Manifestly Phased Communication via Shared Session Types

Chuta Sano, Stephanie Balzer, and Frank Pfenning

Carnegie Mellon University

Abstract. Session types denote message protocols between concurrent processes, allowing a type-safe expression of inter-process communication. Although previous work demonstrate a well-defined notion of subtyping where processes have different perceptions of the protocol, these formulations were limited to linear session types where each channel of communication has a unique provider and client. In this paper, we extend subtyping to shared session types where channels can now have multiple clients instead of a single client. We demonstrate that this generalization can statically capture protocol requirements that span multiple phases of interactions of a client with a shared service provider, something not possible in prior proposals. Moreover, the phases are manifest in the type of the client.

1 Introduction

Session types prescribe bidirectional communication protocols between concurrent processes [15,16]. Variations of this type system were later given logical correspondences with *intuitionistic* [4] and *classical* [22] linear logic where proofs correspond to programs and cut reduction to communication. This correspondence mainly provided an interpretation of *linear session types*, which denote sessions with exactly one client and one provider. *Shared session types*, which encode communication between multiple clients and one provider, were proposed with a *sharing semantics* interpretation in a prior work [2]. Clients communicating along a shared channel follow an *acquire-release* discipline where they must first *acquire* exclusive access to the provider, communicate linearly, and then finally *release* the exclusive access, allowing other clients to acquire.

However, not all protocols that follow this acquire-release paradigm are safe; if a client that successfully acquires some shared channel of type A releases it at an unrelated type B, other clients that are blocked while trying to acquire will still see the channel as type A while the provider will see the channel as type B. To resolve this, we require an additional constraint that clients must release at the same type at which it acquired. This was formally expressed in [2] as the *equi-synchronizing* constraint, which statically verifies that session types encode communication which does not release at the wrong type. Although shared session types serve an important role in making session typed process calculi theory applicable to practical scenarios, we previously [19] showed that shared session types cannot express *phases*, or protocols across successive acquire-release cycles, due to the equi-synchronizing constraint being too restrictive (see Section 5).

We demonstrate that subtyping, first formalized in the session-typed process calculi setting by Gay and Hole [11], and its behavior across the two linear and shared modalities provide the groundwork for an elegant relaxation of the equi-synchronizing constraint, allowing for phases to be *manifest* in the session type. In message passing concurrency, subtyping allows a client and provider to safely maintain their own local views on the session type (or protocol) associated with a particular channel. Although previous papers [11,1] investigate subtyping in the purely linear session type setting, we found that extending these results to the linear and shared session type setting as in [2] yields very powerful results with both practical and theoretical significance.

In this paper, we build upon past results on subtyping and propose a formulation of subtyping compatible with shared session types. We in particular introduce the *subsynchronizing* constraint, a relaxation of the equi-synchronizing constraint.

The main contributions of this paper include:

- A full formalization of a subtyping relation for shared session types and their meta theory.
- The introduction of the subsynchronizing constraint, a relaxation of the equi-synchronizing constraint.
- Illustrations of practical examples in this richer type system, further bridging the gap between session-typed process calculi and practical programming languages.

The rest of the paper will proceed as follows: Section 2 provides a brief introduction to linear and shared session-typed message-passing concurrency. Section 3 demonstrates the inability for prior systems to express phasing and motivates our approach. Section 4 provides an introduction to linear subtyping along with an attempt to extend the relation to the shared setting. Section 5 introduces the notion of phasing and the subsynchronizing judgment. Section 6 presents a message passing concurrent system using our typesystem and the corresponding progress and preservation statements. Section 7 discusses related work. Section 8 concludes the paper with some points of discussion and future work.

An extended version of this paper is available as a technical report [20], containing detailed proofs, a complete formalization of the system, and more complex examples. This paper will focus on our advancements to the type system and key ideas while treating the syntax of the language and the operational interpretation informally.

2 Background

2.1 Linear Session Types

Based on the correspondence established between intuitionistic linear logic and the session-typed π-calculus [4,21] we can interpret a intuitionistic *linear* sequent

$$A_1, A_2, \ldots, A_n \vdash B$$

as the typing judgment for a process P by annotating the linear propositions with channel names:

$$\underbrace{a_1 : A_1, a_2 : A_2, \ldots, a_n : A_n}_{\Delta} \vdash P :: (b : B)$$

Interpreted as a typing judgment, we say that process P *provides* a session of type B along channel b while *using* channels $a_1, \ldots, a_n$ with session types $A_1, \ldots, A_n$, respectively. Interpreted as a sequent, we say that P is a proof of some proposition B with hypotheses $A_1, \ldots, A_n$. Following linear logic, the context Δ is restricted and rejects contraction and weakening. Programatically, this means that linear channels cannot be aliased nor freely deleted – they must be fully consumed exactly once.

Since the session type associated with a channel denotes a bidirectional protocol, each connective has two operational interpretations – one from the perspective of the provider and one from the client. This operationally dual interpretation results in a schema where for any connective, either the client or provider will send while the other will receive as summarized in Table 1.

For example, a channel of type $A \otimes 1$ requires that the provider sends a channel of type A and proceeds as type 1 while the client receives a channel of type A and proceeds as 1. The multiplicative unit 1 denotes the end of the protocol – the provider must terminate and close its channel while a client must wait for the channel to be closed. A channel of type $\oplus\{\overline{l : A}\}$ (n-nary internal choice) requires the provider to choose and send a label i in $\bar{l}$ and proceed as A_i while the client must receive and branch on some label i and proceed as A_i. Similarly, a channel of type $\&\{\overline{l : A}\}$ requires the client to choose and send a label and the provider to receive and branch on a label. The *continuation type* of some session type refers to the type after a message exchange; for example, B would be the continuation type of $A \otimes B$ and similarly A_i of $\oplus\{\overline{l : A}\}$ for some i in $\bar{l}$. The unit 1 does not have a continuation type since it marks the end of communication.

We consider a session type denoting the interaction with a provider of a queue of integers, which we will develop throughout the paper:

$$\begin{aligned}\textbf{queue} = \&\{&\mathit{enqueue} : \text{int} \supset \textbf{queue},\\ &\mathit{dequeue} : \oplus\{\mathit{some} : \text{int} \wedge \textbf{queue}, \mathit{none} : \textbf{queue}\}\}\end{aligned}$$

Type	Interpretation from provider	Interpretation from client	Continuation
1	Close channel (terminate)	Wait for channel to close	-
$A \otimes B$	Send channel of type A	Receive channel of type A	B
$A \multimap B$	Receive channel of type A	Send channel of type A	B
$\oplus\{\overline{l : A}\}$	Send a label $i \in \bar{l}$	Receive and branch on $i \in \bar{l}$	A_i
$\&\{\overline{l : A}\}$	Receive and branch on $i \in \bar{l}$	Send a label $i \in \bar{l}$	A_i

Table 1. A summary of the linear connectives and their operational interpretations

where we informally adopt value input and output $\supset$ and $\wedge$ [21] as value analogues to channel input and output $\multimap$ and $\otimes$, respectively, which are orthogonal to the advancements in this work. Following this protocol, a client must send a label *enqueue* or *dequeue*. If it chooses *enqueue*, it must send an int and then recur, and on the other hand, if it chooses *dequeue*, it will receive either some int as indicated by the *some* branch of the internal choice or nothing as indicated by the *none* branch. In either case, we let the queue recur[1]. Dually, a server must first receive a label *enqueue* or *dequeue* from the client. If it receives an *enqueue*, it will receive an int and then recur. If it receives a *dequeue* instead, it must either send a *some* label followed by the appropriate int and then recur or send a *none* label and then recur.

We adopt an *equi-recursive* [8] interpretation which requires that recursive session types be *contractive* [11], guaranteeing that there are no messages associated with the unfolding of a recursive type. This in particular requires that we reason about session types *coinductively*.

We now attempt to encode a protocol representing an auction based on [9]. An auction transitions between the bidding phase where clients are allowed to place bids and the collecting phase where a winner is given the item while all the losers are refunded their respective bids.

$$\begin{aligned}
\textbf{bidding} &= \&\{bid : \oplus\{ok : \text{id} \supset \text{money} \supset \textbf{bidding}, \\
&\qquad\qquad\quad collecting : \textbf{collecting}\}\} \\
\textbf{collecting} &= \&\{collect : \text{id} \supset \oplus\{prize : \text{item} \wedge \textbf{bidding}, \\
&\qquad\qquad\qquad\qquad refund : \text{money} \wedge \textbf{bidding}, \\
&\qquad\qquad\qquad\qquad bidding : \textbf{bidding}\}\}
\end{aligned}$$

In this example, we make the bidding phase and collecting phase explicit by separating the protocol into **bidding** and **collecting**. Beginning with **bidding**, a client must send a *bid* label [2]. The provider will either respond with an *ok*, allowing the client to make a bid by sending its id, money, and then recursing

[1] We do not consider termination to more easily align with later examples.

[2] The currently unnecessary unary choice will be useful later.

back to **bidding**, or a *collecting*, indicating that the auction is in the collecting phase and thereby making the client transition to **collecting**.

For **collecting**, the client must send a *collect* label. For ease of presentation, we require the client to also send its id immediately, giving enough information to the provider to know if the client should receive a *prize* or a *refund*, along with *bidding* if the client is in the wrong phase. The *prize* branch covers the case where the client won the previous bid, the *refund* branch covers the case where the client lost the bid, and the *bidding* branch informs the client that the auction is currently in the bidding phase.

Because linear channels have exactly one provider and one client, what we have described so far only encodes a single participant auction. One can assert that the provider is actually a broker to an auction of multiple participants, but that does not solve the fundamental problem, that is, encoding shared communication with multiple clients.

2.2 Shared Session Types

Although linear session types and their corresponding process calculi give a system with strong guarantees such as *session fidelity* (preservation) and *deadlock freedom* (progress), as we show in the previous section while attemping to encode an auction, they are not expressive enough to model systems with shared resources. Since multiple clients cannot simultaneously communicate to a single provider in an unrestricted manner, we adopt an *acquire-release* paradigm. The only action a client can perform on a shared channel is to send an acquire request, which the provider must accept. After successfully acquiring, the client is guaranteed to have exclusive access to the provider and therefore can communicate linearly until the client releases its exclusive access.

Instead of treating the acquire and release operations as mere operational primitives, in prior work [2] we extend the type system such that the acquire and release points are manifest in the type by stratifying session types into shared and linear types. Unlike linear channels, shared channels are unrestricted in that they can be freely aliased or deleted. In the remaining sections, we will make the distinction between linear and shared explicit by marking channel names and session type meta-variables with subscripts L and S respectively where appropriate. For example, a linear channel is marked a_L, while a shared channel is marked b_S.

Since shared channels represent unrestricted channels that must first be acquired, we introduce the modal upshift operator $\uparrow_L^S A_L$ for some A_L which requires clients to acquire and then proceed linearly as prescribed by A_L. Similarly, the modal downshift operator $\downarrow_L^S B_S$ for some B_S requires clients to release and proceed as a shared type. Type theoretically, these modal shifts mark transitions between shared to linear and vice versa. In summary, we have:

$$
\begin{array}{lrcl}
\text{(Shared Layer)} & A_S & ::= & \uparrow_L^S A_L \\
\text{(Linear Layer)} & A_L, B_L & ::= & \downarrow_L^S A_S \mid 1 \mid A_L \otimes B_L \mid A_L \multimap B_L \mid \&\{\overline{l{:}A_L}\} \mid \oplus\{\overline{l{:}A_L}\}
\end{array}
$$

where we emphasize that the previously defined (linear) type operators such as $\otimes$ remain only at the linear layer – a shared session type can only be constructed by a modal upshift $\uparrow_L^S$ of some linear session type A_L.

As initially introduced, clients of shared channels follow an *acquire-release* pattern – they must first acquire exclusive access to the channel, proceed linearly, and then finally release the exclusive access that they had, allowing other clients of the same shared channel to potentially acquire exclusive access. The middle linear section can also be viewed as a *critical region* since the client is guaranteed unique access to a shared provider process. Therefore, this system naturally supports atomic operations on shared resources.

Using shared channels, we can encode a shared queue, where there can be multiple clients interacting with the same data:

$$
\begin{aligned}
\mathbf{shared\_queue} = \uparrow_L^S \&\{ & \mathit{enqueue} : \mathrm{int} \supset \downarrow_L^S \mathbf{shared\_queue}, \\
& \mathit{dequeue} : \oplus\{ \mathit{some} : \mathrm{int} \wedge \downarrow_L^S \mathbf{shared\_queue}, \\
& \qquad\qquad\quad \mathit{none} : \downarrow_L^S \mathbf{shared\_queue}\}\}
\end{aligned}
$$

A client of such a channel must first send an **acquire** message, being blocked until the acquisition is successful. Upon acquisition, the client must then proceed linearly as in the previously defined linear queue. The only difference is that before recursing, the client must **release** its exclusive access, allowing other blocked clients to successfully acquire.

3 Equi-synchronizing Rules Out Phasing

We can also attempt to salvage the previous attempt of encoding (multi-participant) auctions by "wrapping" the previous purely linear protocol between $\uparrow_L^S$ and $\downarrow_L^S$.

$$
\begin{aligned}
\mathbf{bidding} &= \uparrow_L^S \&\{ \mathit{bid} : \oplus\{ \mathit{ok} : \mathrm{id} \supset \mathrm{money} \supset \downarrow_L^S \mathbf{bidding}, \\
&\qquad\qquad\qquad\quad\; \mathit{collecting} : \downarrow_L^S \mathbf{collecting}\}\} \\
\mathbf{collecting} &= \uparrow_L^S \&\{ \mathit{collect} : \mathrm{id} \supset \oplus\{ \mathit{prize} : \mathrm{item} \wedge \downarrow_L^S \mathbf{bidding}, \\
&\qquad\qquad\qquad\qquad\qquad\quad\; \mathit{refund} : \mathrm{money} \wedge \downarrow_L^S \mathbf{bidding}, \\
&\qquad\qquad\qquad\qquad\qquad\quad\; \mathit{bidding} : \downarrow_L^S \mathbf{bidding}\}\}
\end{aligned}
$$

A client to **bidding** must first acquire exclusive access as indicated by $\uparrow_L^S$, proceed linearly, and then eventually release at either **bidding** (in the *ok* branch) or **collecting** (in the *collecting* branch). Similarly, a client to **collecting** must first acquire exclusive access, proceed linearly, and then eventually release at **bidding** since all branches lead to **bidding**.

Unfortunately, as formulated so far, this protocol is not sound. For example, consider two auction participants P and Q that are both in the collecting phase and blocked trying to acquire. Suppose P successfully acquires, in which case it follows the protocol linearly and eventually releases at **bidding**. Then, if Q successfully acquires, we have a situation where Q rightfully believes that it acquired at **collecting** but since P previously released at type **bidding**, the auctioneer believes that it currently accepted a connection from **bidding**. The subsequent label sent by the client, *collect* is not an available option for the provider; session fidelity has been violated.

Previous work [2] addresses this problem by introducing an additional requirement that if a channel was acquired at some type A_S, all possible future releases (by looking at the continuation types) must release at A_S. This is formulated as the *equi-synchronizing* constraint, defined coinductively on the structure of session types. In particular, neither **bidding** nor **collecting** are equi-synchronizing because they do not always release at the same type at which it was acquired. For **bidding**, the *collecting* branch causes a release at a different type, and for **collecting**, all branches lead to a release at a different type.

A solution to the auction scenario is to unify the two phases into one:

$$\begin{aligned}
\mathbf{auction} = {\uparrow}_L^S \& \{ & \mathit{bid} : \oplus\{\mathit{ok} : \mathrm{id} \supset \mathrm{money} \supset {\downarrow}_L^S \mathbf{auction}, \\
& \qquad\qquad\quad \mathit{collecting} : {\downarrow}_L^S \mathbf{auction}\}, \\
& \mathit{collect} : \mathrm{id} \supset \oplus\{\mathit{prize} : \mathrm{item} \wedge {\downarrow}_L^S \mathbf{auction}, \\
& \qquad\qquad\quad \mathit{refund} : \mathrm{money} \wedge {\downarrow}_L^S \mathbf{auction}, \\
& \qquad\qquad\quad \mathit{bidding} : {\downarrow}_L^S \mathbf{auction}\}\}
\end{aligned}$$

The type **auction** is indeed equi-synchronizing because all possible release points are at **auction**.

This presentation of the auction however loses the explicit denotation of the two phases; although the previous linear single participant version of the auction protocol can make explicit the bidding and collecting phases in the session type, the equi-synchronizing requirement forces the two phases to merge into one in the case of shared session types. In general, the requirement that all release points are equivalent prevents shared session types to encode protocols across multiple acquire-release cycles since information is necessarily "lost" after a particular acquire-release cycle.

4 Subtyping

So far, there is an implicit requirement that given a particular channel, both its provider and clients agree on its protocol or type. A relaxation of this requirement in the context of linear session types have been investigated by Gay and Hole [11], and in this section, we present subtyping in the context of both linear session types and shared session types.

If $A \leq B$, then a provider viewing its offering channel as type A can safely communicate with a client viewing the same channel as type B. This perspective reveals a notion of *substitutability*, where a process providing a channel of type A can be replaced by a process providing A' such that $A' \leq A$ and dually, a client to some channel of type B can be replaced by another process using the same channel as some type B' such that $B \leq B'$. The following subtyping rules, interpreted coinductively, formalize the subtyping relation between session types:

$$\frac{}{1 \leq 1} \leq_1 \qquad \frac{A_L \leq A'_L \qquad B_L \leq B'_L}{A_L \otimes B_L \leq A'_L \otimes B'_L} \leq_\otimes \qquad \frac{A'_L \leq A_L \qquad B_L \leq B'_L}{A_L \multimap B_L \leq A'_L \multimap B'_L} \leq_\multimap$$

$$\frac{\forall i \in \bar{l} \quad {A_i}_L \leq {A'_i}_L}{\oplus\{\overline{l{:}A_L}\} \leq \oplus\{\overline{l{:}A'_L}, \overline{m{:}B_L}\}} \leq_\oplus \qquad \frac{\forall i \in \bar{l} \quad {A_i}_L \leq {A'_i}_L}{\&\{\overline{l{:}A_L}, \overline{m{:}B_L}\} \leq \&\{\overline{l{:}A'_L}\}} \leq_\&$$

One of the notable consequences of adopting subtyping is that internal and external choices allow one side to have more labels or branches. For internal choice, since the provider sends some label, there is no harm in a client to be prepared to handle additional labels that it will never receive and vice versa for external choice. Another observation is that subtyping of session types is covariant in their continuations; following this paradigm, we can immediately define subtyping for the new type connectives $\uparrow_L^S$ and $\downarrow_L^S$:

$$\frac{A_L \leq B_L}{\uparrow_L^S A_L \leq \uparrow_L^S B_L} \leq_{\uparrow_L^S} \qquad \frac{A_S \leq B_S}{\downarrow_L^S A_S \leq \downarrow_L^S B_S} \leq_{\downarrow_L^S}$$

Remark 1. The subtyping relation $\leq$ is a partial order.

A key principle governing subtyping of session types is that *ignorance is bliss*; neither the client nor the provider need to know the precise protocol that the other party is following, as supported by our extended report [20] which proves the same progress and preservation theorems in an implementation of session typed process calculus with shared channels [2] in a system with subtyping.

Let us revisit the shared queue example:

$$\begin{aligned} \mathbf{shared\_queue} = \uparrow_L^S \&\{ & \mathit{enqueue} : \mathrm{int} \supset \downarrow_L^S \mathbf{shared\_queue}, \\ & \mathit{dequeue} : \oplus\{\mathit{some} : \mathrm{int} \wedge \downarrow_L^S \mathbf{shared\_queue}, \\ & \qquad \mathit{none} : \downarrow_L^S \mathbf{shared\_queue}\}\} \end{aligned}$$

Instead of allowing all clients to freely enqueue and dequeue, suppose we only allow certain clients to enqueue and certain clients to dequeue. With subtyping, we first fix the provider's type to be **shared_queue**. Next, we restrict writer clients by removing the *dequeue* label and similarly restrict reader clients by removing the *enqueue* label:

$$\begin{aligned}\mathbf{producer} &= \uparrow_L^S \&\{\mathit{enqueue} : \text{int} \supset \downarrow_L^S \mathbf{producer}\} \\ \mathbf{consumer} &= \uparrow_L^S \&\{\mathit{dequeue} : \oplus\{\mathit{some} : \text{int} \wedge \downarrow_L^S \mathbf{consumer}, \mathit{none} : \downarrow_L^S \mathbf{consumer}\}\}\end{aligned}$$

where it is indeed the case that **shared_queue** $\leq$ **producer** and **shared_queue** $\leq$ **consumer**, justifying both the writer and reader clients' views on the type of the channel.

We will defer the detailed discussion of the subtle interactions that occur between the notion of equi-synchronizing constraint and subtyping to Section 5.1. For this example however, the fact that all three types **shared_queue**, **producer**, and **consumer** are independently equi-synchronizing is a strong justification of its soundness.

5 Phasing

One of the most common patterns when encoding data structures and protocols via session types is to begin the linear type with an external choice. When these types recur, we are met with another external choice. A notion of *phasing* emerges from this pattern, where a single phase spans from the initial external choice to the recursion.

We introduced an auction protocol, which in its linear form can make explicit the two distinct phases, yet in its shared form cannot due to the equi-synchronizing constraint. With subtyping however, this seems to no longer be a problem; the auctioneer can view the protocol as auction whereas the clients can independently view the protocol as **bidding** or **collecting** depending on their current phase since **auction** $\leq$ **bidding** and **auction** $\leq$ **collecting**.

$$\text{provider}\begin{cases}\mathbf{auction} = \uparrow_L^S \&\{\mathit{bid} : \oplus\{\mathit{ok} : \text{id} \supset \text{money} \supset \downarrow_L^S \mathbf{auction}, \\ \qquad\qquad\qquad\qquad \mathit{collecting} : \downarrow_L^S \mathbf{auction}\}, \\ \qquad\qquad \mathit{collect} : \text{id} \supset \oplus\{\mathit{prize} : \text{item} \wedge \downarrow_L^S \mathbf{auction}, \\ \qquad\qquad\qquad\qquad \mathit{refund} : \text{money} \wedge \downarrow_L^S \mathbf{auction}, \\ \qquad\qquad\qquad\qquad \mathit{bidding} : \downarrow_L^S \mathbf{auction}\}\}\end{cases}$$

$$\text{clients}\begin{cases}\qquad \mathbf{bidding} = \uparrow_L^S \&\{\mathit{bid} : \oplus\{\mathit{ok} : \text{id} \supset \text{money} \supset \downarrow_L^S \mathbf{bidding}, \\ \qquad\qquad\qquad\qquad\qquad \mathit{collecting} : \downarrow_L^S \mathbf{collecting}\}\} \\ \mathbf{collecting} = \uparrow_L^S \&\{\mathit{collect} : \text{id} \supset \oplus\{\mathit{prize} : \text{item} \wedge \downarrow_L^S \mathbf{bidding}, \\ \qquad\qquad\qquad\qquad\qquad \mathit{refund} : \text{money} \wedge \downarrow_L^S \mathbf{bidding}, \\ \qquad\qquad\qquad\qquad\qquad \mathit{bidding} : \downarrow_L^S \mathbf{bidding}\}\}\end{cases}$$

Unfortunately, there is a critical issue with this solution. Since shared channels can be aliased, a client in the collecting phase can alias the channel, follow the protocol, and then ignore the released type (bidding phase) – it can then use the previously aliased channel to communicate as if in the collecting phase. In general, the strategy of encoding phases in shared communication through a shared supertype allows malicious clients to re-enter previously encountered phases since they may internally store aliases. Thus, what we require is a subtyping relation across shared and linear modes since linear channels are restricted and in particular cannot be aliased.

We first add two new linear connectives $\uparrow_L^L$ and $\downarrow_L^L$ that, like $\uparrow_L^S$ and $\downarrow_L^S$, have operationally an acquire-release semantics but enforce a linear treatment of the associated channels. Prior work [14] has already explored such intra-layer shifts, albeit for the purpose of enforcing synchronization in an asynchronous message-passing system. Thus for example, the protocol denoted by $\uparrow_L^L A_L$ requires the client to "acquire" as in the shared case. If the provider happens to provide a linear channel $\uparrow_L^L A_L$, then this merely adds a synchronization point in the communication. The more interesting case is when the provider is actually providing a shared channel, some $\uparrow_L^S A_L$; a client should be able to view the session type as $\uparrow_L^L A_L$ without any trouble. We formalize this idea to the following additional subtyping relations:

$$\frac{A_L \leq B_L}{\uparrow_L^S A_L \leq \uparrow_L^L B_L} \leq_{\uparrow_L^S \uparrow_L^L} \quad \frac{A_S \leq B_L}{\downarrow_L^S A_S \leq \downarrow_L^L B_L} \leq_{\downarrow_L^S \downarrow_L^L} \quad \frac{A_L \leq B_L}{\uparrow_L^L A_L \leq \uparrow_L^L B_L} \leq_{\uparrow_L^L} \quad \frac{A_L \leq B_L}{\downarrow_L^L A_L \leq \downarrow_L^L B_L} \leq_{\downarrow_L^L}$$

Using the new connectives, we can complete the auction protocol where the two phases are manifest in the session type; a client must actually view the auction protocol linearly!

$$\begin{aligned}
\mathbf{bidding} &= \uparrow_L^L \& \{ \mathit{bid} : \oplus \{ \mathit{ok} : \mathrm{id} \supset \mathrm{money} \supset \downarrow_L^L \mathbf{bidding}, \\
&\qquad\qquad\qquad\qquad \mathit{collecting} : \downarrow_L^L \mathbf{collecting} \} \} \\
\mathbf{collecting} &= \uparrow_L^L \& \{ \mathit{collect} : \mathrm{id} \supset \oplus \{ \mathit{prize} : \mathrm{item} \wedge \downarrow_L^L \mathbf{bidding}, \\
&\qquad\qquad\qquad\qquad \mathit{refund} : \mathrm{money} \wedge \downarrow_L^L \mathbf{bidding}, \\
&\qquad\qquad\qquad\qquad \mathit{bidding} : \downarrow_L^L \mathbf{bidding} \} \}
\end{aligned}$$

where **auction** $\leq$ **bidding** and **auction** $\leq$ **collecting**. Compared to the initially presented linear auction protocol, this version inserts the purely linear shifts $\uparrow_L^L$ and $\downarrow_L^L$ where appropriate such that the protocol is compatible with the shared auction protocol that the auctioneer provides. Therefore, the addition of $\uparrow_L^L$ and $\downarrow_L^L$ to our system allows a natural subtyping relation between shared session types and linear session types, where they serve as a means to safely bridge between shared and linear modalities.

Remark 2. A protocol spanning multiple phases can also be interpreted as a deterministic finite autonomata (DFA) where nodes represent the phase or the state of the protocol and edges represent choice branches. The previous auction protocol can be encoded as a two state DFA as shown in Figure 1.

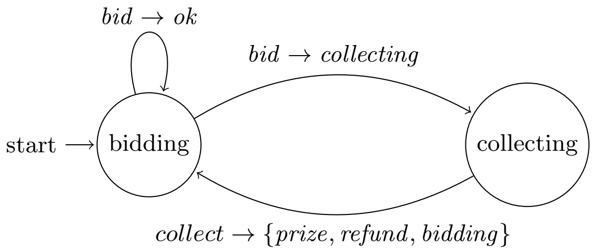

Fig. 1. A DFA representation of the two phases in the auction protocol. Multiple labels enclosed in brackets as in $\{\mathit{prize}, \mathit{refund}, \mathit{bidding}\}$ mean that any of those labels can be selected.

5.1 Subsynchronizing Constraint

We note in Section 2.2 that in previous work [2], we require session types to be equi-synchronizing, which requires that processes following the protocol are released at the exact type at which it was acquired. This constraint guarantees that clients do not acquire at a type that they do not expect. With the introduction of subtyping however, there are two major relaxations that we propose on this constraint.

Releasing at a subtype A client P using some channel as some type $a_S{:}A_S$ can safely communicate with any (shared) process offering a channel of type $a_S{:}A'_S$ such that $A'_S \leq A_S$ due to subtyping. If another client acquires a_S and releases it at some A''_S such that $A''_S \leq A'_S$, then P can still safely communicate along a_S since $A''_S \leq A_S$ by transitivity. Thus, one reasonable relaxation to the equi-synchronizing constraint is that processes do not need to be released at the same exact type but instead a subtype.

Branches that never occur A major consequence of subtyping is that providers and clients can wait on some branches in the internal and external choices which in fact never will be sent by the other party. For example, suppose a provider P provides a channel of type $A_S = \uparrow_L^S \& \{a : \downarrow_L^S A_S, b : \downarrow_L^S B_S\}$. Assuming some unrelated B_S, we can see that A_S is not equi-synchronizing because the b branch can lead to releasing at a different type. However, suppose some client C views the channel as $\uparrow_L^S \& \{a : \downarrow_L^S A_S\}$ – in this case, P can only receive a, and the b branch can safely be ignored since C will never send the b label. This points to the necessity of using both the provider and client types to more finely verify the synchronizing constraint. Of course, if there is another client D that views the channel in a way that the b branch can be taken, then the entire setup is not synchronizing. Thus, we must verify the synchronization constraint for all pairs of providers and clients.

Following previous work [2], we formulate constraints by extending the shared types: $\hat{A} ::= \bot \mid A_S \mid \top$ where $\bot \leq A_S \leq \top$ for any A_S. Intuitively, $\top$ indicates a channel that has not been acquired yet (no constraints on a future release), A_S indicates the previous presentation of shared channels, and $\bot$ indicates a channel that will never be available (hence, any client attempting to acquire from this channel will never succeed and be blocked).

We are now ready to present the *subsynchronizing* judgment, interpreted coinductively, which is of the form $\vdash (A, B, \hat{D})$ ssync for some A and B such that $A \leq B$. It asserts that a provider providing a channel of type A and a client using that channel with type B is subsynchronizing with respect to some constraint $\hat{D}$. To verify a pair of types A and B to be subsynchronizing, we take $\top$ as its initial constraint (recall that $\top$ represents no constraint), that is, we say that A and B are subsynchronizing if $\vdash (A, B, \top)$ ssync.

$$\frac{}{\vdash (1, 1, \hat{D})\ \text{ssync}}\ S1$$

$$\frac{\vdash (B_L, B'_L, \hat{D})\ \text{ssync}}{\vdash (A_L \otimes B_L, A'_L \otimes B'_L, \hat{D})\ \text{ssync}}\ S\otimes \qquad \frac{\vdash (B_L, B'_L, \hat{D})\ \text{ssync}}{\vdash (A_L \multimap B_L, A'_L \multimap B'_L, \hat{D})\ \text{ssync}}\ S\multimap$$

$$\frac{\forall i \in \bar{l} \quad \vdash (A_{iL}, A'_{iL}, \hat{D})\ \text{ssync}}{\vdash (\oplus\{\overline{l{:}A_L}\}, \oplus\{\overline{l{:}A'_L}, \overline{m{:}B_L}\}, \hat{D})\ \text{ssync}}\ S\oplus \qquad \frac{\forall i \in \bar{l} \quad \vdash (A_{iL}, A'_{iL}, \hat{D})\ \text{ssync}}{\vdash (\&\{\overline{l{:}A_L}, \overline{m{:}B_L}\}, \&\{\overline{l{:}A'_L}\}, \hat{D})\ \text{ssync}}\ S\&$$

$$\frac{\vdash (A_L, A'_L, \hat{D})\ \text{ssync}}{\vdash (\uparrow_L^L A_L, \uparrow_L^L A'_L, \hat{D})\ \text{ssync}}\ S\uparrow_L^L \qquad \frac{\vdash (A_L, A'_L, \hat{D})\ \text{ssync}}{\vdash (\downarrow_L^L A_L, \downarrow_L^L A'_L, \hat{D})\ \text{ssync}}\ S\downarrow_L^L$$

$$\frac{\vdash (A_L, A'_L, \uparrow_L^S A_L)\ \text{ssync}}{\vdash (\uparrow_L^S A_L, \uparrow_L^S A'_L, \top)\ \text{ssync}}\ S\uparrow_L^S \qquad \frac{\vdash (A_S, A'_S, \top)\ \text{ssync} \quad \downarrow_L^S A_S \leq \hat{D}}{\vdash (\downarrow_L^S A_S, \downarrow_L^S A'_S, \hat{D})\ \text{ssync}}\ S\downarrow_L^S$$

$$\frac{\vdash (A_L, A'_L, \uparrow_L^S A_L)\ \text{ssync}}{\vdash (\uparrow_L^S A_L, \uparrow_L^L A'_L, \top)\ \text{ssync}}\ S\uparrow_L^S\uparrow_L^L \qquad \frac{\vdash (A_S, A'_L, \top)\ \text{ssync} \quad \downarrow_L^S A_S \leq \hat{D}}{\vdash (\downarrow_L^S A_S, \downarrow_L^L A'_L, \hat{D})\ \text{ssync}}\ S\downarrow_L^S\downarrow_L^L$$

The general progression of derivations to verify that two types are subsynchronizing is to first look for an upshift $\uparrow_L^S$ on the provider's type, involving either $S\uparrow_L^S$ or $S\uparrow_L^S\uparrow_L^L$. After encountering a $\uparrow_L^S$, it "records" the provider's type as the constraint and continues to look at the continuations of the types. When encountering internal and external choices, it only requires the continuations for the common branches to be subsynchronizing. When it encounters a downshift $\downarrow_L^S$ from the provider's side, it checks if the release point as denoted by the continuation of $\downarrow_L^S$ is a subtype of the recorded constraint, in which case it continues with the derivation with the $\top$ constraint.

Remark 3. Subsynchronizing is a strictly weaker constraint than equi-synchronizing. In particular, if A is equi-synchronizing, then the pair A, A are subsynchronizing.

6 Metatheory

In this section we present the progress and preservation theorems in a synchronous message passing concurrent system implementing our type system. We defer many of the technical details of the system and the proofs to our extended report [20] which follows a similar style to the system in a previous work [2]. In particular, the two theorems are equally strong as the ones in [2], justifying our subtyping extension.

6.1 Process Typing

We take the typing judgment presented in Section 2.1 and extend it with shared channels as introduced in Section 2.2:

$$\Gamma \vdash P :: (a_S{:}A_S)$$
$$\Gamma; \Delta \vdash Q :: (a_L{:}A_L)$$

where $\Gamma = a_{1S}{:}\hat{A}_1, \ldots, a_{nS}{:}\hat{A}_n$ is a structural context of shared channels and constraints ($\bot$ and $\top$) which can appear at runtime.

The first judgment asserts that P provides a shared channel $a_S{:}A_S$ while using shared channels in Γ; the lack of dependence on any linear channels Δ is due to the *independence principle* presented in [2]. The second judgment asserts that Q provides a linear channel $a_L{:}A_L$ while using shared channels in Γ and linear channels in Δ.

Forwarding is a fundamental operation that allows a process to identify its offering channel with a channel it uses if the types match.

$$\frac{B_L \leq A_L}{\Gamma; y_L{:}B_L \vdash \mathsf{fwd}\ x_L\ y_L :: (x_L{:}A_L)}\ ID_L \qquad \frac{\hat{B} \leq A_S}{\Gamma, y_S{:}\hat{B} \vdash \mathsf{fwd}\ x_S\ y_S :: (x_S{:}A_S)}\ ID_S$$

$$\frac{\hat{B} \leq A_L}{\Gamma, y_S{:}\hat{B}; \cdot \vdash \mathsf{fwd}\ x_L\ y_S :: (x_L{:}A_L)}\ ID_{LS}$$

The rules ID_L and ID_S require the offering channel to be a supertype of the channel it is being identified with. Since we syntactically distinguish shared channels and linear channels, we require an additional rule ID_{LS} that allows linear channels to be forwarded with a shared channel provided the subtyping relation holds.

We also show the right rule of $\otimes$, which requires the provider to send a channel y_L alongside its offering channel x_L:

$$\frac{A'_L \leq A_L \quad \Gamma; \Delta \vdash P :: (x_L{:}B_L)}{\Gamma; \Delta, y_L{:}A'_L \vdash \mathsf{send}\ x_L\ y_L; P :: (x_L{:}A_L \otimes B_L)}\ \otimes R$$

Similar to the forwarding case, a shared channel can instead be sent if the appropriate subtyping relation holds:

$$\frac{\hat{A} \leq A_L \quad \Gamma, y_S{:}\hat{A}; \Delta \vdash P :: (x_L{:}B_L)}{\Gamma, y_S{:}\hat{A}; \Delta \vdash \mathsf{send}\ x_L\ y_S; P :: (x_L{:}A_L \otimes B_L)} \otimes R_S$$

One important observation is that typing judgments remain local in the presence of subtyping; the channels in Γ and Δ may be provided by processes at some subtype (maintained in the configuration; see Section 6.3) and need not match. We therefore do not adopt a general subsumption rule that allows arbitrary substitutions that preserve subtyping and instead precisely manage where subtyping occurs in the system.

6.2 Processes and Configuration

To reason about session types and process calculi, we must consider a collection of message passing processes, which is known as a *configuration*. In our system, we split the configuration into the shared fragment Λ and the linear fragment Θ, where Λ is a list of *process predicates* that offer shared channels and Θ is similarly a list of process predicates that offer linear channels.

The most fundamental process predicate denotes a process term P that provides some channel a and is of form $\mathrm{proc}(a, P)$. We also introduce the predicate $\mathrm{unavail}(a_S)$, which represents a shared process that is unavailable to be acquired, for example, due to it being acquired by another process, and $\mathrm{connect}(a_L, b_S)$, which provides a linear reference a_L to a shared channel b_S, needed to express shared to linear subtyping.

We require the linear configuration Θ to obey an ordering that processes can only depend on processes that appear to its right; $\mathrm{proc}(a_L, P), \mathrm{proc}(b_L, Q)$ would be ill-formed if P depends on b_L. On the other hand, Λ has no ordering constraints. For the subsequent sections, we require that configurations are *well-formed*, which essentially requires that both shared and linear processes provide unique channel names thereby avoiding naming conflicts.

6.3 Configuration Typing

The configuration typing judgment asserts that a given configuration collectively provides a set of shared and linear channels; each fragment is checked separately as shown by the only typing rule for the combined configuration:

$$\frac{\Gamma \models \Lambda :: (\Gamma) \quad \Gamma \models \Theta :: (\Delta)}{\Gamma \models \Lambda; \Theta :: (\Gamma; \Delta)} \ \Omega$$

The shared context Γ appears on both sides due to circularity; the appearance on the left side allows any processes to depend on a particular shared channel in Γ while the appearance on the right side asserts that Λ collectively provides Γ. In most cases, $\hat{A}$ is some shared session type A_S, but the maximal and minimal types $\perp$ and $\top$ can appear at runtime.

For the shared fragment, we check each process predicate independently; in particular, a configuration typing rule for some $\mathrm{proc}(a_S, P)$ is shown below.

$$\frac{\vdash (A'_S, A_S, \top)\ \mathrm{ssync} \quad \Gamma \vdash P :: (a_S{:}A'_S)}{\Gamma \models \mathrm{proc}(a_S, P) :: (a_S{:}A_S)}\ \Lambda 3$$

An important point is that a_S is of type A_S in Γ and A'_S is only local to the process typing judgment; thus, the provider P views the channel a_S at type A'_S while all clients view the channel a_S at type A_S. The subtyping relation $A'_S \leq A_S$ is subsumed by the subsynchronizing judgment $\vdash (A'_S, A_S, \top)$ ssync which guarantees that the pair (A'_S, A_S) is subsynchronizing.

A configuration typing rule for some (linear) $\mathrm{proc}(a_L, P), \Theta'$ is shown below.

$$\frac{a_S{:}\hat{A} \in \Gamma \quad \vdash (A'_L, A_L, \hat{A})\ \mathrm{ssync} \quad \Gamma; \Delta_a \vdash P :: (a_L{:}A'_L) \quad \Gamma \models \Theta' :: (\Delta_a, \Delta')}{\Gamma \models \mathrm{proc}(a_L, P), \Theta' :: (a : A_L, \Delta')}\ \Theta 3$$

Since P may use some linear channels Δ_a, we split the offering channels of Θ' to Δ_a, Δ' and make explicit that P will consume Δ_a. Similar to the shared case, the process typing judgment (third premise) locally assumes a_L is of type A'_L such that $A'_L \leq A_L$ (again, subsumed by the subsynchronizing judgment), guaranteeing that a client of a_L view the channel as type A_L.

6.4 Dynamics

The operational semantics of the system is formulated through *multiset rewriting rules* [5], which is of form $S_1, \ldots, S_n \to T_1, \ldots, T_m$, where each S_i and T_j corresponds to a process predicate. Each rule captures a transition in a subset of the configuration; for example, the following is one of three rules that capture the semantics of forwarding:

$$\mathrm{proc}(a_L, \mathrm{fwd}\ a_L\ b_S) \to \mathrm{connect}(a_L, b_S) \qquad \text{(D-FWDLS)}$$

Connect predicates are consumed if a process acquires linearly on a channel that is provided by a shared process. We first show how a shared process providing some b_S can be acquired:

$$\begin{matrix}\mathrm{proc}(a_L, x_L \leftarrow \mathrm{acq}_S\ b_S; P) \\ \mathrm{proc}(b_S, x_L \leftarrow \mathrm{acc}_S\ b_S; Q)\end{matrix} \to \begin{matrix}\mathrm{proc}(a_L, [b_L/x_L]P), \\ \mathrm{proc}(b_L, [b_L/x_L]Q)\end{matrix}, \mathrm{unavail}(b_S) \qquad (\text{D-}\uparrow_L^S)$$

The rule says that a client can successfully acquire if there is a corresponding accept by the provider. In the following rule, a connect predicate "coordinates" the acquire/accept between different modes:

$$\begin{matrix}\mathrm{proc}(a_L, x_L \leftarrow \mathrm{acq}_L\ b_L; P) \\ \mathrm{proc}(c_S, x_L \leftarrow \mathrm{acc}_S\ c_S; Q)\end{matrix}, \mathrm{connect}(b_L, c_S) \to \begin{matrix}\mathrm{proc}(a_L, [c_L/x_L]P), \\ \mathrm{proc}(c_L, [c_L/x_L]Q)\end{matrix}, \mathrm{unavail}(c_S)$$

$$(\text{D-}\uparrow_L^S 2)$$

6.5 Theorems

So far, we have incompletely introduced the statics [20, Appendix D] and the dynamics [20, Appendix E] of the system, focusing on the interesting cases that depart from the system presented in [2].

The preservation theorem, or session fidelity, guarantees that well-typed configurations remain well-typed. In particular, this means that processes will always adhere to the protocol denoted by the session type.

Theorem 1 (Preservation). *If $\Gamma \models \Lambda; \Theta :: (\Gamma; \Delta)$ for some Λ, Θ, Γ, and Δ, and $\Lambda; \Theta \to \Lambda'; \Theta'$ for some $\Lambda'; \Theta'$, then $\Gamma' \models \Lambda'; \Theta' :: (\Gamma'; \Delta)$ where $\Gamma' \preceq \Gamma$.*

Proof. By induction on the dynamics and constructing a well-typed configuration for each case. See [20, Appendix F] for the detailed proof, covering all cases.

The $\Gamma' \preceq \Gamma$ captures the idea that the configuration can gain additional shared processes and that the types of shared channels can become smaller. For example, if a process spawns an additional shared process, then the configuration will gain an additional channel in Γ and if a shared channel is released to a smaller type, the type of the shared channel in Γ can become smaller. Note that although it is indeed true that linear processes can be spawned, it will never appear in Δ since the linear channel that the newly spawned process offers must be consumed by the process that spawned the channel, meaning Δ is unchanged.

The progress theorem is as in [2], where we only allow configurations to be stuck due to failure of some client to acquire, for example, due to deadlock. A *poised* process [17,2] is one that is currently trying to communicate across its offering channel and is analogous to the role of *values* in typical functional languages. Both shared and linear configurations are poised if and only if all its processes are trying to communicate across its offering channel.

Theorem 2 (Progress). *If $\Gamma \models \Lambda; \Theta :: (\Gamma; \Delta)$ then either:*

1. *$\Lambda; \Theta \to \Lambda'; \Theta$ for some Λ' or*
2. *Λ is poised and one of:*
   (a) *$\Lambda; \Theta \to \Lambda'; \Theta'$ or*
   (b) *Θ is poised or*
   (c) *a linear process in Θ is unable stuck and therefore unable to acquire*

Proof. By induction on the typing of the configuration $\Lambda; \Theta$. We begin by induction on the typing of Λ to prove either (1) or Λ is poised. After, we prove (2) by induction on the typing of Θ while assuming Λ is poised. See [20, Appendix G] for the detailed proof.

Remark 4. Another paper [3] introduces additional static restrictions to allow a stronger and more common notion of progress, which are orthogonal to our results. Adopting this extension to the system we present here would give the usual notion of progress with deadlock freedom.

7 Related Work

Our paper serves as an extension to the manifest sharing system defined in [2] by introducing a notion of subtyping to the system which allows us to statically relax the equi-synchronizing constraint. Early glimpses of subtyping can be seen in the previous system with the introduction of $\bot$ and $\top$ as the minimal and maximal constraints, which happened to be compatible with our subtyping relation.

Subtyping for session types was first proposed by Gay and Hole [11], and a slightly modified style of session types guided from the correspondence with intuitionistic linear logic was given a subtyping extension [1]. Both these papers do not investigate the more recently discovered modal shifts, which is our contribution to the subtyping front.

There have also been many recent developments in subtyping in the context of multiparty session types [7,6,12,13], which are a different class of type systems that describe protocols between an arbitrary number of participants from a neutral global point of view. Understanding the relation of our subtyping system to these systems is an interesting item for future work.

8 Conclusion

We propose a subtyping extension to a message passing concurrency programming language introduced in previous work [2] and showed examples highlighting the expressiveness that this new system provides. Throughout the paper, we follow two important principles, *substitutability* and *ignorance is bliss*, which gave a rich type system that in particular allows *phases* (in a shared setting) to be manifest in the type.

One immediate application of shared subtyping is that combined with refinement types [10,9], it can encode finer specifications of protocols. For example in the auction scenario, we can statically show that each client that does not win a bid gets refunded precisely the exact amount of money it bid. Without shared to linear subtyping, specifications of shared communication across multiple acquire-release cycles were not possible.

A future work in a more theoretical platform is to extend the setting to adjoint logic [18], which provides a more general framework of reasoning about modal shifts in a message passing system. In particular, we found that affine session types, where contraction (aliasing) is rejected, have immediate applications.

Acknowledgements We would like to thank the anonymous reviewers for feedback on the initially submitted version of this paper.

References

1. Acay, C., Pfenning, F.: Intersections and unions of session types. In: Kobayashi, N. (ed.) 8th Workshop on Intersection Types and Related Systems (ITRS'16). pp. 4–19. EPTCS 242, Porto, Portugal (Jun 2016)
2. Balzer, S., Pfenning, F.: Manifest sharing with session types. In: International Conference on Functional Programming (ICFP). pp. 37:1–37:29. ACM (Sep 2017), extended version available as Technical Report CMU-CS-17-106R, June 2017.
3. Balzer, S., Toninho, B., Pfenning, F.: Manifest deadlock-freedom for shared session types. In: Caires, L. (ed.) 28th European Symposium on Programming (ESOP 2019). pp. 611–639. Springer LNCS 11423, Prague, Czech Republic (Apr 2019)
4. Caires, L., Pfenning, F.: Session types as intuitionistic linear propositions. In: Proceedings of the 21st International Conference on Concurrency Theory (CONCUR 2010). pp. 222–236. Springer LNCS 6269, Paris, France (Aug 2010)
5. Cervesato, I., Scedrov, A.: Relating state-based and process-based concurrency through linear logic. Information and Computation **207**(10), 1044–1077 (Oct 2009)
6. Chen, T.c., Dezani-Ciancaglini, M., Scalas, A., Yoshida, N.: On the Preciseness of Subtyping in Session Types. Logical Methods in Computer Science **Volume 13, Issue 2** (Jun 2017). https://doi.org/10.23638/LMCS-13(2:12)2017
7. Chen, T.C., Dezani-Ciancaglini, M., Yoshida, N.: On the preciseness of subtyping in session types. In: Proceedings of the Conference on Principles and Practice of Declarative Programming (PPDP'14). ACM, Canterbury, UK (Sep 2014)
8. Crary, K., Harper, R., Puri, S.: What is a recursive module? In: In SIGPLAN Conference on Programming Language Design and Implementation. pp. 50–63. ACM Press (1999)
9. Das, A., Balzer, S., Hoffmann, J., Pfenning, F., Santurkar, I.: Resource-aware session types for digital contracts. In: Küsters, R., Naumann, D. (eds.) 34th Computer Security Foundations Symposium (CSF 2021). IEEE, Dubrovnik, Croatia (Jun 2021), to appear
10. Das, A., Pfenning, F.: Session types with arithmetic refinements. In: Konnov, I., Kovács, L. (eds.) 31st International Conference on Concurrency Theory (CONCUR 2020). pp. 13:1–13:18. LIPIcs 171, Vienna, Austria (Sep 2020)
11. Gay, S.J., Hole, M.: Subtyping for session types in the π-calculus. Acta Informatica **42**(2–3), 191–225 (2005)
12. Ghilezan, S., Jakšić, S., Pantović, J., Scalas, A., Yoshida, N.: Precise subtyping for synchronous multiparty sessions. Journal of Logical and Algebraic Methods in Programming **104**, 127 – 173 (2019). https://doi.org/https://doi.org/10.1016/j.jlamp.2018.12.002
13. Ghilezan, S., Pantović, J., Prokić, I., Scalas, A., Yoshida, N.: Precise subtyping for asynchronous multiparty sessions (2020)
14. Griffith, D.: Polarized Substructural Session Types. Ph.D. thesis, University of Illinois at Urbana-Champaign (2015), in preparation
15. Honda, K.: Types for dyadic interaction. In: Best, E. (ed.) 4th International Conference on Concurrency Theory (CONCUR 1993). pp. 509–523. Springer LNCS 715 (1993)
16. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) 7th European Symposium on Programming Languages and Systems (ESOP 1998). pp. 122–138. Springer LNCS 1381 (1998)

17. Pfenning, F., Griffith, D.: Polarized substructural session types. In: Pitts, A. (ed.) Proceedings of the 18th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS 2015). pp. 3–22. Springer LNCS 9034, London, England (Apr 2015), invited talk
18. Pruiksma, K., Pfenning, F.: A message-passing interpretation of adjoint logic. In: Martins, F., Orchard, D. (eds.) Workshop on Programming Language Approaches to Concurrency and Communication-Centric Software (PLACES). pp. 60–79. EPTCS 291, Prague, Czech Republic (Apr 2019)
19. Sano, C.: On Session Typed Contracts for Imperative Languages. Masters thesis, Carnegie Mellon University (Dec 2019), available as Technical Report CMU-CS-19-133, December 2019.
20. Sano, C., Balzer, S., Pfenning, F.: Manifestly phased communication via shared session types. CoRR **abs/2101.06249** (2021), https://arxiv.org/abs/2101.06249
21. Toninho, B.: A Logical Foundation for Session-based Concurrent Computation. Ph.D. thesis, Carnegie Mellon University and Universidade Nova de Lisboa (May 2015), available as Technical Report CMU-CS-15-109
22. Wadler, P.: Propositions as sessions. In: Proceedings of the 17th International Conference on Functional Programming (ICFP 2012). pp. 273–286. ACM Press, Copenhagen, Denmark (Sep 2012)