Polymorphic Typestate for Session Types

Hannes Safrich
University of Freiburg
Germany
safrich@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 calculus, 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 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.

CCS CONCEPTS
• Software and its engineering → General programming languages.

KEYWORDS
binary session types, typestate, polymorphism, existential types

ACM Reference Format:
Hannes Safrich and Peter Thiemann. 2023. Polymorphic Typestate for Session Types. In International Symposium on Principles and Practice of Declarative Programming (PPDP 2023), October 22–23, 2023, Lisboa, Portugal. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3610612.3610624

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

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 $?Int. !Int. s0$ 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
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

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, en-
ables programming in direct style; it does not require linear han-
dling 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 iden-
tity. 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 environment. As a compile-time side
effect, receive changes the typestate to Σ’ = {α !→ Int.S} that
maps channel identities to session types. An invocation of this function has to
create a channel with the same identity α. Shapes also facilitate an extension of PolyVGR
with algebraic datatypes like lists, which was not considered in
previous work.

A final ingredient of the function type in PolyVGR is the innocuous
extension of the function type in VGR the function 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.

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 Chan α. It can only be used in a context that
provides the same channel α, which is somewhat hidden in the VGR
type of server’:

\{u : Unit ; ?·; Chan α\} ; Chan α
(5)

but which becomes very clear in its PolyVGR type:

\{u : Unit ; ?· ; Chan α\}
(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.

\[\forall(\sigma : Session) . \forall(\alpha : Dom(\Sigma)) . \}
\{\alpha : ?Int.? !Int.\sigma \} ; Chan α \rightarrow ?· ; Chan α; Unit \]
(2)

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

The quantification of α is novel to PolyVGR. Its kind, Dom(Σ),
indicates that α ranges over all channel identities. We call Σ a
shape. Shapes allow us to talk about and quantify over the (channel)
resources embedded in a value in PolyVGR. For example, Σ ⊢ Σ 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.

The operation receive takes an integer to transmit and
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.

\[\forall(\sigma : Session) . \forall(\alpha : Dom(\Sigma)) . \}
\{\alpha : ?Int.? !Int.\sigma ; Chan α \rightarrow ?· ; Chan α; Unit \]
(2)
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-kind 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).

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 (8)); 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.

\[
\begin{align*}
\text{C-Accept} & : \Gamma ; \nu \rightarrow [S] \quad \text{fresh} \ c \\
\Gamma ; \Sigma ; \text{accept} \nu \rightarrow \Sigma ; \text{Chan} \ c ; \{c : S\} \\
\text{C-Request} & : \Gamma ; \nu \rightarrow [S] \quad \text{fresh} \ c \\
\Gamma ; \Sigma ; \text{request} \nu \rightarrow \Sigma ; \text{Chan} \ c ; \{c : \bar{S}\}
\end{align*}
\]

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 \(\Sigma_2\). Incoming channels described in \(\Sigma_1\) are either passed through to \(\Sigma_2\) or they are closed in the function. All channels mentioned in \(\Sigma_2\), but not in \(\Sigma_1\) 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:

\[
\text{acc} = \Lambda (\sigma : \text{Session}). \lambda (\cdot : x : [\sigma]). \text{accept} \ x \ni \forall (\sigma : \text{Session}). \lambda (\cdot : [\sigma] \rightarrow \exists \gamma : \text{Dom}(\sigma)). \gamma \rightarrow \sigma ; \text{Chan} \ \gamma)
\]

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:

\[
(\Sigma_1; T_1 \rightarrow \exists \sigma : \text{Dom}(\sigma)). \Sigma_2; T_2).
\]

The incoming state \(\Sigma_1\) specifies the part of the state that is needed by the function; it can be applied in any state \(\Sigma\) 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 \(\exists \sigma : \text{Dom}(\sigma)\).

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 \(\sigma\). Right of the arrow, the existential quantification \(\exists \gamma : \text{Dom}(\gamma)\) abstracts over the created channel. The kind \(\text{Dom}(\gamma)\) indicates abstraction over exactly one channel name. Hence, the variable \(\gamma\) can be used like a channel name in constructing a state. The returned value is a channel reference for \(\gamma\). The existential serves as a modular alternate of the fresh \(c\).

\[\text{We defer further discussion of other shapes } n \text{ and the meaning of } \text{Dom}(n) \text{ to Sections 2.3.3 and 2.3.4.}\]
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 $\alpha$, for some fixed name $\alpha$.

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 $\alpha$ regardless of the name $\alpha$ and the type of server $\tau^*$, 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 $\Sigma^S.S$ classifies a channel on which we can send a channel of type $S^\prime$. Here is the VGR typing rule for the underlying operation:

\[
C\text{-Send}\Gamma; v \mapsto \text{Chan } \beta \quad \Gamma; v' \mapsto \text{Chan } \alpha
\]

\[
\frac{\Gamma; \Sigma, \alpha : \Sigma^S.S, \beta : S'; \text{send } v \text{ on } v' \mapsto \Sigma; \text{Unit}; \alpha : S}{\Gamma; \Sigma, \alpha : \Sigma^S.S, \beta : S'}
\]

The premises are value typings that indicate that $v$ and $v'$ are references to fixed channels $\beta$ and $\alpha$ under variable environment $\Gamma$. The conclusion is an expression typing of the form \( \Gamma; \Sigma; e \mapsto \Sigma_1; T; \Sigma_2 \) where $\Sigma$ is the incoming state, $\Sigma_1$ is the part of $\Sigma$ that is passed through without change, and $\Sigma_2$ is the outgoing state after executing expression $e$ which returns a result of type $T$. The rule states that channels $\beta$ and $\alpha$ have session type $S'$ and $\Sigma^S.S$, respectively. The channel $\beta$ is consumed (because it is sent to the other end of channel $\alpha$) and $\alpha$ 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 $\beta$ is not part of channel $\alpha$’s session type. If the sender still holds references to channel $\beta$, then these references can no longer be exercised as $\beta$ has been removed from $\Sigma$. So one can say that rule C-SendS passes ownership of channel $\beta$ 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 $D$.

\[
C\text{-Send}\Gamma; v \mapsto D \quad \Gamma; v' \mapsto \text{Chan } \alpha
\]

\[
\frac{\Gamma; \Sigma, \alpha : \Sigma^S.S, \beta : S'; \text{send } v \text{ on } v' \mapsto \Sigma; \text{Unit}; \alpha : S}{\Gamma; \Sigma, \alpha : \Sigma^S.S, \beta : S'}
\]

One possibility for type $D$ is a function type like

\[
D_1 = \{ \beta : S'; \text{Unit} \mapsto \text{Unit}; \beta : S'' \}
\]

A function of this type captures a channel named $\beta$ which may or may not occur in $\Sigma$. 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 $\Sigma$ already contains channel $\beta$ of appropriate session type, then we will be able to invoke the function.

If channel $\beta$ 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 $d$:

\[
C\text{-Receive}\Gamma; v \mapsto \text{Chan } \alpha \quad \text{fresh } d
\]

\[
\frac{\Gamma; \Sigma; v \mapsto \Sigma; \text{Chan } d; \alpha : S}{\Gamma; \Sigma; v \mapsto \Sigma; \text{Chan } d; \alpha : S'}
\]

For the same reason, it is impossible to send channel $\beta$ first and then the closure that refers to it: $\beta$ gets renamed to some fresh $d$ while the closure still refers to $\beta$. 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:

\[
S := !(\exists \alpha : \text{Dom}(N).\Sigma).S \mid !(\exists \alpha : \text{Dom}(N).\Sigma).S \mid \ldots
\]

A channel package can be instantiated by a state $\Sigma$ and a payload type $T$. All channels referenced in $T$ must be bound in $\Sigma$ 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 $\Sigma$ is gauged with $\alpha$ which determines its domain as indicated in its kind $\text{Dom}(N)$ where $N$ is the shape of the domain. Shapes range over $I$ (the empty shape), $X$ (the shape with one binding), and $N_1 \uplus N_2$ 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 $\text{Int}$. In the general pattern $!(\exists \alpha : \text{Dom}(N).\Sigma).S$ we find that

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

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

\[
\text{send}0 = \Lambda(c : \text{Dom}(X)).\Lambda(\sigma : \text{Session}).
\]

\[
\lambda(x : \text{Int}).\lambda(e \mapsto !(\exists \alpha : \text{Dom}(1).\cdot; \text{Int}).\sigma; y : \text{Chan} c).
\]

\[
\text{send } x \text{ on } y
\]

\[
: \forall(c : \text{Dom}(X)).\forall(\sigma : \text{Session}).
\]

\[
(\cdot : \text{Int} \rightarrow \cdot ;
\]

\[
(c \mapsto !(\exists \alpha : \text{Dom}(1).\cdot; \text{Int}).\sigma ; \text{Chan } c \mapsto e \mapsto \sigma ; \text{Unit})
\]

(9)
2.3.2 One channel. We instantiate the general pattern
\[
\forall (\exists \alpha \colon \text{Dom}(\mathcal{X}), \Sigma, T). S
\]
as follows to send a channel of type \(S'\).
- \(\Sigma\) is now a state with a single binding, so that \(\alpha\) must range over \(\text{Dom}(\mathcal{X})\);
- consequently, \(\Sigma\) 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 \(\alpha\) and \(c\) are different, so that they can be used as keys in the state. The disjointness constraint \((\alpha \neq c)\) specifies that names in \(\sigma\) are disjoint from names in \(c\).

| send1: | \(\forall (\alpha \colon \text{Dom}(\mathcal{X})). (\forall c \colon \text{Dom}(\mathcal{X})). \ (\alpha \neq c) \Rightarrow \forall (\sigma \colon \text{Session}). \ \exists \alpha \colon \text{Chan} \alpha \rightarrow \cdots ; \ (a \mapsto S', c \mapsto \exists \alpha \colon \text{Dom}(\mathcal{X}), \alpha \mapsto S' ; \text{Chan} \alpha ; \sigma; \text{Chan} c \mapsto c \mapsto \sigma; \text{Unit}))/ | | |
|---|---|---|---|

2.3.3 Two channels. Sending two channels of type \(S'\) and \(S''\) re-quires new ingredients and illustrates the general case. The instantiation of the pattern \((\exists \alpha \colon \text{Dom}(\mathcal{X}), \Sigma, T). S\) is as follows:
- the state \(\Sigma\) must have two bindings, one for each payload channel, so that \(\alpha\) must range over a two element domain, e.g., \(\text{Dom}(\mathcal{X} \uplus \mathcal{Y})\);
- to write down \(\Sigma\), we need notation to address the \(\mathcal{X}\)-shaped components of \(\alpha\) 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: \(T = \text{Chan} (\pi_1 \alpha) \times \text{Chan} (\pi_2 \alpha)\).

\[
\text{let } S(\sigma) = \exists (\exists \alpha \colon \text{Dom}(\mathcal{X} \uplus \mathcal{Y}), \pi_1 \alpha \mapsto S', \pi_2 \alpha \mapsto S'' \cdot \text{Chan} (\pi_1 \alpha) \times \text{Chan} (\pi_2 \alpha)). \sigma \text{ in }
\forall (\alpha \colon \text{Dom}(\mathcal{X})). (\forall \beta \colon \text{Dom}(\mathcal{X})). (\alpha \neq \beta) \Rightarrow
\forall (\sigma \colon \text{Session}). \ (\exists \alpha \colon \text{Dom}(\mathcal{X})). (\alpha \neq \beta) \Rightarrow \forall (\sigma \colon \text{Session}).
\ (\exists \alpha \colon \text{Chan} \alpha \rightarrow \cdots ; \ (a \mapsto S', \beta \mapsto S'', c \mapsto S(\sigma)); \text{Chan} c \mapsto c \mapsto \sigma; \text{Unit}))/
\]

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 \(\alpha : \text{Dom}(\mathcal{X})\) and \(\beta : \text{Dom}(\mathcal{Y})\) and the "uncurried" kind \(\text{Dom}(\mathcal{X} \uplus \mathcal{Y})\) expected by the existential. To rectify this discrepancy, the term pairs the two domains to obtain some \(\gamma = (\alpha, \beta)\) with \(\gamma: \text{Dom}(\mathcal{X} \uplus \mathcal{Y})\) as needed for the existential. This definition of \(\gamma\) 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 \(\text{gsend}\) and show how to obtain the previous examples by instantiation.

\[
gsend: \forall (n: \text{Shape}). (\forall (\alpha : \text{Dom}(n)). \\forall (\xi : \text{Dom}(n) \rightarrow \text{State}). (\forall (\gamma : \text{Dom}(n) \rightarrow \text{Type})).
\forall (c : \text{Dom}(\mathcal{X})). (\alpha \neq c) \Rightarrow \forall (\sigma : \text{Session}).
(\exists \alpha \colon \text{Chan} \alpha \rightarrow \cdots ; \ (\exists \alpha, c \mapsto (\exists \alpha \colon \text{Dom}(n), \xi; \gamma \colon \text{Chan} \alpha ; \sigma; \text{Chan} c \mapsto c \mapsto \sigma; \text{Unit}))/
\]

We abstract over the shape, \(n\), and the corresponding domain. As the state depends on the domain \(\sigma\), we supply it as a closed function \(\xi\) 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 \(\gamma\) from the domain. The remaining quantification over the channel name and the continuation session type is as usual. The disjointness constraint forces the channel name to be different from any name in \(\alpha\). In the body of the type we have a function that takes an argument of type \(\gamma\) \(\alpha\). It returns a function that takes a channel \(c\) along with the resources provided by the state \(\xi\) \(\alpha\). 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 \(\text{gsend}\):
- \(\text{send0} = \text{gsend} \ I \ast (\lambda \gamma. \cdot) (\lambda \gamma. \text{Int})\)
- \(\text{send1} = \Lambda (\alpha : \text{Dom}(\mathcal{X})). \ (\exists \alpha \colon \text{Dom}(\mathcal{X})). (\exists \xi : \text{Dom}(n) \rightarrow \text{State}). (\exists \gamma : \text{Dom}(n) \rightarrow \text{Type}). \ (\text{Chan} \alpha \rightarrow \cdots ; \ (\exists \alpha, c \mapsto \exists (\exists \alpha \colon \text{Dom}(n), \xi; \exists \gamma : \text{Dom}(n) \rightarrow \text{Type})). \ (\exists \alpha \colon \text{Chan} \alpha \rightarrow \cdots ; \ (\exists \alpha, c \mapsto \exists (\exists \alpha \colon \text{Dom}(n), \xi; \exists \gamma : \text{Dom}(n) \rightarrow \text{Type})). \ (\text{Chan} \alpha \rightarrow \cdots ; \ (\exists \alpha, c \mapsto \exists (\exists \alpha \colon \text{Dom}(n), \xi; \exists \gamma : \text{Dom}(n) \rightarrow \text{Type})); \ (\exists \gamma : \text{Chan} (\pi_1 \gamma) \times \text{Chan} (\pi_2 \gamma))\)

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 \(\mathcal{K}\) distinguish between plain types (Type), session types (Session ranged over by metavariable \(S\)), states (State ranged over by \(\Sigma\)), shapes (Shape ranged over by \(N\)), domains (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 \(\alpha\), application, and abstraction over domains to support arrow kinds. Universal quantification over types of any kind is augmented with constraints \(C\), function types contain pre- and post-states as well as existential quantification as explained in Section 2.1. There are channel references that refer to a domain, access points that refer to a session type, the unit type (representing base types), and products to characterize the values of expressions. Session type comprise sending and receiving (cf. Section 2.3), as well as choice and branch types limited to two alternatives, End to indicate the end of a protocol, and \(\overline{S}\) to indicate the dual of a session type (which flips sending and receiving operations as well as choice and branch). Shapes
Kinds $ K ::= $ Type $ | $ Session $ | $ State $ | $ Shape $ | $ Dom(N) $ | $ K \to K $ 

Labels $ \ell ::= 1 \mid 2 $ 

Types $ T, S, N, D, \Sigma ::= \alpha \mid T \mid \lambda(\alpha) \mid \text{Dom}(N) \mid T $ 

Expression Types $ \forall (\alpha : K), C \equiv T \mid (S : T \to \exists \Gamma, \Sigma; T) \mid $ 

$ \text{Chan D} \mid \text{Unit} \mid T \times T $ 

Session Types $ ! (\exists \alpha : \text{Dom}(N), \Sigma; T), S \mid $ 

$ ? (\exists \alpha : \text{Dom}(N), \Sigma; T), S \mid $ 

$ S \uplus S \mid S \& S \mid \text{End} \mid S $ 

Shapes $ I \mid \Sigma \mid N \mid N^\ast $ 

Domains $ * \mid D, D \mid \pi_t D $ 

Session State $ * \mid D \to S \mid \Sigma, \Sigma $ 

Type Environments $ \Gamma ::= - \mid \Gamma, x : T \mid \Gamma, \alpha : K \mid \Gamma, D \# D $ 

Constraints $ C ::= - \mid \Gamma, D \# D $ 

Expressions $ e ::= 0 \mid \text{let } x = e \text{ in } e \mid o \mid \pi_t o \mid a[T] \mid $ 

$ \text{fork } o \mid \text{new } S \text{ accept } o \mid \text{request } o \mid $ 

$ \text{send } o \text{ on } o \mid \text{receive } o \mid \text{select } \ell \text{ on } o \mid $ 

$ \text{case } o \text{ of } \{ e \mid e \} \mid \text{close } o $ 

Values $ v ::= x \mid \text{chan } \alpha \mid \text{unit} \mid (o, v) \mid $ 

$ \lambda(\Sigma; x : T), e \mid \Lambda(\alpha : K), C \Rightarrow o $ 

Configurations $ C ::= e \mid [\Gamma, C] \mid \nu a, \alpha \Rightarrow S, C \mid \nu x : [S, C] $ 

Expression Contexts $ \mathcal{E} ::= \emptyset \mid \text{let } x = e \text{ in } e $ 

Configuration Contexts $ C ::= \emptyset \mid \nu a, \alpha \Rightarrow S, C \mid \nu x : [S, C] \mid (\Gamma \parallel C) $ 

Figure 1: Syntax of PolyVGR

The empty shape $ I $, the single-channel shape $ \Sigma $, and the combination of two shapes $ _1 \mid _1 $. The corresponding domains are the empty domain $ * $, the combination of two domains $ _1 \& _1 $, and the first/second projection of a domain. A session state 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 $ \Gamma $ contain bindings for expression variables and type variables, as well as disjointness constraints between domains. Constraints $ 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 $ \Gamma \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 $ \Gamma \equiv T $,  
- constraint entailment $ \Gamma \vdash C $,  
- context restriction operators $ [\Gamma] $ and $ [\Gamma]_{\parallel} $,  
- disjoint context extension operator $ \Gamma \uplus \Gamma $.

Context formation (Figure 2) is standard up to the case for disjointness constraints. For those, we have to show that each domain is wellformed with respect to the current context $ \Gamma $, 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 \( \Gamma_2 \) of channels created by the function. This state must be disjoint from the assumptions as indicated by \( \Gamma_1 \rightarrow \Gamma_2 \) (see Figure 8). We also make sure that \( \Gamma_2 \) only contains domains.

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

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

The remaining rules for session types are standard.

Figure 5 contains the rules for shapes, domains, and states. We discussed shapes with their syntax already. The domain rules are similar to product rules with the additional disjointness constraint on the components of the combined domain. Empty states are trivially wellformed. A single binding is wellformed if it maps a single-channel domain to a session type. Figure 6 defines type conversion, where we omit the standard rules for reflexivity, transitivity, symmetry, and congruence. Conversion comprises beta reduction for functions and pairs, 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-kind type variable \( \alpha \), then the operator cannot be fully eliminated as in \( \bar{\alpha} \). Once a type application
instantiates $\alpha$, 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 · on · 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 conjunctsions of constraints in the obvious way.

The context restriction operators, $\Gamma'$ and $\Gamma_1$, 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 $\Gamma_1 \cdot_2 \Gamma_2$. The assumption is that $\Gamma_1$ is known to contain disjoint bindings. The generated constraints $C_2$ make sure that $\Gamma_2$’s bindings are also disjoint and $C_{12}$ ensures that they are also disjoint from $\Gamma_1$’s bindings.

\[ \Gamma_1 \cdot_2 \Gamma_2 = \Gamma_1 \cdot_2 \Gamma_2, C_2, C_{12} \]

where

\[ C_2 = \{ \alpha \cdot \beta \mid \alpha, \beta \in \text{dom}(\Gamma_2), \alpha \neq \beta \} \]

\[ C_{12} = \{ \alpha \cdot \beta \mid \alpha \in \text{dom}(\Gamma_1), \beta \in \text{dom}(\Gamma_2) \} \]

**Figure 8: Disjoint context extension ($\Gamma_1 \cdot_2 \Gamma_2$)**

\[ \frac{\Gamma, x : T \vdash x : T}{\Gamma \vdash \text{unit} : \text{Unit}} \]

\[ \frac{\Gamma \vdash \alpha_1 : T_1 \quad \Gamma \vdash \alpha_2 : T_2}{\Gamma \vdash \langle \alpha_1, \alpha_2 \rangle : T_1 \times T_2} \]

\[ \frac{\Gamma \vdash \forall (\alpha : K), \ C \Rightarrow T : \text{Type}}{\Gamma \vdash \lambda (\alpha : K), C \Rightarrow \alpha : T} \]

\[ \frac{\Gamma \vdash \Sigma : \text{Type} \quad \Gamma \vdash e : \Sigma_1, e : \Sigma_2}{\Gamma \vdash \lambda (\Sigma_1, e : \Sigma_2) : T_1 \rightarrow \Sigma_2, \Sigma_2 : T_2} \]

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

### 3.3 Statics for expressions and processes

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

- **value typing** $\Gamma \vdash \alpha : T$,
- **expression typing** $\Gamma, \Sigma \vdash e : \forall \Gamma, \Sigma; T$, 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 well-formedness 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 $\Sigma$ 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 $\Sigma_1$ required by the header expression $e_1$ and $\Sigma_2$ for the continuation $e_2$, but then it feeds the entire outgoing state of $e_1$ combined with $\Sigma_2$ 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 partial part of this judgment is reintegrated into the state in the T-Let rule, which inserts the necessary disjunctive constraints via the disjoint append-operator rule. 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 for 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 route one end of the freshly created channel, so that the channel’s domain is existentially quantified. The kind of this domain is \( \text{Dom}(X) \) (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 \( \Sigma \), which will be passed along with the value. The rule guesses a domain \( D’ \) such that the state expected in the session type matches the state \( \Sigma \) 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 \( S \).

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

Rule T-Select performs the standard rewrite of the type state for selecting a branch in the protocol. The dual type 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 \( \Sigma \). This way, operations on the closed channel are disabled, but it is still possible to have references to it in dead code.

### 3.4 Dynamics

Figure 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 as our rule for function application.
new process that applies the fork

channel abstraction

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

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

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

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
rarearranges 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 re-
quest 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).
The value expression is either a value, stuck on a communication (or fork), or configuration. Besides type variables and constraints, they can only inductively.

Another process to reduce.

\_ 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:
  - \( e = \text{fork} \, \lambda(x : T).e_1 \),
  - \( e = \text{select} \, \ell \) on chan \( D \),
  - \( e = \text{case} \, \text{chan} \, D \) of \( \{ e_1 : e_2 \} \),
  - \( e = \text{accept} \, v \),
  - \( e = \text{receive chan} \, D \),
  - \( e = \text{send} \, v \) on chan \( D \),
  - \( e = \text{close chan} \, D \),

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 \( \Gamma \) is defined by:

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

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

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 \( (\forall x : [S], C_1) \) if Final \( C_1 \), or
- Final \( (\forall x, \alpha' \mapsto \text{End}, C_1) \) if Final \( C_4 \).

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 \( v \neq \text{new} \, S \).
2. For all configuration contexts \( C \), if \( C = C[\forall x : [S], C'] \), then
   - if \( C' = C_1[\forall x_1 : [S], (C_2 \parallel C_3)] \), then there is no \( C_2, C_3 \) such that \( C' = C_2[C_3]\),
   - if \( C' = C_1[\forall x_1 : [S], C_2] \), then there is no \( C_2, C_3 \) such that \( C' = C_2[C_3]\).
3. For all configuration contexts \( C \), if \( C = C[\forall x_1, \alpha_2 \mapsto S, C'] \), then
   - if \( C' = C_1[\forall x_1, \alpha_2 \mapsto S, C_2] \), then there is no \( C_2, C_3 \) such that \( C' = C_2[C_3]\).

Lemma 4.8 (Progress for configurations).

\[ \Gamma \vdash \text{Outer} \Gamma \quad \Gamma \vdash \Sigma : \text{State} \quad \Gamma, \Sigma \vdash C \]

Final \( C \lor \text{Deadlock} \quad C \lor \exists C'. \quad 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 \( \Gamma \vdash C \) is decidable by normalizing and decomposing \( \Gamma \) and \( C \) into closed sets of atomic constraints \( A_T \) and \( A_C \) and checking \( A_T \subseteq A_C \). Decomposition is done according to CE-Split and CE-Sym, yielding constraints of form \( d_1 \equiv d_2 \) where \( d_i = \pi_{i_1} \ldots \pi_{i_n} \). Then the closure is taken with respect to CE-ProjMerge, CE-ProjSplit, and CE-Sym.
- The T-LET and T-PAR rules nondeterministically split the input state \( E_1, E_2 \) 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.
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 α + Chan β 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? Clearly, Σ = α ⊳ S₁; β ⊳ S₂ 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 α + Chan β, 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

\[ T, N, D ::= \cdots | T + T | N + N | D + D | \tau ; D \]

Expressions

\[ e ::= \cdots | \text{inj}_1 e | \text{match } \sigma \text{ of } \{ x : e; x : e \} \]

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 \( \tau ; D \) and \( \tau ; D \) 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{align*}
\text{K-Sum} & \quad \Gamma \vdash T_1 : \text{Type} \quad \Gamma \vdash T_2 : \text{Type} \\
\hline
\Gamma \vdash T_1 + T_2 : \text{Type} & \\
\text{K-Shapesum} & \quad \Gamma \vdash N_1 : \text{Shape} \quad \Gamma \vdash N_2 : \text{Shape} \\
\hline
\Gamma \vdash N_1 + N_2 : \text{Shape} & \\
\text{K-DomFrom} & \quad \Gamma \vdash D : \text{Dom}(N_1 + N_2) \\
\hline
\Gamma \vdash \tau ; D : \text{Dom}(N_1) & \\
\text{K-DomSum} & \quad (\forall y) \; \Gamma \vdash D_1 : \text{Dom}(N_1) \\
\hline
\Gamma \vdash \tau ; D_1 + D_2 : \text{Dom}(N_1 + N_2) & \\
\end{align*}

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

\begin{align*}
\text{T-Inj1} & \quad (\forall y) \; \Gamma \vdash \text{inj}_1 e : \text{Dom}(N_1 + N_2) \\
\hline
\Gamma \vdash e : T & \\
\text{T-Match} & \quad (\forall y) \; \Gamma \vdash T_1 + T_2 : \text{Dom}(N_1 + N_2) \\
\hline
\Gamma \vdash \text{match } \sigma \text{ of } \{ x : e; x : e \} & \\
\end{align*}

In rule T-Inj1, we are given a value \( e : T \) along with some \( \Sigma \) that describes the channels contained in \( e \). We assume that the shape of \( \Sigma \) is described by \( N_1 \) and corresponding domain \( D \). We further assume that the alternatives of the sum are described by type functions \( \Sigma_1, T_1 \) and \( \Sigma_2, T_2 \). The point is that the pair labeled 1 describes the real resources in \( e \) represented by \( \Sigma \) 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 \( \beta \). 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 \( \text{inj}_2 e \).

The alert reader might wonder why we do not treat \( \text{inj}_2 e \) as a value. Indeed, \( \text{inj}_1 e \) comes with a reduction to create the virtual resource \( \beta \), which returns a syntactic value \( \text{inj}_2 e \). We elide the corresponding value typing rule, which is obtained from T-Inj1 by stripping the \( \Sigma \) components and assuming the presence of both domains in \( \Gamma \).

To match on a value \( e \) 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 \( \perp \) and the corresponding state is empty.) As in the introduction rule, the type and state functions \( \tilde{T} \) and \( \tilde{\Sigma} \) 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 \( \pi \)-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 \( \pi \)-calculus [36]. Caires et al.
Analogous to the channel types in our system, pointers in the aliasing, but the labels in the records types are fixed and two records calculating with symmetric concatenation [16].

Xie et al. [39] show that calculi for disjoint polymorphism. 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 and the shape of the store is described by aliasing constraints of the form \( \alpha : K(Y, Z) \) where \( K = \text{Type} \), \( Y = \alpha \), 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.

Dardha et al. [9] extend an encoding of session types into \( \pi \)-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 \( K = \text{Type} \), \( Y = \alpha \), 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.

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 \( \text{IO} \) 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 [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 excessively 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 typestate, 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 typestate.
- 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.

Proofs and further examples may be found in our accompanying technical report [30].

REFERENCES

[1] Bernardo Almeida, Andrieta 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 Alpírio, Bruno C. d. S. Oliveira, and Zhiyuan Shi. 2017. Directioy Polymorphism. In Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017 (Lecture Notes in Computer Science, Vol. 10203), Hoangseok Yang (Ed.). Springer, Uppsala, Sweden, 1–28. https://doi.org/10.1007/978-3-662-54434-1_1
[3] Davide Ancona, Viviana Bonc, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Dénériol, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascari, Fabrizio Montesi, Rumyana Neykova, Nicolás Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. 2016. Behavioral Types in Programming Languages. Found. Trends Program. Lang. 3, 2-3 (2016), 95–230. https://doi.org/10.1561/2500000031
[4] Robert Atkey. 2009. Parameterised Notions of Computation. In Joint Conferences on Theory and Practice of Software, ETAPS 2009, Eindhoven, The Netherlands, April 2-8, 2009, Proceedings (Lecture Notes in Computer Science, Vol. 5492), Pedro Pitts and Andy Wosawski (Eds.). Springer, 401–418. https://doi.org/10.1007/978-3-642-11407-2_16
[5] 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), Petrida Stevens and Andrew Wosawski (Eds.). Springer, 399–418. https://doi.org/10.1007/978-3-662-46965-7_24
[6] Hans Hüttel, Ivan Lanes, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Dénériol, Dimitri Mosesrout, Luca Padovani, António Ravao, Emilio Tuosto, Hugo Torres Vieira, and Gianlugi Zavattaro. 2016. Foundations of Session Types and Behavioural Contracts. ACM Comput. Sur. 49, 1 (2016), 1–33-36. https://doi.org/10.1145/2783652
[7] 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
[8] Simon Gay and António Ravara (Eds.). 2017. Behavioural Types; from Theory to Tools. ACM, 178–198. https://doi.org/10.1145/3437792.3439914
[9] Alceste Scalas and Nobuko Yoshida. 2016. Lightweight Session Programming with (Almost) 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/10.1145/99583.99680
[10] Jonas Kastberg Hinrichsen, Daniel Louwir, Robert Krebsen, and Jasper Bengtson. 2021. Machine-checked session typing system. In CPP ’21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17–19, 2021, Catalin Hritcu and Andrea Pnepescu (Eds.). ACM, 178–198. https://doi.org/10.1145/3437792.3439914
[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/10.1007/3-540-44625-5_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