Relating Functional and Imperative Session Types

Hannes Saffrich, Peter Thiemann

▶ **To cite this version:**

Hannes Saffrich, Peter Thiemann. Relating Functional and Imperative Session Types. 23th International Conference on Coordination Languages and Models (COORDINATION), Jun 2021, Valletta, Malta. pp.61-79, 10.1007/978-3-030-78142-2_4 . hal-03387840

HAL Id: hal-03387840

https://inria.hal.science/hal-03387840v1

Submitted on 20 Oct 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Relating Functional and Imperative Session Types

Hannes Saffrich[0000-0002-1825-0097] and Peter Thiemann[0000-0002-9000-1239]

University of Freiburg, Germany
{saffrich,thiemann}@informatik.uni-freiburg.de

Abstract. Imperative session types provide an imperative interface to session-typed communication in a functional language. Compared to functional session type APIs, the program structure is simpler at the surface, but typestate is required to model the current state of communication throughout.
Most work on session types has neglected the imperative approach. We demonstrate that the functional approach subsumes previous work on imperative session types by exhibiting a typing and semantics preserving translation into a system of linear functional session types.
We further show that the untyped backwards translation from the functional to the imperative calculus is semantics preserving. We restrict the type system of the functional calculus such that the backwards translation becomes type preserving. Thus, we precisely capture the difference in expressiveness of the two calculi and conclude that the lack of expressiveness in the imperative calculus is solely due to its type system.

Keywords: Session types · distributed programming · translation.

1 Introduction

Session types provide a type discipline for bidirectional communication protocols in concurrent programs. They originate with papers by Honda and others [9, 25], who proposed them as an expressive type system for binary communication in pi-calculus. Later work considered embeddings in functional and object-oriented languages, both theoretically and practically oriented [7, 10, 22, 15].

A typical incarnation of session types [7] supports a data type of channel ends described by a session type s governed by a grammar like this one:

$$s ::= !t.s \mid ?t.s \mid \oplus\{\ell_i : s_i\} \mid \&\{\ell_i : s_i\} \mid \mathsf{End} \qquad t ::= s \mid t \to t \mid t \otimes t \mid \dots$$

Here, t ranges over all types in the language (functions, pairs, etc) including session types s. The session type $!t.s$ describes a channel on which we can send a value of type t and then continue communicating according to s. Dually, we can receive a value of type t and continue according to s on a channel of type $?t.s$. The internal choice type $\oplus\dots$ selects a choice ℓ_i and continues according to s_i. The external choice $\&\dots$ continues with s_i if it receives ℓ_i. The session type End signifies the end of the conversation.

```
let server u =
  let (x, u) = receive u in     (* u: ?Int.?Int.!Int.s' *)
  let (y, u) = receive u in     (* u: ?Int.!Int.s' *)
  send (x+y, u)                 (* u: !Int.s' *)
```

Listing 1.1. Example server in functional style

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

Listing 1.2. Example server

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

Listing 1.3. Example server with capture

Functional vs Imperative Session Types Most of the embedded session type systems rely on a functional treatment of channel ends. That is, the communication operations transform (the type of) a channel end as shown in this example where the receive operation consumes a channel of type $?t.s$ and returns a pair of the received value of type t and the continuation channel of type s:

$$\mathsf{receive} : ?t.s \to (t \otimes s)$$

This design forces a programmer to explicitly thread the channel reference through the program. Moreover, the channel reference must be treated linearly because a repeated use at the same type would break the protocol. The typical programming pattern is to rebind a variable, say `u`, containing the channel end with a different type in every line as in Listing 1.1 (typings refer to the state *before* the operation in that line). Writing a program in this style feels like functional programming before the advent of monads, when programmers loudly complained about the need for "plumbing" as demonstrated with `u`. Moreover, this style is not safe for session types because most languages do not enforce the linearity needed to avoid aliasing of channel ends at compile time.

Embeddings in object-oriented languages make use of fluent interfaces, which favor the chaining of method calls [11], while embeddings in functional languages wrap a channel into a monad [17], which does not scale well to programs that process multiple channels. But much less work can be found that takes the alternative, imperative approach inspired by typestate-based programming [24].

Vasconcelos, Gay, and Ravara [27] proposed a session type calculus embedded in a multithreaded functional language, which we call VGR. It is a bit of a mystery why VGR was not called imperative[1] because it enables rewriting the program fragment in Listing 1.1 as shown in Listing 1.2. The parameter `u` of the `server` function is a reference to a communication channel. The operation `receive` takes a channel associated with session type $?\mathsf{Int}.S$ and returns an integer[2]. Executing `receive` changes the type of the channel referred to by `u` to S, which

[1] The conference version of their paper [28] is called "Session Types for Functional Multithreading".

[2] Uppercase letters denote types in the VGR calculus.

indicates that the VGR calculus is a typestate-based system [24]. The function `send_on_` takes an integer to transmit and a channel associated with session type $!\mathsf{Int}.S$. It returns a unit value and updates the channel's type to S.

Taken together, the `server` function in Listing 1.2 expects that its argument `u` refers to a channel of type $?\mathsf{Int}.?\mathsf{Int}.!\mathsf{Int}.S'$ and leaves it in a state corresponding to type S' on exit. This functionality is reflected in the shape of a function type in VGR: $\Sigma_1; T_1 \to T_2; \Sigma_2$. In this type, T_1 and T_2 are argument and return type of the function. The additional components Σ_1 and Σ_2 are environments that reflect the state (session type) of the channels before (Σ_1) and after (Σ_2) calling the function. The type of a channel, $\mathsf{Chan}\ \alpha$, serves as a pointer to the entry for α in the current channel environment Σ. Channels in T_1 refer to entries in Σ_1 and channels in T_2 refer to entries in Σ_2, but both environments may refer to further channels that describe channel references captured by the function (Σ_1) or created by the function (Σ_2). In Listing 1.2, the type of `server` is

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

for some fixed channel name α and session type S.

Compared to other session type systems [7, 6], VGR does **not** require linear handling of channel references, as can be seen by the multiple uses of variable `u` in Listing 1.2. Instead, it keeps track of the current state of every channel using the environment Σ, which is threaded linearly through the typing rules.

In § 2 we give deeper insights into VGR, the kind of programs that it accepts, and the programs that fail to typecheck. To give a glimpse of its peculiarities, we examine the type of `server` in eq. (1) more closely.

First, the type refers to the *name* α. This name identifies a certain channel so that the function cannot be invoked on other channels. Second, a function of this type can be typechecked without knowledge of the channel names that are currently in use and their state. This property enables the definition of the `server` function in a library, say, but the typechecker does not allow us to call the function on a channel named differently than α, even if its session type matches. Hence, the library may end up defining a function that cannot be called.

A variation of the type in eq. (1) replaces the argument type by Unit:

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

This type can be assigned to a function like `server'` in Listing 1.3 that is closed over a reference to a channel of type $\mathsf{Chan}\ \alpha$. In this context, the fixation on a certain channel name α is required for soundness: While we might want to apply a function to different channels, it is not possible to replace a channel captured in a closure. A function of type (2) may be called any time the channel α is in a state matching the "before" session type of the function.

Subsequently, Gay and Vasconcelos created a functional session type calculus based on a linear type system, which was later called LFST[3] [7]. While LFST is still monomorphic, a function like `server` can be applied to several different

[3] Linear Functional Session Types.

channels with the same session type. In LFST, we can also close over a channel, but doing so turns a function like `server'` into a single-use function, whereas it can be called many times in VGR provided the channel α is available at the right type in the environment. Clearly, LFST lifts some restrictions of the VGR calculus, but it seems to impose other restrictions. In any case, the exact correspondence between the two calculi has never been studied.

There is another line of session-type research based on the Curry-Howard correspondence between fragments of linear logic and process calculi [4]. Programs/processes in these systems may also be regarded as handling channels "imperatively", perhaps even more so than VGR. We discuss these approaches in Section 7 along with other related work.

Contributions

- We show that LFST is at least as expressive as VGR by giving a typing-preserving translation which simulates VGR in LFST (§ 5).
- We show that untyped VGR is at least as expressive as LFST by giving a backwards translation which simulates LFST in VGR (§ 6).
- We exhibit a type system for LFST that characterizes the shortcomings of VGR exactly. The backwards translation becomes type preserving with respect to this system (§ 6.2).

In this paper we omit session type choice and recursion because these features are straightforward to add and our results extend seamlessly. An extended version of this paper with further proofs, the full rulesets of VGR and LFST, and full definitions of the translations is available at http://arxiv.org/abs/2010.08261.

2 Motivation

Channel Identities Our discussion of VGR's function type $\Sigma_1; T_1 \to T_2; \Sigma_2$ in the introduction shows that a function that takes a channel as a parameter can only be applied to a single channel. A function like `server` (Listing 1.2) must be applied to the channel of type $\mathsf{Chan}\ \alpha$, for some fixed name α.

LFST sidesteps this issue by not encoding the identity of a channel in the type. It rather posits that session types are linear so that channel references cannot be duplicated. In consequence, the operations of the session API must consume a channel and return a (nother) channel to continue the protocol.

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

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

The premises are *value typings* that indicate that v and v' are references to different, fixed channels β and α under variable environment Γ. The conclusion is an *expression typing* of the form $\Gamma; \Sigma; e \mapsto \Sigma_1; T; \Sigma_2$ where Σ is the incoming channel environment, Σ_1 is the part of Σ that is passed through without change, and Σ_2 is the outgoing channel environment after the operation indicated by expression e which returns a result of type T. The rule states that channels β and α have session type S' and $!S'.S$, respectively. The channel β is consumed (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. Alas, if the sender holds references to channel β (i.e., values of type $\mathsf{Chan}\ \beta$), then these references can no longer be exercised as β has been removed from Σ. So one can say that rule C-SendS passes ownership of channel β to the receiver.

Abstracting over the send operation is not useful because it would fix channel names in the function type.

However, there is another way to send a channel reference over a channel, namely if it is captured in a closure. To see what happens in this case, we look at VGR's typing rules for sending and receiving data of type D. Types of the form D comprise first-order types and function types, but not channels.

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

One possibility for type D is a function type like $D_1 = \{\beta : S'\}; \mathsf{Unit} \to \mathsf{Unit}; \{\beta : S''\}$. 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, it turns out we cannot send it later, as the received channel gets assigned a fresh name d:

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

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

None of these issues arise in LFST because channels have no identity. Hence, any value whatsoever can be sent over a channel, higher-order session types are possible, and there is just one typing rule for sending and receiving, respectively.

Channel Aliasing The VGR paper proposes the following function `sendSend`.

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

```
fun sendSend u v sigma =
  let (cu, sigma) = sigma.u in
  let cu' = send 1 on cu in
  let sigma = sigma * {u: cu'} in
  let (cv, sigma) = sigma.v in
  let cv' = send 2 on cv in
  let sigma = sigma * {v: cv'} in
  ((), sigma)
```

Listing 1.4. Without aliasing

```
fun sendSend w sigma =
  let (cw, sigma) = sigma.w in
  let cw' = send 1 on cw in
  let sigma = sigma * {w: cw'} in
  let (cw, sigma) = sigma.w in
  let cw' = send 2 on cw in
  let sigma = sigma * {w: cw'} in
  ((), sigma)
```

Listing 1.5. With aliasing

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

$$\mathsf{sendSend} : \Sigma_1; \mathsf{Chan}\ u \to (\Sigma_1; \mathsf{Chan}\ v \to \mathsf{Unit}; \Sigma_2); \Sigma_1 \tag{3}$$

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

Ignoring the types we observe that it would be semantically sound to pass a reference to the same channel `w`, say, of session type `!Int.!Int.End` for `u` and `v`. However, `sendSend w w` does not type check with type (3) because `w` would have to have identity u and v at the same time, but environment formation mandates they must be different.

Another typing of `sendSend` in VGR would be

$$\mathsf{sendSend}' : \Sigma_1; \mathsf{Chan}\ w \to (\Sigma_1; \mathsf{Chan}\ w \to \mathsf{Unit}; \Sigma_2); \Sigma_1 \tag{4}$$

with $\Sigma_1 = \{w : !\mathsf{Int}.!\mathsf{Int}.S_w\}$ and $\Sigma_2 = \{w : S_w\}$. With this typing, `sendSend w w` type checks. Indeed, the typing forces the two arguments to be aliases!

In LFST, the invocation `sendSend w w` is not legal as it violates linearity. Indeed, to simulate the two differently typed flavors of `sendSend` requires two different expressions in LFST. As an illustration, we show LFST expressions as they are produced by our type-driven translation in Section 5. In the code fragment in Listing 1.4, `u` and `v` have unit type (translated from $\mathsf{Chan}\ u$ and $\mathsf{Chan}\ v$) and `sigma` is a linear record with fields `u` and `v` that contain the respective channels. The dot operator performs field selection and `*` is disjoint record concatenation. The notation for record literals is standard.

In the translation of `sendSend'` in Listing 1.5 record `sigma` only has one field `w` containing the channel.

Abstraction over Channel Creation A server typically accepts many connections on the same access point and performs the same initialization (e.g., authentication) on each channel. Hence, it makes sense to abstract over the creation of a channel.

```
fun acceptAdd () =
  let c = accept addService in
  // authenticate client on c (omitted)
  c
```

However, in VGR, the freshness condition on the channel created by `accept` only applies inside the function body. The actual VGR type of `acceptAdd` does not reflect

$$
\begin{array}{ll}
C ::= \langle t \rangle \mid (C \| C) \mid (\nu x : [S])C \mid (\nu\gamma)C & \\
t ::= v \mid \mathsf{let}\, x = e\, \mathsf{in}\, t \mid \mathsf{fork}\, t; t & T ::= D \mid \mathsf{Chan}\ \alpha \\
e ::= t \mid v\, v \mid \mathsf{new}\, S \mid \mathsf{accept}\, v \mid \mathsf{request}\, v & D ::= [S] \mid \Sigma; T \to T; \Sigma \mid \mathsf{Unit} \\
\quad \mid \mathsf{send}\, v\, \mathsf{on}\, v \mid \mathsf{receive}\, v \mid \mathsf{close}\, v & S ::= ?D.S \mid !D.S \mid ?S.S \mid !S.S \mid \mathsf{End} \\
v ::= \alpha \mid \lambda(\Sigma; x : T).e \mid () & \Sigma ::= \varnothing \mid \Sigma, \alpha : S \qquad (\alpha \notin \Sigma) \\
\alpha ::= x \mid \gamma^p & \Gamma ::= \varnothing \mid \Gamma, x : T \qquad (x \notin \Gamma) \\
p ::= + \mid - &
\end{array}
$$

Fig. 1. Syntax of VGR

$$E[(\lambda(\Sigma; y : T).e)v] \Rightarrow_e E[e[v/y]] \tag{5}$$

$$E[\mathsf{let}\, x = v\, \mathsf{in}\, t] \Rightarrow_e E[t[v/x]] \tag{6}$$

$$E[\mathsf{request}\, n] \overset{\mathsf{request}\gamma}{\Rightarrow_e} E[\gamma^+] \qquad E[\mathsf{accept}\, n] \overset{\mathsf{accept}\gamma}{\Rightarrow_e} E[\gamma^-] \tag{7}$$

$$E[\mathsf{receive}\, \gamma^p] \overset{\gamma^p ?v}{\Rightarrow_e} E[v] \qquad E[\mathsf{send}\, v\, \mathsf{on}\, \gamma^p] \overset{\gamma^p !v}{\Rightarrow_e} E[()] \tag{8}$$

$$\frac{t_1 \overset{\mathsf{request}\gamma}{\Rightarrow_e} t_1' \qquad t_2 \overset{\mathsf{accept}\gamma}{\Rightarrow_e} t_2'}{\langle t_1 \rangle \| \langle t_2 \rangle \overset{\mathsf{accept}}{\Rightarrow_p} (\nu\gamma)\langle t_1' \rangle \| \langle t_2' \rangle} \qquad \frac{t_1 \overset{\gamma^p ?v}{\Rightarrow_e} t_1' \qquad t_2 \overset{\gamma^{\overline{p}} !v}{\Rightarrow_e} t_2'}{\langle t_1 \rangle \| \langle t_2 \rangle \overset{\mathsf{send}}{\Rightarrow_p} \langle t_1' \rangle \| \langle t_2' \rangle} \tag{9}$$

$$\langle E[\mathsf{new}\ S] \rangle \overset{\mathsf{new}}{\Rightarrow_p} (\nu n : [S])\langle E[n] \rangle \tag{10}$$

$$\langle E[\mathsf{fork}\, t_1; t_2] \rangle \overset{\mathsf{fork}}{\Rightarrow_p} \langle t_1 \rangle \| \langle E[t_2] \rangle \tag{11}$$

$$\frac{t \Rightarrow_e t'}{\langle t \rangle \Rightarrow_p \langle t' \rangle} \qquad \frac{C \overset{\ell}{\Rightarrow_p} C'}{(\nu\gamma)C \overset{\ell}{\Rightarrow_p} (\nu\gamma)C'} \qquad \frac{C \overset{\ell}{\Rightarrow_p} C'}{(\nu n : T)C \overset{\ell}{\Rightarrow_p} (\nu n : T)C'} \qquad \frac{C \overset{\ell}{\Rightarrow_p} C'}{C \| C'' \overset{\ell}{\Rightarrow_p} C' \| C''} \tag{12}$$

Fig. 2. Semantics of VGR

freshness anymore as the name α, say, is fixed in the function type:

$$\{\}; \mathsf{Unit} \to \mathsf{Chan}\ \alpha; \{\alpha : ?\mathsf{Int}.?\mathsf{Int}.!\mathsf{Int}.S'\}$$

In consequence, VGR cannot invoke `acceptAdd` twice in a row as the second invocation would result in an ill-formed environment that contains two specifications for channel α.

LFST elides this issue, again, by not tracking channel identities.

3 VGR: Imperative Session Types

Figure 1 defines the syntax of VGR [27]. Notably, expressions t are in A-normal form and types distinguish between data types D and channels because two different sets of typing rules govern sending and receiving of data vs. sending and receiving a channel. We already used this syntax informally in the examples.

$$\textsc{C-Const}\ \ \Gamma; () \mapsto \mathsf{Unit} \qquad \textsc{C-Chan}\ \ \Gamma; \gamma^p \mapsto \mathsf{Chan}\ \gamma^p \qquad \textsc{C-Var}\ \ \Gamma, x : T; x \mapsto T$$

$$\textsc{C-Abs}\ \ \frac{\Gamma, x : T; \Sigma; e \mapsto \Sigma_1; U; \Sigma_2}{\Gamma; \lambda x.e \mapsto (\Sigma; T \to U; \Sigma_1, \Sigma_2)}$$

Fig. 3. Value typing rules of VGR

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

$$\textsc{C-Val}\ \ \frac{\Gamma; v \mapsto T}{\Gamma; \Sigma; v \mapsto \Sigma; T; \varnothing} \qquad \textsc{C-App}\ \ \frac{\Gamma; v \mapsto (\Sigma; T \to U; \Sigma') \qquad \Gamma; v' \mapsto T}{\Gamma; \Sigma, \Sigma''; v\, v' \mapsto \Sigma''; U; \Sigma'}$$

$$\textsc{C-Fork}\ \ \frac{\Gamma; \Sigma; t_1 \mapsto \Sigma_1; T_1; \{\} \qquad \Gamma; \Sigma_1; t_2 \mapsto \Sigma_2; T_2; \{\}}{\Gamma; \Sigma; (\mathsf{fork}\, t_1; t_2) \mapsto \Sigma_2; T_2; \{\}}$$

Fig. 4. Expression typing rules of VGR (excerpt)

We omit choices as they present no significant problem and as they can be simulated using channel passing. We also omit the standard congruence rules for processes and silently apply reduction rules up to congruence: parallel composition is a commutative monoid and the ν-binders admit scope extrusion.

Figure 2 defines the semantics of VGR. We use a slightly different, but equivalent definition than in the literature. We define evaluation contexts $E, F ::= \square \mid \mathsf{let}\, x = E\, \mathsf{in}\, t$ that are used in many of the rules. This formulation avoids the commuting conversion rule R-Let in the literature and fixes an issue with the original reduction relation.[4] We distinguish between expression reduction $\overset{\ell}{\Rightarrow}_e$ and process reduction $\overset{\ell}{\Rightarrow}_p$, both of which are tagged with a label ℓ. This label indicates the effect of the reduction and it ranges over

$$\ell ::= \mathsf{accept} \mid \mathsf{send} \mid \mathsf{new} \mid \mathsf{fork} \mid \tau \qquad \text{processes}$$
$$\ell ::= \mathsf{accept}\gamma \mid \mathsf{request}\gamma \mid \gamma?v \mid \gamma!v \mid \tau \qquad \text{expressions}$$

where τ stands for effect freedom and can be omitted. Labeled expression reductions are paired with their counterpart at the process level as familiar from process calculi [13], that is, $\gamma?v$ ($\gamma!v$) stand for receiving (sending) v on γ which resolves to label send at the process level (9). Similarly, $\mathsf{accept}\gamma$ ($\mathsf{request}\gamma$) stands for accepting (requesting) a connection on fresh channel γ and resolves to label accept at the process level.

[4] $\mathsf{let}\, x = \mathsf{fork}\, t; t'\, \mathsf{in}\, t''$ is stuck in the original work [27].

Constants	$k ::= \mathsf{fix} \mid \mathsf{fork} \mid \mathsf{send} \mid \mathsf{receive} \mid \mathsf{accept} \mid \mathsf{request} \mid \mathsf{new}$
Expressions	$e ::= x \mid \alpha \mid k \mid () \mid \lambda x.e \mid e\ e \mid (e, e) \mid \mathsf{let}\,(x, y) = e\,\mathsf{in}\,e$
	$\mid \{\} \mid \{\alpha = e\} \mid e \cdot e \mid e.\alpha \mid \mathsf{split}^*\, e, \alpha^*$
Configurations	$C ::= \langle e \rangle \mid C \| C \mid (\nu\gamma\delta)C \mid (\nu n)C$
Types	$t ::= s \mid [s] \mid \mathsf{Unit} \mid t \to t \mid t \multimap t \mid t \otimes t \mid \{r\}$
SessionTypes	$s ::= ?t.s \mid !t.s \mid \mathsf{End}$
Rows	$r ::= \varnothing \mid r, \alpha : t$
Environments	$\Gamma ::= \varnothing \mid \Gamma, \alpha : t \mid \Gamma, x : t$

Fig. 5. Syntax of LFST

Typing for VGR comes in three parts: value typing $\Gamma; v \mapsto T$ in Figure 3, expression typing $\Gamma; \Sigma; e \mapsto \Sigma'; T; \Sigma''$ in Figure 4, and configuration typing $\Gamma; \Sigma; C \mapsto \Sigma'$ (omitted). The value typing judgment relates an environment Γ and a value v to a type T. The expression typing judgment is very similar to a type state system. It relates a typing environment Γ, an incoming channel environment Σ, and an expression to an environment $\Sigma' \subseteq \Sigma$ which contains the channels not used by e, the type T, and the outgoing channel environment Σ''. Σ'' contains typings for channels that have been used by e or created by e. The configuration typing relates Γ, incoming Σ, and configuration C with $\Sigma' \subseteq \Sigma$ which contains the channels not used by C.

4 Linear Functional Session Types

On the functional side, we consider an extension of a synchronous variant of the LFST calculus [7] by linear records with disjoint concatenation. Figure 5 gives its syntax. The constants k and the first line of the expression grammar are taken from the literature. The second line of the expression grammar is new and defines operations on linear records. The empty record is $\{\}$, $\{\alpha = e\}$ constructs a singleton record with field α given by e, $e_1 \cdot e_2$ is the disjoint concatenation of records e_1 and e_2, $e.\alpha$ projects field α out of the record e and returns a pair of the contents of the field and the remaining record, $\mathsf{split}^*\, e, \alpha^*$ generalizes splitting to a list of names α^* and returns a pair of two records, one with the fields α^* and the other with the remaining fields.

A configuration C can be a single thread, two configurations running in parallel, a channel abstraction binding the two ends to γ and δ, or an access point abstraction $(\nu n)C$. The latter is a straightforward addition to LFST, which assumes the existence of globally known access points.

Metavariable t ranges over types, s ranges over session types, and r ranges over rows, which are lists of bindings of names to types.

$$\mathrm{unr}\,\mathsf{End} \qquad \mathrm{unr}\,([s]) \qquad \mathrm{unr}\,\mathsf{Unit} \qquad \frac{\mathrm{unr}\,t_1 \qquad \mathrm{unr}\,t_2}{\mathrm{unr}\,(t_1 \otimes t_2)} \qquad \mathrm{unr}\,(t_1 \to t_2) \qquad \frac{(\forall \alpha : t \in r)\ \mathrm{unr}\,t}{\mathrm{unr}\,\{r\}}$$

$$\text{T-Fork}\ \frac{\Gamma \vdash e : \mathsf{Unit}}{\Gamma \vdash \mathsf{fork}\,e : \mathsf{Unit}} \qquad \text{T-Send}\ \frac{\Gamma = \Gamma_1 + \Gamma_2 \qquad \Gamma_1 \vdash e_1 : t \qquad \Gamma_2 \vdash e_2 : !t.s}{\Gamma \vdash \mathsf{send}\,e_1\,\mathsf{on}\,e_2 : s}$$

$$\text{T-Recv}\ \frac{\Gamma \vdash e : ?t.s}{\Gamma \vdash \mathsf{receive}\,e : t \otimes s} \qquad \text{T-New}\ \frac{\mathrm{unr}\,\Gamma}{\Gamma \vdash \mathsf{new}\,s : [s]} \qquad \text{T-Accept}\ \frac{\Gamma \vdash e : [s]}{\Gamma \vdash \mathsf{accept}\,e : s} \qquad \text{T-Request}\ \frac{\Gamma \vdash e : [s]}{\Gamma \vdash \mathsf{request}\,e : \overline{s}}$$

$$\text{T-Emp}\ \frac{\mathrm{unr}\,\Gamma}{\Gamma \vdash \{\} : \{\}} \qquad \text{T-Single}\ \frac{\Gamma \vdash e : t}{\Gamma \vdash \{\alpha = e\} : \{\alpha : t\}} \qquad \text{T-SplitRecord}\ \frac{\Gamma \vdash e : \{r_1 + r_2\} \qquad \mathrm{dom}(r_1) = \alpha^*}{\Gamma \vdash \mathsf{split}^*\,e, \alpha^* : \{r_1\} \otimes \{r_2\}}$$

$$\text{T-Concat}\ \frac{\Gamma = \Gamma_1 + \Gamma_2 \qquad \Gamma_1 \vdash e_1 : \{r_1\} \qquad \Gamma_2 \vdash e_2 : \{r_2\} \qquad r_1 \nparallel r_2}{\Gamma \vdash e_1 \cdot e_2 : \{r_1, r_2\}} \qquad \text{T-Field}\ \frac{\Gamma \vdash e : \{r, \alpha : t\}}{\Gamma \vdash e.\alpha : t \otimes \{r\}}$$

Fig. 6. Typing rules for communication primitives and record operations in LFST

Figure 6 recalls the definition of the predicate $\mathrm{unr}\,t$ for unrestricted types, which we lift pointwise to typing environments. It also contains the well-known typing rules for the communication primitives as well as for the record fragment of LFST. The rule T-Emp typechecks the empty record with the premise $\mathrm{unr}\,\Gamma$ which states that Γ only contains unrestricted types. The rule T-Single is unsurprising. Premise $\Gamma = \Gamma_1 + \Gamma_2$ of rule T-Concat splits the incoming environment Γ so that bindings to a linear type end up either in Γ_1 or in Γ_2. Premise $r_1 \nparallel r_2$ states that rows r_1 and r_2 are disjoint, which means they bind different field names. Under these assumptions the (disjoint) concatenation of records e_1 and e_2 is accepted.

The rules for field access and splitting of the record generalize the elimination rule for linear pairs. Rule T-Field shows that fields access singles out the field named α. Its content is paired up with a record comprising the remaining fields. Linearity of the record's content is preserved as the pair is also linear. Rule T-SplitRecord is similar, but splits its subject e according to a list α^* of names which must be present in e. The result is a linear pair of two records. We consider an empty record to be unrestricted so that we can drop it if needed.

The remaining typing rules are taken from the original paper [7] (cf. also Figure 10). We modified the operational semantics to perform synchronous communication and to fit with the labeled transition style used for VGR in Section 3. Its formalization is omitted because of its similarity to VGR.

$$\text{C-App}\quad \langle\!\langle \frac{\Gamma; v \mapsto (\Sigma; T \to U; \Sigma') \qquad \Gamma; v' \mapsto T}{\Gamma; \Sigma, \Sigma''; v\, v' \mapsto \Sigma''; U; \Sigma'} \rangle\!\rangle_\sigma = \langle\!\langle v \rangle\!\rangle\, \langle\!\langle v' \rangle\!\rangle\, \sigma$$

$$\text{C-SendS}\quad \langle\!\langle \frac{\Gamma; v \mapsto \mathsf{Chan}\ \beta \qquad \Gamma; v' \mapsto \mathsf{Chan}\ \alpha}{\Gamma; \Sigma, \alpha : !S'.S, \beta : S'; \mathsf{send}\, v\, \mathsf{on}\, v' \mapsto \Sigma; \mathsf{Unit}; \alpha : S} \rangle\!\rangle_\sigma = \begin{array}{l} \mathsf{let}\,(c, \sigma) = \sigma.\alpha\,\mathsf{in} \\ \mathsf{let}\,(p, \sigma) = \sigma.\beta\,\mathsf{in} \\ \mathsf{let}\, c = \mathsf{send}\, p\, \mathsf{on}\ c\,\mathsf{in} \\ ((), \sigma \cdot \{\alpha = c\}) \end{array}$$

$$\text{C-Fork}\quad \langle\!\langle \frac{\Gamma; \Sigma; t_1 \mapsto \Sigma_1; T_1; \{\} \qquad \Gamma; \Sigma_1; t_2 \mapsto \Sigma_2; T_2; \{\}}{\Gamma; \Sigma; (\mathsf{fork}\, t_1; t_2) \mapsto \Sigma_2; T_2; \{\}} \rangle\!\rangle_\sigma = \begin{array}{l} \mathsf{let}\,(\sigma_1, \sigma_2) = \mathsf{split}^\star\, \sigma, \mathrm{dom}(\Sigma_1 \setminus \Sigma_2)\,\mathsf{in} \\ \mathsf{let}\ \_ = \mathsf{fork}\, \langle\!\langle t_1 \rangle\!\rangle_{\sigma_1}\,\mathsf{in} \\ \langle\!\langle t_2 \rangle\!\rangle_{\sigma_2} \end{array}$$

$$\text{C-New}\quad \langle\!\langle \Gamma; \Sigma; \mathsf{new}\, S \mapsto \Sigma; [S]; \varnothing \rangle\!\rangle_\sigma = (\mathsf{new}\, \langle\!\langle S \rangle\!\rangle, \sigma)$$

Fig. 7. Translation of expressions and threads (excerpt)

$$\text{C-Thread}\quad \langle\!\langle \frac{\vec{x} : [\vec{S}]; \Sigma; t \mapsto \Sigma'; T; \{\}}{\vec{x} : [\vec{S}]; \Sigma; \langle t \rangle \mapsto \Sigma'} \rangle\!\rangle = \langle \mathsf{let}\, \sigma = \{\vec{\gamma} = \vec{\gamma}\}\, \mathsf{in}\, \langle\!\langle t \rangle\!\rangle_\sigma \rangle \quad \textit{where}\ \vec{\gamma} = \mathrm{dom}(\Sigma' \setminus \Sigma)$$

Fig. 8. Translation of configurations (excerpt)

5 Translation: Imperative to Functional

As a first step, we discuss the translation of the imperative session type calculus VGR into the linear functional session type calculus LFST-rec. The extension with record types is not essential, but it makes stating the translation more accessible. All records could be elided by replacing them with suitably nested pairs and mapping record labels to indices.

The translation from VGR to LFST-rec is type driven, i.e., it is a translation of typing derivations. The gist of the approach is to translate VGR expressions into a parameterized linear state transformer monad. It is parameterized in the sense of Atkey [2] because the type of the state changes during the computation.

In particular, we map derivations for VGR value typing, VGR expression typing, and VGR configuration typing such that the following typing preservation results hold. For brevity, we indicate the translation with $\langle\!\langle e \rangle\!\rangle$ and $\langle\!\langle C \rangle\!\rangle$ where the arguments are really the typing derivations for e and C, respectively. The translations on types $\langle\!\langle T \rangle\!\rangle$, environments $\langle\!\langle \Gamma \rangle\!\rangle$, $\langle\!\langle \Sigma \rangle\!\rangle$, and values $\langle\!\langle v \rangle\!\rangle$ are

Translation of types

$$
\begin{aligned}
\langle\!\langle !T.S \rangle\!\rangle &= !\langle\!\langle T \rangle\!\rangle.\langle\!\langle S \rangle\!\rangle \\
\langle\!\langle ?T.S \rangle\!\rangle &= ?\langle\!\langle T \rangle\!\rangle.\langle\!\langle S \rangle\!\rangle \\
\langle\!\langle \mathsf{End} \rangle\!\rangle &= \mathsf{End} \\
\langle\!\langle \mathsf{Unit} \rangle\!\rangle &= \mathsf{Unit} \\
\langle\!\langle [S] \rangle\!\rangle &= [\langle\!\langle S \rangle\!\rangle] \\
\langle\!\langle \mathsf{Chan}\ \alpha \rangle\!\rangle &= \mathsf{Unit} \\
\langle\!\langle \Sigma_1; T_1 \to T_2; \Sigma_2 \rangle\!\rangle &= \langle\!\langle T_1 \rangle\!\rangle \to \{\langle\!\langle \Sigma_1 \rangle\!\rangle\} \to \\
&\quad (\langle\!\langle T_2 \rangle\!\rangle \times \{\langle\!\langle \Sigma_2 \rangle\!\rangle\})
\end{aligned}
$$

Translation of values

$$
\begin{aligned}
\langle\!\langle () \rangle\!\rangle &= () \\
\langle\!\langle \gamma^{\pm} \rangle\!\rangle &= () \\
\langle\!\langle x \rangle\!\rangle &= x \\
\langle\!\langle \lambda x.e \rangle\!\rangle &= \lambda x.\lambda\sigma.\langle\!\langle e \rangle\!\rangle_\sigma
\end{aligned}
$$

Fig. 9. Type translation

homomorphic by induction on the syntax (see extended version), except for

$$
\begin{aligned}
\langle\!\langle \mathsf{Chan}\ \alpha \rangle\!\rangle &= \mathsf{Unit} & \langle\!\langle \gamma^{\pm} \rangle\!\rangle &= () \\
\langle\!\langle \Sigma_1; T_1 \to T_2; \Sigma_2 \rangle\!\rangle &= \langle\!\langle T_1 \rangle\!\rangle \to \{\langle\!\langle \Sigma_1 \rangle\!\rangle\} \to (\langle\!\langle T_2 \rangle\!\rangle \times \{\langle\!\langle \Sigma_2 \rangle\!\rangle\}) & \langle\!\langle \lambda x.e \rangle\!\rangle &= \lambda x.\lambda\sigma.\langle\!\langle e \rangle\!\rangle_\sigma
\end{aligned}
$$

Proposition 1 (Typing Preserving Translation).

$$
\textsc{Preserve-Value}\ \frac{\Gamma; v \mapsto T}{\langle\!\langle \Gamma \rangle\!\rangle \vdash \langle\!\langle v \rangle\!\rangle : \langle\!\langle T \rangle\!\rangle} \qquad \textsc{Preserve-Expression}\ \frac{\Gamma; \Sigma; e \mapsto \Sigma_1; T; \Sigma_2}{\langle\!\langle \Gamma \rangle\!\rangle, \sigma : \{\langle\!\langle \Sigma \setminus \Sigma_1 \rangle\!\rangle\} \vdash \langle\!\langle e \rangle\!\rangle_\sigma : \langle\!\langle T \rangle\!\rangle \times \{\langle\!\langle \Sigma_2 \rangle\!\rangle\}} \qquad \textsc{Preserve-Config}\ \frac{\Gamma; \Sigma; C \mapsto \Sigma_1}{\langle\!\langle \Gamma \rangle\!\rangle, \langle\!\langle \Sigma \setminus \Sigma_1 \rangle\!\rangle \vdash \langle\!\langle C \rangle\!\rangle}
$$

These statements are proved by mutual induction on the derivations of the VGR judgments in the premises. The VGR typing judgments for expressions and configurations pass through unused channels (in Σ_1) in the style of leftover typings [1]. While this style is convenient for some proofs, it cannot be used for the translation as it fails when trying to translate the term $\mathsf{fork}\, t_1; t_2$. The first premise of its typing rule C-Fork is $\Gamma; \Sigma; t_1 \mapsto \Sigma_1; T_1; \{\}$, which says that executing t_1 consumes some of the incoming channels Σ and does not touch the ones in Σ_1. The second premise $\Gamma; \Sigma_1; t_2 \mapsto \Sigma_2; T_2; \{\}$ picks up Σ_1 and demands that t_2 consumes all its channels. This pattern does not work for the translation, which is based on explicit channel passing: if we would pass all channels in Σ to t_1, which is forked as a new thread, there would be no way to obtain the leftover channels Σ_1 after thread t_1 has finished. Moreover, these channels have to be available for t_2 even before t_1 has finished! The same issue arises with translating the parallel composition of two configurations. For that reason, in LFST-rec the translated expressions and configurations are supplied with exactly the channels needed. Hence the necessity to compute the needed channels as the difference $\Sigma \setminus \Sigma_1$.

Figure 9 contains the details of the type translation, the translation of environments, and the translation of values. The only interesting case of the type

translation is the one for function types, which maps a function to a Kleisli arrow in a linear, parameterized state monad. The incoming and outgoing channel environments are mapped to the incoming and outgoing state record types. The other observation is that any channel type is mapped to the unit type.

The translation of values has two interesting cases. A channel value is mapped to unit because channels are handled on the type level and channel references are resolved by accessing the corresponding field of the state record. Lambdas obtain an extra argument σ for the incoming state. The body of a lambda is translated by the expression translation which is indexed by the incoming state record σ and returns a pair of the result and the outgoing state record.

Figure 7 shows select cases from the translation of expressions that demonstrate the role of the record operations. The conclusion of Preserve-Expression shows that an expression is translated to a linear state transformer as in the translation of the function type.

Figure 8 contains the translation of the C-Thread configuration rule, which is the only interesting case. Its reifies the channels that are used in the thread by collecting them in a record σ and injecting that record as the initial state of the state monad. This record is transformed by the expression translation which returns a pair of the return value of type $\langle\!\langle T \rangle\!\rangle$ and the final record of type $\{\langle\!\langle \varnothing \rangle\!\rangle\}$. It is easy to see that this pair is unrestricted because the translation of a type T is generally an unrestricted type and the empty record is also unrestricted.

We would like the translation to induce a simulation in that each step of a typed VGR configuration C gives rise to one or more steps in its translation $\langle\!\langle C \rangle\!\rangle$ in LFST-rec. Unfortunately, the situation is not that simple because administrative reductions involving the state get in the way.

Proposition 2 (Simulation). *If $\Gamma; \Sigma; C \mapsto \Sigma'$ and $C \overset{\ell}{\Rightarrow}_p C'$ in VGR, then there is a configuration $\overline{C}$ in LFST-rec such that $\langle\!\langle C \rangle\!\rangle \overset{\ell}{\rightarrow}_p{}^+ \overline{C}$ and $\langle\!\langle C' \rangle\!\rangle \overset{\tau}{\rightarrow}_p{}^+ \overline{C}$.*

6 Translation: Functional to Imperative

For the backwards translation we consider LFST programs without records and we informally extend the expression language of VGR with pairs—analogous to LFST, but unrestricted.

We first consider an untyped translation that demonstrates that the calculi are equally expressive. Then we restrict the type system of LFST to identify a subset on which the translation preserves typing.

6.1 Untyped Translation

In a first approximation, the backwards translation might map the send and receive operations naively as follows.

$$(\!|\mathsf{send}\, e_1\, \mathsf{on}\, e_2|\!) = \mathsf{let}\, x = (\!|e_1|\!)\, \mathsf{in}\, \mathsf{let}\, y = (\!|e_2|\!)\, \mathsf{in}\, \mathsf{let}\, z = \mathsf{send}\, x\, \mathsf{on}\, y\, \mathsf{in}\, y \tag{13}$$

$$(\!|\mathsf{receive}\, e|\!) = \mathsf{let}\, y = (\!|e|\!)\, \mathsf{in}\, \mathsf{let}\, x = \mathsf{receive}\, y\, \mathsf{in}\, (x, y) \tag{14}$$

This mapping, extended to the rest of LFST, yields a program in A-normal form to fit with VGR's syntactic restrictions. The functional send operation returns the updated channel, so we have to duplicate the channel reference y in its image in VGR. Similarly, the functional receive operation returns a pair of the received value and the updated channel, so the translation needs to construct a pair from the received value and the updated channel y.

However, to prove a tight relation between reduction in LFST and VGR, we need to be more careful to avoid administrative reductions. For example, if e in (14) is already a value, then the inserted $\mathsf{let}\, y = (\!|e|\!)\, \mathsf{in}\, \ldots$ is gratuitous and results in an extra (administrative) reduction in VGR.

This phenomenon is known since Plotkin's treatise of the CPS translation [16]. Hence, we factor the backwards translation in two steps. The first step transforms the LFST program to A-normal form using an approach due to Sabry and Felleisen [19]. This transformation is known to give rise to a strong operational correspondence (a reduction correspondence [20]), it is typing preserving, and it is applicable to LFST because it preserves linearity. The definition for this translation $\llbracket e \rrbracket$ may be found in the extended version.

This refined ANF translation is compatible with evaluation because it is compatible with values, evaluation contexts, and substitution.

Proposition 3 (ANF Simulation).

1. If $e \to_e e'$, *then* $\llbracket e \rrbracket \to_e^+ \llbracket e' \rrbracket$. *2. If* $C \xrightarrow{\ell}_p C'$, *then* $\llbracket C \rrbracket \xrightarrow{\ell}_p^+ \llbracket C' \rrbracket$.

The second step is the expression translation $(\!|e|\!)$ from LFST-ANF to VGR. This translation is very simple because the source calculus is already in A-normal form. The idea of the translation as stated at the beginning of this section is clearly reflected in the first two lines of the expression translation $(\!|e|\!)$. The remaining cases work homomorphically (see extended version).

$$\begin{aligned}(\!|\mathsf{send}\, v\, \mathsf{on}\, w|\!) &= \mathsf{let}\, z = \mathsf{send}\, (\!|v|\!)\, \mathsf{on}\, (\!|w|\!)\, \mathsf{in}\, (\!|w|\!)\\ (\!|\mathsf{receive}\, v|\!) &= \mathsf{let}\, x = \mathsf{receive}\, (\!|v|\!)\, \mathsf{in}\, (x, (\!|v|\!))\\ (\!|\mathsf{fork}\, e|\!) &= \mathsf{fork}\, (\!|e|\!);\, ()\end{aligned}$$

This setup establishes a tight connection between LFST-ANF and VGR, because the translation preserves values, evaluation contexts, and substitution.

Proposition 4 (Backwards simulation). *Let* e, e' *and* C, C' *be expressions and configurations in LFST-ANF.*

1. If $e \to_e e'$, *then* $(\!|e|\!) \Rightarrow_e (\!|e'|\!)$. *2. If* $C \xrightarrow{\ell}_p C'$, *then* $(\!|C|\!) \Rightarrow_p^{\ell +} (\!|C'|\!)$.

Putting the results for the two steps together, we obtain the desired tight simulation result by composing Propositions 3 and 4.

Proposition 5 (Full Backwards Simulation). *Suppose that* e, e' *and* C, C' *are expressions and configurations in LFST.*

1. If $e \to_e e'$, *then* $(\!|\llbracket e \rrbracket|\!) \Rightarrow_e^+ (\!|\llbracket e' \rrbracket|\!)$. *2. If* $C \xrightarrow{\ell}_p C'$, *then* $(\!|\llbracket C \rrbracket|\!) \Rightarrow_p^{\ell +} (\!|\llbracket C' \rrbracket|\!)$.

$$\text{T-LamU'}\quad \frac{\text{unr}\,\Gamma \qquad \Gamma, x : t_2 \vdash' e : t_1/\Sigma_0 \mapsto \Sigma_1}{\Gamma \vdash' \lambda x.e : t_2 \to^{\Sigma_0 \mapsto \Sigma_1} t_1/\Sigma \mapsto \Sigma}$$

$$\text{T-App'}\quad \frac{\Gamma = \Gamma_1 + \Gamma_2 \qquad \Gamma_1 \vdash' e_1 : t_2 \multimap^{\Sigma_2 \mapsto \Sigma_3} t_1/\Sigma_0 \mapsto \Sigma_1 \qquad \Gamma_2 \vdash' e_2 : t_2/\Sigma_1 \mapsto \Sigma_2, \Sigma_2'}{\Gamma \vdash' e_1\, e_2 : t_1/\Sigma_0 \mapsto \Sigma_3, \Sigma_2'}$$

$$\text{T-Send''}\quad \frac{\Gamma = \Gamma_1 + \Gamma_2 \qquad \Gamma_1 \vdash' e_1 : {s'}_\beta/\Sigma \mapsto \Sigma' \qquad \Gamma_2 \vdash' e_2 : (!s'.s)_\alpha/\Sigma' \mapsto \Sigma'', \alpha : !t.s, \beta : s'}{\Gamma \vdash' \mathsf{send}\, e_1\, \mathsf{on}\, e_2 : s_\alpha/\Sigma \mapsto \Sigma'', \alpha : s}$$

$$\text{T-Recv''}\quad \frac{\Gamma \vdash' e : (?s'.s)_\alpha/\Sigma \mapsto \Sigma', \alpha : ?s'.s}{\Gamma \vdash' \mathsf{receive}\, e : {s'}_\beta \otimes s_\alpha/\Sigma \mapsto \Sigma', \alpha : s, \beta : s'} \qquad \text{T-New'}\quad \frac{\text{unr}\,\Gamma}{\Gamma \vdash' \mathsf{new}\, s : [s]/\Sigma \mapsto \Sigma}$$

Fig. 10. Typing rules for LFST-EFF (excerpt)

6.2 Typed Backwards Translation

One can add the necessary information for a typed backwards translation to the type system of LFST, at the price of making it more restrictive. We start with an informal review of the requirements.

As VGR tracks channel identities, they have to be reflected in the LFST type systems. Following Padovani [14], we tag session types as in s_α consisting of a session type s tagged with an identity α.

The function type in VGR specifies a transformation on the channels that are implicitly or explicitly affected by the function. Hence, we must augment the LFST type system with tracking the identities of channels, on which the program performs an effect. To this end, we equip LFST with a suitable sequential effect system [8]. It distinguishes between incoming and outgoing channels, Σ_i and Σ_o, which are also reflected in the latent effect on the function arrow.

$$\Gamma \vdash' e : t/\Sigma_i \mapsto \Sigma_o$$

We define tagged session types by adding an identity tag α to all session types and augmenting function types with a set of uniquely tagged sessions. We carve out a set of data types d, which can be transmitted in VGR programs. Hence, session types proper (denoted by s) are a subset of LFST's session types.

Types	$t ::= s_\alpha \mid [s] \mid \mathsf{Unit} \mid t \to^{\Sigma \mapsto \Sigma} t \mid t \multimap^{\Sigma \mapsto \Sigma} t \mid t \otimes t$
Data	$d ::= [s] \mid \mathsf{Unit} \mid t \to^{\Sigma \mapsto \Sigma} t \mid t \multimap^{\Sigma \mapsto \Sigma} t$
Sessions	$s ::= ?d.s \mid !d.s \mid ?s.s \mid !s.s \mid \mathsf{End}$

Using mostly standard effect typing rules (see Figure 10), we show that effect typing is a proper restriction of LFST typing.

$$
\begin{array}{lll}
(\!|!t.s|\!) = !(\!|t|\!).(\!|s|\!) & (\!|[s]|\!) = [(\!|s|\!)] & (\!|t_1 \to^{\Sigma_1 \mapsto \Sigma_2} t_2|\!) = \Sigma_1; (\!|t_1|\!) \to (\!|t_2|\!); \Sigma_2 \\
(\!|?t.s|\!) = ?(\!|t|\!).(\!|s|\!) & (\!|s_\alpha|\!) = \mathsf{Chan}\ \alpha & (\!|t_1 \multimap^{\Sigma_1 \mapsto \Sigma_2} t_2|\!) = \Sigma_1; (\!|t_1|\!) \to (\!|t_2|\!); \Sigma_2 \\
(\!|\mathsf{End}|\!) = \mathsf{End} & (\!|\mathsf{Unit}|\!) = \mathsf{Unit} & (\!|t_1 \otimes t_2|\!) = (\!|t_1|\!) \times (\!|t_2|\!)
\end{array}
$$

Fig. 11. Type translation from LFST-EFF to VGR

Lemma 1 (Conservative Extension). *$\Gamma \vdash' e : t/\Sigma \mapsto \Sigma'$ implies $|\Gamma| \vdash e : |t|$.*

The translation to ANF does not affect LFST typing with effects.

Lemma 2 (ANF Compatible). *Suppose that $\Gamma \vdash' e : t/\Sigma \mapsto \Sigma'$. Then $\Gamma \vdash' [\![e]\!] : t/\Sigma \mapsto \Sigma'$.*

Figure 11 contains the backwards translation for types. An α-tagged session type turns into the channel type $\mathsf{Chan}\ \alpha$ and the effect annotation on function types gets mapped to the before and after environments in VGR function types.

Proposition 6 (Typing Preservation (Backwards)).
Suppose that $\Gamma \vdash' e : t/\Sigma_1 \mapsto \Sigma_2$ for some expression e in LFST-ANF. Then for all Σ such that $\Sigma \# \Sigma_1$ and $\Sigma \# \Sigma_2$, $(\!|\Gamma|\!); \Sigma, \Sigma_1; (\!|e|\!) \mapsto (\!|t|\!); \Sigma; \Sigma_2$.

7 Related Work

Pucella and Tov [17] give an embedding of a session type calculus in Haskell. Like our translation, their embedding relies on a parameterized monad, which is layered on top of the IO monad using phantom types. Linearity is enforced by the monad abstraction. Multiple channels are implemented by stacking so that channel names are de Bruijn indices. Stacking only happens at the (phantom) type level, so that stack rearrangement has no operational consequences. The paper comes with a formalization and a soundness proof of the implementation. Sackman and Eisenbach [21] also encode session types for a single channel in Haskell using an indexed (parameterized) monad.

Imai and coworkers [12] propose an encoding of binary session-based communication as a library in OCaml. This library is based on an indexed state monad that maintains the current state of a set of channels in a tuple. Channel names are encoded by lenses operating on this state and operations an a channel change the index type at the position indicated by the lens. The programming style resembles VGR, but it is explicitly monadic. The monad and its type indexing are closely related to our encoding, which is linear by typing.

Another line of work on session types is based on process calculi obtained through the Curry-Howard correspondence applied to fragments of linear logic [4, 3, 5]. The resulting programs have an imperative flavor as they are based on process calculus. The correspondence structures communication as a string of interactions on a channel name. This channel name "changes type" by rebinding

at each communication operation. There is a monadic embedding of this approach into a pure functional language [26]. In this stratified language, processes are snippets of imperative code encapsulated as first-class monadic values into the functional language. These values can be plugged into a process term by a suitable version of the monadic bind operation. Processes may transmit channel names or values from the functional stratum. Processes have the imperative flavor as already mentioned. It would be interesting future work to relate this line of work with the correspondence developed in the present paper.

Alias types [23] presents 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. 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 [18] and ensures that locations are unique. Analogous to our channel types, pointers in the alias types system can be duplicated and have a singleton type indicating their store location. Alias types also include non-linear constraints, which are not required in our system.

8 Conclusion

Disregarding types, the imperative and functional session calculi are equally powerful. But typing is the essence of a session calculus so that the imperative calculus is strictly less expressive. Two issues are responsible for the limitations.

1. Identity tracking for channels restricts the usability of functional abstraction. With explicit identity, functions are fixed to specific channels.
2. Having different typing rules for sending channels and sending (other) data gives rise to lack of abstraction and hinders modularity. Higher-order channel passing has subtle problems that render a transmitted channel useless.

Our results suggest that the type system severely restricts VGR's expressiveness. Hence, it is an interesting future work to extend VGR's type system such that there are type and semantics preserving translations in both directions.

References

1. Allais, G.: Typing with leftovers - A mechanization of intuitionistic multiplicative-additive linear logic. In: 23rd International Conference on Types for Proofs and Programs, TYPES 2017, May 29-June 1, 2017, Budapest, Hungary. LIPIcs, vol. 104, pp. 1:1–1:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017). https://doi.org/10.4230/LIPIcs.TYPES.2017.1
2. Atkey, R.: Parameterised notions of computation. J. Funct. Program. **19**(3-4), 335–376 (2009). https://doi.org/10.1017/S095679680900728X
3. Balzer, S., Toninho, B., Pfenning, F.: Manifest deadlock-freedom for shared session types. In: Caires, L. (ed.) Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Prague, Czech Republic. LNCS, vol. 11423, pp. 611–639. Springer (2019). https://doi.org/10.1007/978-3-030-17184-1_22
4. Caires, L., Pfenning, F.: Session types as intuitionistic linear propositions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings. LNCS, vol. 6269, pp. 222–236. Springer (2010). https://doi.org/10.1007/978-3-642-15375-4_16
5. Das, A., Pfenning, F.: Session types with arithmetic refinements. In: Konnov, I., Kovács, L. (eds.) 31st International Conference on Concurrency Theory, CONCUR 2020, September 1-4, 2020, Vienna, Austria (Virtual Conference). LIPIcs, vol. 171, pp. 13:1–13:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020). https://doi.org/10.4230/LIPIcs.CONCUR.2020.13
6. Fowler, S., Lindley, S., Morris, J.G., Decova, S.: Exceptional asynchronous session types: Session types without tiers. Proc. ACM Program. Lang. **3**(POPL), 28:1–28:29 (2019). https://doi.org/10.1145/3290341
7. Gay, S.J., Vasconcelos, V.T.: Linear type theory for asynchronous session types. J. Funct. Program. **20**(1), 19–50 (2010). https://doi.org/10.1017/S0956796809990268
8. Gordon, C.S.: A generic approach to flow-sensitive polymorphic effects. In: 31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain. LIPIcs, vol. 74, pp. 13:1–13:31. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017). https://doi.org/10.4230/LIPIcs.ECOOP.2017.13
9. Honda, K.: Types for dyadic interaction. In: CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings. LNCS, vol. 715, pp. 509–523. Springer (1993). https://doi.org/10.1007/3-540-57208-2_35
10. Hu, R., Kouzapas, D., Pernet, O., Yoshida, N., Honda, K.: Type-safe eventful sessions in Java. In: ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings. LNCS, vol. 6183, pp. 329–353. Springer (2010). https://doi.org/10.1007/978-3-642-14107-2_16
11. Hu, R., Yoshida, N.: Hybrid session verification through endpoint API generation. In: Fundamental Approaches to Software Engineering - 19th International Conference, FASE 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings. LNCS, vol. 9633, pp. 401–418. Springer (2016). https://doi.org/10.1007/978-3-662-49665-7_24
12. Imai, K., Yoshida, N., Yuen, S.: Session-OCaml: A session-based library with polarities and lenses. Sci. Comput. Program. **172**, 135–159 (2019). https://doi.org/10.1016/j.scico.2018.08.005

13. Milner, R.: Communicating and mobile systems - the Pi-calculus. Cambridge University Press (1999)
14. Padovani, L.: Context-free session type inference. In: 26th European Symposium on Programming, ESOP 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings. LNCS, vol. 10201, pp. 804–830. Springer (2017). https://doi.org/10.1007/978-3-662-54434-1_30
15. Padovani, L.: A simple library implementation of binary sessions. J. Funct. Program. **27**, e4 (2017). https://doi.org/10.1017/S0956796816000289
16. Plotkin, G.D.: Call-by-name, call-by-value and the lambda-calculus. Theor. Comput. Sci. **1**(2), 125–159 (1975). https://doi.org/10.1016/0304-3975(75)90017-1
17. Pucella, R., Tov, J.A.: 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. pp. 25–36. ACM (2008). https://doi.org/10.1145/1411286.1411290
18. Reynolds, J.C.: 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. pp. 55–74. IEEE Computer Society (2002). https://doi.org/10.1109/LICS.2002.1029817
19. Sabry, A., Felleisen, M.: Reasoning about programs in continuation-passing style. LISP Symb. Comput. **6**(3-4), 289–360 (1993)
20. Sabry, A., Wadler, P.: A reflection on call-by-value. ACM Trans. Program. Lang. Syst. **19**(6), 916–941 (1997). https://doi.org/10.1145/267959.269968
21. Sackman, M., Eisenbach, S.: Session types in Haskell updating message passing for the 21st century (2008), https://spiral.imperial.ac.uk:8443/handle/10044/1/5918
22. Scalas, A., Yoshida, N.: Lightweight session programming in Scala. In: 30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy. LIPIcs, vol. 56, pp. 21:1–21:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016). https://doi.org/10.4230/LIPIcs.ECOOP.2016.21
23. Smith, F., Walker, D., Morrisett, J.G.: Alias types. In: 9th European Symposium on Programming, ESOP 2000, Berlin, Germany, March 25 - April 2, 2000, Proceedings. LNCS, vol. 1782, pp. 366–381. Springer (2000). https://doi.org/10.1007/3-540-46425-5_24
24. Strom, R.E., Yemini, S.: Typestate: A programming language concept for enhancing software reliability. IEEE Trans. Software Eng. **12**(1), 157–171 (1986). https://doi.org/10.1109/TSE.1986.6312929
25. Takeuchi, K., Honda, K., Kubo, M.: 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. LNCS, vol. 817, pp. 398–413. Springer (1994). https://doi.org/10.1007/3-540-58184-7_118
26. Toninho, B., Caires, L., Pfenning, F.: Higher-order processes, functions, and sessions: A monadic integration. In: Felleisen, M., Gardner, P. (eds.) Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Rome, Italy. LNCS, vol. 7792, pp. 350–369. Springer (2013). https://doi.org/10.1007/978-3-642-37036-6_20
27. Vasconcelos, V.T., Gay, S.J., Ravara, A.: Type checking a multithreaded functional language with session types. Theor. Comput. Sci. **368**(1-2), 64–87 (2006). https://doi.org/10.1016/j.tcs.2006.06.028
28. Vasconcelos, V.T., Ravara, A., Gay, S.J.: Session types for functional multithreading. In: CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK, August 31 - September 3, 2004, Proceedings. LNCS, vol. 3170, pp. 497–511. Springer (2004). https://doi.org/10.1007/978-3-540-28644-8_32