ unit} \quad \Gamma; \Sigma_2 \vdash \mathcal{E}[x]}{\Gamma; \Sigma_1, \Sigma_2 \vdash v \text{ unit} \parallel \mathcal{E}[x]} \text{LEMMA A.15} \\ \text{LEMMA A.16} \quad \frac{\Gamma; \Sigma_1, \Sigma_2 \vdash v \text{ unit} \parallel \mathcal{E}[x]}{\Gamma_0; \Sigma_0 \vdash C[v \text{ unit} \parallel \mathcal{E}[x]]} \end{array}$$

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

$$\begin{array}{c} \text{CR-NEW} \\ C[\mathcal{E}[\text{new } S]] \hookrightarrow_C C[vx : [S]. \mathcal{E}[x]] \end{array} \quad \begin{array}{c} \text{T-NEW} \quad \frac{\Gamma \vdash S : \text{Session}}{\Gamma; \Sigma_1 \vdash \text{new } S : \exists \cdot; [S]} \\ \text{LEMMA A.15} \quad \frac{\Gamma; \Sigma_1 \vdash \text{new } S : \exists \cdot; [S]}{\Gamma; \Sigma \vdash \mathcal{E}[\text{new } S] : \exists \Gamma'; \cdot; T} \\ \text{T-EXP} \quad \frac{\Gamma; \Sigma \vdash \mathcal{E}[\text{new } S] : \exists \Gamma'; \cdot; T}{\Gamma; \Sigma \vdash \mathcal{E}[\text{new } S]} \\ \text{LEMMA A.16} \quad \frac{\Gamma; \Sigma \vdash \mathcal{E}[\text{new } S]}{\Gamma_0; \Sigma_0 \vdash C[\mathcal{E}[\text{new } S]]} \end{array}$$

The result follows via

$$\begin{array}{c} \text{T-NUACCESS} \quad \frac{x \text{ fresh} \quad \Gamma \vdash S : \text{Session}}{\Gamma; \Sigma \vdash vx : [S]. \mathcal{E}[x]} \\ \text{LEMMA A.16} \quad \frac{\Gamma; \Sigma \vdash vx : [S]. \mathcal{E}[x]}{\Gamma_0; \Sigma_0 \vdash C[vx : [S]. \mathcal{E}[x]]} \end{array} \quad \begin{array}{c} \text{T-VAR} \quad \frac{\Gamma, x : [S] \vdash x : [S]}{\Gamma, x : [S]; \cdot \vdash x : \exists \cdot; [S]} \\ \text{T-VAL} \quad \frac{\Gamma, x : [S]; \cdot \vdash x : \exists \cdot; [S]}{\Gamma, x : [S]; \Sigma \vdash \mathcal{E}[x] : \exists \Gamma'; \cdot; T} \text{LEMMA A.15} \\ \text{T-EXP} \quad \frac{\Gamma, x : [S]; \Sigma \vdash \mathcal{E}[x] : \exists \Gamma'; \cdot; T}{\Gamma, x : [S]; \Sigma \vdash \mathcal{E}[x]} \end{array}$$

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

$$\begin{array}{c} \text{CR-REQUESTACCEPT} \\ \frac{\alpha, \alpha' \text{ fresh} \quad C \equiv C[vx : [S]. (\mathcal{E}_1[\text{request } x] \parallel \mathcal{E}_2[\text{accept } x] \parallel C')]}{C \hookrightarrow_C C[vx : [S]. vx, \alpha' \mapsto S. (\mathcal{E}_1[\text{chan } \alpha] \parallel \mathcal{E}_2[\text{chan } \alpha'] \parallel C')]} \end{array} \quad \frac{\Gamma; \Sigma \vdash vx : [S]. C}{\Gamma_0; \Sigma_0 \vdash C[vx : [S]. C]} \text{LEMMA A.16}$$

Applying Lemma A.18 to the configuration typing and the congruency yields a configuration typing, which by inversion has 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:

$$\begin{array}{c} \text{T-PAR} \quad \frac{(1) \quad \Gamma, x : [S]; \Sigma_1, \Sigma_2 \vdash (\mathcal{E}_1[\text{request } x] \parallel \mathcal{E}_2[\text{accept } x]) \quad (2) \quad \Gamma, x : [S]; \Sigma_3 \vdash C'}{\Gamma, x : [S]; \Sigma_1, \Sigma_2, \Sigma_3 \vdash (\mathcal{E}_1[\text{request } x] \parallel \mathcal{E}_2[\text{accept } x] \parallel C')} \text{T-PAR} \\ \text{LEMMA A.16} \quad \frac{\Gamma, x : [S]; \Sigma_1, \Sigma_2, \Sigma_3 \vdash (\mathcal{E}_1[\text{request } x] \parallel \mathcal{E}_2[\text{accept } x] \parallel C')}{\Gamma; \Sigma_1, \Sigma_2, \Sigma_3 \vdash vx : [S]. (\mathcal{E}_1[\text{request } x] \parallel \mathcal{E}_2[\text{accept } x] \parallel C')} \text{T-NUACCESS} \end{array}$$

where

$$(1) \frac{\frac{\Gamma, x : [S] \vdash x : [S]}{\Gamma, x : [S]; \cdot \vdash \text{request } x : \exists \alpha : \text{Dom}(\mathbb{X}). \alpha \mapsto S; \text{Chan } \alpha} \text{T-REQUEST}}{\frac{\Gamma, x : [S]; \Sigma_1 \vdash \mathcal{E}_1[\text{request } x] : \exists \Gamma'_1, \alpha : \text{Dom}(\mathbb{X}).; T}{\Gamma, x : [S]; \Sigma_1 \vdash \mathcal{E}_1[\text{request } x]} \text{T-EXP}} \text{LEMMA A.15}$$

$$(2) \frac{\frac{\Gamma, x : [S] \vdash x : [S]}{\Gamma, x : [S]; \cdot \vdash \text{accept } x : \exists \alpha : \text{Dom}(\mathbb{X}). \alpha \mapsto \bar{S}; \text{Chan } \alpha} \text{T-ACCEPT}}{\frac{\Gamma, x : [S]; \Sigma_1 \vdash \mathcal{E}_2[\text{accept } x] : \exists \Gamma'_2, \alpha : \text{Dom}(\mathbb{X}).; T}{\Gamma, x : [S]; \Sigma_1 \vdash \mathcal{E}_2[\text{accept } x]} \text{T-EXP}} \text{LEMMA A.15}$$

The result follows via

$$\frac{\frac{\alpha, \alpha' \text{ fresh} \quad \Gamma \vdash S : \text{Session} \quad (3)}{\Gamma, x : [S]; \Sigma_1, \Sigma_2, \Sigma_3 \vdash v\alpha, \alpha' \mapsto S. (\mathcal{E}_1[\text{chan } \alpha] \parallel \mathcal{E}_2[\text{chan } \alpha'] \parallel C')} \text{T-NUCHAN}}{\frac{\Gamma; \Sigma_1, \Sigma_2, \Sigma_3 \vdash v\alpha, \alpha' \mapsto S. (\mathcal{E}_1[\text{chan } \alpha] \parallel \mathcal{E}_2[\text{chan } \alpha'] \parallel C')}{\Gamma_0; \Sigma_0 \vdash C[v\alpha, \alpha' \mapsto S. (\mathcal{E}_1[\text{chan } \alpha] \parallel \mathcal{E}_2[\text{chan } \alpha'] \parallel C')]} \text{T-NUACCESS}} \text{LEMMA A.16}$$

where

$$(3) \frac{\text{T-PAR} \frac{(4) \quad (5)}{\Gamma'; \Sigma_1, \Sigma_2, \alpha \mapsto S, \alpha' \mapsto \bar{S} \vdash (\mathcal{E}_1[\text{chan } \alpha] \parallel \mathcal{E}_2[\text{chan } \alpha'])} \quad \Gamma'; \Sigma_3 \vdash C'}{\Gamma'; \Sigma_1, \Sigma_2, \Sigma_3, \alpha \mapsto S, \alpha' \mapsto \bar{S} \vdash (\mathcal{E}_1[\text{chan } \alpha] \parallel \mathcal{E}_2[\text{chan } \alpha'] \parallel C')} \text{T-PAR}$$

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

$$(4) \frac{\frac{\frac{\Gamma' \vdash \alpha : \text{Dom}(\mathbb{X})}{\Gamma' \vdash \text{chan } \alpha : \text{Chan } \alpha} \text{T-TVAR}}{\frac{\Gamma' \vdash \text{chan } \alpha : \text{Chan } \alpha}{\Gamma'; \cdot \vdash \text{chan } \alpha : \exists \cdot; \text{Chan } \alpha} \text{T-CHAN}} \frac{\Gamma'; \cdot \vdash \text{chan } \alpha : \exists \cdot; \text{Chan } \alpha}{\Gamma'; \Sigma_1, \alpha \mapsto S \vdash \mathcal{E}_1[\text{chan } \alpha] : \exists \Gamma'_1; T} \text{T-VAL}}{\frac{\Gamma'; \Sigma_1, \alpha \mapsto S \vdash \mathcal{E}_1[\text{chan } \alpha] : \exists \Gamma'_1; T}{\Gamma'; \Sigma_1, \alpha \mapsto S \vdash \mathcal{E}_1[\text{chan } \alpha]} \text{T-EXP}} \text{LEMMA A.15}$$

(5) Similar to (4).

Note that the channels, which in the pre-reduction tree are introduced existentially by the request \cdot and accept \cdot 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:

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

$$\text{LEMMA A.16} \frac{\frac{\alpha, \alpha' \text{ not free in } \Gamma \quad \Gamma \vdash S' : \text{Session} \quad \Gamma'; \Sigma, \alpha \mapsto S', \alpha' \mapsto \bar{S} \vdash C}{\Gamma; \Sigma \vdash v\alpha, \alpha' \mapsto S'. C} \text{T-NUCHAN}}{\Gamma_0; \Sigma_0 \vdash C[v\alpha, \alpha' \mapsto S'. C]}$$

where $\Gamma' = \Gamma, \alpha : \text{Dom}(\mathbb{X}), \alpha' : \text{Dom}(\mathbb{X})$ and $S' = !(\exists \alpha'' : \text{Dom}(N). \Sigma'; T').S$.

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 $\Sigma = \Sigma_1, \Sigma_1, \Sigma_2, \Sigma_3$, where Σ_1 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:

$$\frac{(1) \quad (2)}{\frac{\Gamma'; \Sigma_1, \Sigma_1, \Sigma_2, \alpha \mapsto S', \alpha' \mapsto \bar{S} \vdash (\mathcal{E}_1[\text{send } v \text{ on chan } \alpha] \parallel \mathcal{E}_2[\text{receive chan } \alpha'])}{\Gamma'; \Sigma_1, \Sigma_1, \Sigma_2, \Sigma_3, \alpha \mapsto S', \alpha' \mapsto \bar{S} \vdash (\mathcal{E}_1[\text{send } v \text{ on chan } \alpha] \parallel \mathcal{E}_2[\text{receive chan } \alpha'] \parallel C')} \text{T-PAR}} \text{T-PAR}$$

where

$$(1) \frac{\frac{\Gamma \vdash D : \text{Dom}(N) \quad \{D/\alpha''\} \Sigma' \equiv \Sigma_1 \quad \{D/\alpha''\} T' \equiv T'' \quad \Gamma \vdash v : T'' \quad \Gamma \vdash \text{chan } \alpha : \text{Chan } \alpha}{\Gamma'; \Sigma_1, \alpha \mapsto S' \vdash \text{send } v \text{ on chan } \alpha : \exists \cdot \alpha \mapsto S; \text{Unit}} \text{T-SEND}}{\frac{\Gamma'; \Sigma_1, \Sigma_1, \alpha \mapsto S' \vdash \mathcal{E}_1[\text{send } v \text{ on chan } \alpha] : \exists \Gamma'_1; T_1}{\Gamma'; \Sigma_1, \Sigma_1, \alpha \mapsto S' \vdash \mathcal{E}_1[\text{send } v \text{ on chan } \alpha]} \text{T-EXP}} \text{LEMMA A.15}$$

$$(2) \frac{\frac{\Gamma \vdash \alpha' : \text{Dom}(\mathbb{X}) \quad \Gamma \vdash \text{chan } \alpha' : \text{Chan } \alpha'}{\Gamma'; \alpha' \mapsto \bar{S} \vdash \text{receive chan } \alpha' : \exists \alpha'' : \text{Dom}(N). \Sigma', \alpha' \mapsto \bar{S}; T'} \text{T-RECV}}{\frac{\Gamma'; \Sigma_2, \alpha' \mapsto \bar{S} \vdash \mathcal{E}_2[\text{receive chan } \alpha'] : \exists \Gamma'_2, \alpha'' : \text{Dom}(N).; T_2}{\Gamma'; \Sigma_2, \alpha' \mapsto \bar{S} \vdash \mathcal{E}_2[\text{receive chan } \alpha']} \text{T-EXP}} \text{LEMMA A.15}$$

The result follows via

$$\frac{\frac{\alpha, \alpha' \text{ not free in } \Gamma \quad \Gamma \vdash S : \text{Session} \quad \frac{(3) \quad (4)}{\Gamma'; \Sigma_1, \Sigma_1, \Sigma_2, \alpha \mapsto S, \alpha' \mapsto \bar{S} \vdash \mathcal{E}_1[\text{unit}] \parallel \mathcal{E}_2[v] \parallel C'} \text{T-PAR}}{\Gamma; \Sigma_1, \Sigma_1, \Sigma_2, \Sigma_3 \vdash v\alpha, \alpha' \mapsto S. (\mathcal{E}_1[\text{unit}] \parallel \mathcal{E}_2[v] \parallel C')} \text{T-NUCHAN}}{\Gamma_0; \Sigma_0 \vdash C[v\alpha, \alpha' \mapsto S. (\mathcal{E}_1[\text{unit}] \parallel \mathcal{E}_2[v] \parallel C')]} \text{LEMMA A.16}$$

where

$$(3) \quad \frac{\frac{\frac{\Gamma' \vdash \text{unit} : \text{Unit}}{\Gamma' \vdash \text{unit} : \exists \cdot \cdot; \text{Unit}} \text{T-UNIT}}{\Gamma'; \Sigma_1, \alpha \mapsto S \vdash \mathcal{E}_1[\text{unit}] : \exists \Gamma'_1 \cdot \cdot; T_1} \text{T-VAL}}{\Gamma'; \Sigma_1, \alpha \mapsto S \vdash \mathcal{E}_1[\text{unit}]} \text{LEMMA A.15} \quad \text{T-EXP}$$

$$(4) \quad \frac{\frac{\frac{\Gamma' \vdash v : T'}{\Gamma' \vdash v : \exists \cdot \cdot; T'} \text{T-VAL}}{\Gamma'; \Sigma_1, \Sigma_2, \alpha' \mapsto \bar{S} \vdash \mathcal{E}_2[v] : \exists \Gamma'_2 \cdot \cdot; T_2} \text{LEMMA A.15}}{\Gamma'; \Sigma_1, \Sigma_2, \alpha' \mapsto \bar{S} \vdash \mathcal{E}_2[v]} \text{T-EXP}$$

- *Case CR-SELECTCASE, CR-CLOSE.* Similar to the previous cases.

□

LEMMA A.20 (CONTEXT INVERSION).

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

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

□

LEMMA A.21 (CANONICAL FORMS). *Suppose that $\Gamma \vdash v : T$ and $\text{Outer } \Gamma$.*

- If T is $(\Sigma; T \rightarrow \exists \Gamma'. \Sigma'; T')$, then v is $\lambda(\Sigma; x : T).e$, 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 $\forall(\alpha : K). \mathbb{C} \Rightarrow T$, then v is $\Lambda(\alpha : K). \mathbb{C} \Rightarrow v_1$, for some v_1 .
- If T is Unit , then v is unit .
- If T is $\text{Chan } D$, then v is $\text{chan } D$, for some D .
- If T is $[S]$, then v is x , for some $x \in \text{dom}(\Gamma)$.

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

□

LEMMA 4.5 (PROGRESS FOR EXPRESSIONS).

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

PROOF. The proof is by induction on the expression e .

Case v : Value v holds.

Case let $x = e_1$ in e_2 : By the IH for e_1 we have three cases:

- if Value e_1 , then let $x = e_1$ in $e_2 \hookrightarrow_e \{e_1/x\}e_2$ by ER-BETALET;
- if Comm e_1 , then Comm (let $x = e_1$ 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

$$\Gamma \vdash v_1 : (\Sigma; T \rightarrow \exists \Gamma'. \Sigma'; T') \quad (15)$$

$$\Gamma \vdash v_2 : T \quad (16)$$

By Lemma A.21, $v_1 = \lambda(\Sigma; x : T).e_1$, 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

$$\Gamma \vdash v : T_1 \times T_2 \quad (17)$$

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

$$\Gamma \vdash v : \forall(\alpha : K). \mathbb{C} \Rightarrow T \quad (18)$$

$$\Gamma \vdash T'' : K \quad (19)$$

$$\Gamma \vdash \{T''/\alpha\} \mathbb{C} \quad (20)$$

$$\{T''/\alpha\} T \equiv T' \quad (21)$$

By Lemma A.21, v is $\Lambda(\alpha : K). \mathbb{C} \Rightarrow v_1$, for some v_1 . Hence, $v[T''] \hookrightarrow_e$ by ER-BETAALL.

Case fork v : Inversion of $\Gamma; \Sigma \vdash \text{fork } v : \exists \Gamma'. \Sigma'; T'$ yields $\Gamma' = \cdot, \Sigma' = \cdot, T' = \text{Unit}$, and

$$\Gamma \vdash v : (\Sigma; \text{Unit} \rightarrow \cdot; \text{Unit}) \quad (22)$$

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

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

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

$$\Gamma \vdash v : [S] \quad (23)$$

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

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

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

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

$$\Gamma \vdash D' : \text{Dom}(N) \quad (24)$$

$$\{D'/\alpha'\} \Sigma' \equiv \Sigma_1 \quad (25)$$

$$\{D'/\alpha'\} T'' \equiv T \quad (26)$$

$$\Gamma \vdash D : \text{Dom}(\mathbb{X}) \quad (27)$$

$$\Gamma \vdash v_1 : T \quad (28)$$

$$\Gamma \vdash v_2 : \text{Chan } D \quad (29)$$

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

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

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

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

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

LEMMA 4.8 (PROGRESS FOR CONFIGURATIONS).

$$\frac{\vdash \Gamma \quad \text{Outer } \Gamma \quad \Gamma \vdash \Sigma : \text{State} \quad \Gamma; \Sigma \vdash C}{\text{Final } C \vee \text{Deadlock } C \vee \exists C'. C \hookrightarrow_C C'}$$

PROOF. Suppose that $\neg \text{Final } C$ and $\neg \text{Deadlock } C$. 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 C such that $C = C[e]$ and $\neg \text{Value } e$ and $\neg \text{Comm } e$ or $e = \mathcal{E}[\text{fork } v]$ or $e = \mathcal{E}[\text{new } S]$, for some \mathcal{E}, v, S .

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

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

If $e = \mathcal{E}[\text{new } S]$, 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, $C[e] \hookrightarrow_C C[e']$.

Suppose item 2 is violated. That is, there are configuration and evaluation contexts $C, C_1, C_2, \mathcal{E}_1$, and \mathcal{E}_2 such that $C = C[\nu x: [S]. C']$ and $C' = C_1[\mathcal{E}_1[\text{request } x]]$ and $C' = C_2[\mathcal{E}_2[\text{accept } x]]$. Exploiting congruence we can find a configuration context C' and process C'' such that $C \equiv C'[\nu x: [S]. \mathcal{E}_1[\text{request } x] \parallel \mathcal{E}_2[\text{accept } x] \parallel C'']$, 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 $C, C_1, C_2, \mathcal{E}_1$, and \mathcal{E}_2 such that $C = C[\nu \alpha_1, \alpha_2 \mapsto S. C']$ and $C' = C_1[\mathcal{E}_1[\text{send } v \text{ on chan } \alpha_\ell]]$ and $C' = C_2[\mathcal{E}_2[\text{receive chan } \alpha_{3-\ell}]]$. Exploiting congruence we can find a configuration context C' and process C'' such that

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

which reduces by CR-SENDRECV.

The remaining cases are similar. \square