Multiparty GV: Functional Multiparty Session Types with Certified Deadlock Freedom

JULES JACOBS, Radboud University Nijmegen, The Netherlands
STEPHANIE BALZER, Carnegie Mellon University, USA
ROBBERT KREBBERS, Radboud University Nijmegen, The Netherlands

Session types have recently been integrated with functional languages, bringing message-passing concurrency to functional programming. Channel endpoints then become first-class and can be stored in data structures, captured in closures, and sent along channels. Representatives of the GV (Wadler's "Good Variation") session type family are of particular appeal because they not only assert session fidelity but also deadlock freedom, inspired by a Curry-Howard correspondence to linear logic. A restriction of current versions of GV, however, is the focus on binary sessions, limiting concurrent interactions within a session to two participants. This paper introduces Multiparty GV (MPGV), a functional language with multiparty session types, allowing concurrent interactions among several participants. MPGV upholds the strong guarantees of its ancestor GV, including deadlock freedom, despite session interleaving and delegation. MPGV has a novel redirecting construct for modular programming with first-class endpoints, thanks to which we give a type-preserving translation from binary session types to MPGV to show that MPGV is strictly more general than binary GV. All results in this paper have been mechanized using the Coq proof assistant.

CCS Concepts: • **Software and its engineering → Concurrent programming languages**.

Additional Key Words and Phrases: Session types, message-passing concurrency, deadlock freedom

ACM Reference Format:
Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. 2022. Multiparty GV: Functional Multiparty Session Types with Certified Deadlock Freedom. *Proc. ACM Program. Lang.* 6, ICFP, Article 107 (August 2022), 30 pages.
https://doi.org/10.1145/3547638

1 INTRODUCTION

Session types are a type discipline for message-passing concurrency. Originally developed in the context of process calculi by Honda [1993]; Honda et al. [1998], they were later generalized to object-oriented [Dezani-Ciancaglini et al. 2006] and functional languages [Gay and Vasconcelos 2010] leading to implementations in mainstream languages like Haskell [Pucella and Tov 2008; Imai et al. 2010; Lindley and Morris 2016a], Scala [Scalas and Yoshida 2016], OCaml [Padovani 2017; Imai et al. 2019], and Rust [Jespersen et al. 2015; Kokke 2019; Chen et al. 2022].

A particularly exciting development is the GV ("Good Variation") session type family, pioneered by Gay and Vasconcelos [2010], later coined GV and refined by Wadler [2012], and further developed by *e.g.*, Lindley and Morris [2015, 2016b, 2017]; Fowler et al. [2019, 2021]; Kokke and Dardha [2021b]; Jacobs et al. [2022a]. The GV family combines session types with functional programming by treating session-typed channels as first-class data, similar to references in ML. Channels can be stored in data structures (like lists), captured by closures, and sent along channels (even when contained in

Authors' addresses: Jules Jacobs, Radboud University Nijmegen, The Netherlands, mail@julesjacobs.com; Stephanie Balzer, Carnegie Mellon University, USA, balzers@cs.cmu.edu; Robbert Krebbers, Radboud University Nijmegen, The Netherlands, mail@robbertkrebbers.nl.

2475-1421/2022/8-ART107
https://doi.org/10.1145/3547638

data structures, thus generalizing delegation). Similarly, the SILL family of session type languages [Toninho et al. 2013; Pfenning and Griffith 2015; Toninho 2015] integrates a process language via a contextual monad with an unrestricted functional language.

Aside from a tight integration with functional programming, a key strength of GV and SILL representatives is that they do not only guarantee type safety ("well-typed programs cannot get stuck due to illegal operations"), but also deadlock freedom or global progress ("well-typed programs cannot end up waiting for each other"). This result follows from adopting a session initialization pattern based on cut, inspired by the Curry-Howard correspondence between linear logic and the session-typed π-calculus [Caires and Pfenning 2010; Wadler 2012]. Such a pattern *combines* session creation and thread spawning to avoid deadlocks. The family of session types based on the pioneering work by Honda [1993]; Honda et al. [1998], in contrast, *separates* session creation from thread (process) spawning and thus does not prevent deadlocks. A cut-based initialization pattern also seamlessly integrates with channels as first-class data.

The restriction of interactions to *two* participants, present in GV, SILL, and session types based on the pioneering work by Honda [1993]; Honda et al. [1998], led to the development of multiparty session types [Honda et al. 2008, 2016]. Multiparty session types allow an arbitrary but statically determined number of participants ("roles") to engage in a session. The key ingredient of multiparty session types is a *global type* that defines a protocol from the perspective of the entire session, from which local types for each participant can be generated. A global type not only increases expressivity but also establishes deadlock freedom for a system consisting of a single session.

The development of GV-style session types and multiparty session types has mostly happened independently of each other. There exists no system that combines the flexibility of functional programming with the expressivity of multiparty session types. This paper introduces **Multiparty GV (MPGV)**—a linear lambda calculus with first-class multiparty sessions and dynamic thread and channel initialization. Deadlock freedom is guaranteed purely by linear type checking and an n-ary "fork" inspired by a cut-based initialization pattern. MPGV complements linear sessions with standard unrestricted functional types and language features, such as general recursive functions and algebraic data types. The integration of multiparty session types into a GV-style functional language brings a number of challenges:

Deadlock freedom. Although global types guarantee deadlock freedom for a single multiparty session, global types alone cannot guarantee deadlock freedom for interleaved multiparty sessions. To establish deadlock freedom in the presence of dynamic session spawning and session delegation, where participants can engage in several multiparty sessions simultaneously, Kobayashi-style "orders/priorities" [Kobayashi 1997, 2002, 2006] have been used to rule out cyclic dependencies among channel actions. The resulting interaction type systems [Coppo et al. 2013; Bettini et al. 2008; Coppo et al. 2016] are complementary in terms of expressivity compared to GV. They are more powerful in the sense that they allow cyclic communication topologies within and between sessions. However, well-typed programs in GV cannot be translated into these systems because orders/priorities are static and sessions are not first-class data.

In this paper we take the GV approach to deadlock freedom—MPGV features an n-ary "fork" that combines the creation of n threads and multiparty session for n participants. While this makes the MPGV type system and operational semantics simple, proving that it in fact guarantees deadlock freedom is challenging. To handle dynamic thread and channel creation, direct-style deadlock freedom proofs of GV (like those by Lindley and Morris [2015]; Fowler et al. [2021]; Jacobs et al. [2022a]) crucially rely on the communication topology remaining acyclic during program execution. For multiparty session types this is not the case—the communication topology *between* sessions is acyclic, but the communication topology *within* a session is not. The key insight of our work is to represent the cyclic communication topology within sessions as an acyclic graph at the logical level, without needing central coordination in the operational semantics.

Participant redirecting. Binary session types specify the types of data that is being sent and received, while local multiparty session types also specify the participants *names* to/from whom that data is received. These names make programming with first-class sessions non modular since the exact participants are fixed in type signatures. Suppose that one has library functions f and g such that f returns a session of a certain session type, and g expects an argument with that same session type, but with different participant names. We introduce a "redirecting" construct, which allows an endpoint to be passed to functions where different participant names are expected. Using this construct, we give a type-preserving translation from binary session types into MPGV, showing that MPGV restricted to two participants per session is at least as expressive as GV.

Mechanization. The complexities of session types, especially in the multiparty setting, and the existence of published broken proofs—including the failure of subject reduction for several multiparty systems [Scalas and Yoshida 2019a]—gave the impetus for mechanization. Whereas there exists extensive work on mechanizing the meta-theory of binary session types [Thiemann 2019; Rouvoet et al. 2020; Hinrichsen et al. 2021; Tassarotti et al. 2017; Goto et al. 2016; Ciccone and Padovani 2020; Castro-Perez et al. 2020; Gay et al. 2020], deadlock freedom for binary session types has only recently been mechanized by Jacobs et al. [2022a]. For multiparty session types, the only mechanization is Zooid by Castro-Perez et al. [2021], which mechanizes the trace semantics of a *single* multiparty session and proves that it conforms to its global type. In the spirit of this line of work, we provide a full mechanization of all our results in the Coq proof assistant.

Contributions and outline. Our main contribution is **MPGV**—the first deadlock-free linear lambda calculus with first-class multiparty sessions, dynamic thread and channel initialization, and functional features like general recursive functions and algebraic data types. Concretely:

- We explain the key ideas behind MPGV in the context of new and existing examples (§ 2).
- We formalize the type system and operational semantics of MPGV (§ 3).
- We give a type-preserving embedding of GV-style binary session types into MPGV, using our new redirecting construct, showing that MPGV goes strictly beyond binary session types (§ 4).
- We prove a combined partial deadlock and memory-leak freedom theorem for multiparty session types that also subsumes type safety and global progress (§ 5 and § 7).
- Inspired by Scalas and Yoshida [2019a], we extend MPGV with a more flexible notion of consistency that does not rely on global types (§ 6).
- We mechanize all our results in the Coq proof assistant (§ 8).

2 MPGV BY EXAMPLE

We introduce MPGV's features, based on examples (§ 2.1–§ 2.8), and provide the main intuitions for how MPGV guarantees deadlock freedom for cyclic intra-session topologies (§ 2.9).

2.1 Global and Local Types

Similar to original multiparty session types [Honda et al. 2008], sessions in MPGV can be described by a *global* type. A simple example of a global type is:

$$G \triangleq [0 \rightarrow 1]\mathbf{N}.[1 \rightarrow 2]\mathbf{N}.[2 \rightarrow 0]\mathbf{N}.\mathsf{End}.$$

This global type says that participant 0 first sends a value of natural number type $\mathbf{N}$ to participant 1, then 1 sends a $\mathbf{N}$ to 2, then 2 sends a $\mathbf{N}$ to 0, and finally the protocol ends. The global type G induces *local types* for each participant p via projections $G \downharpoonright p$:

$$G \downharpoonright 0 = \mathbf{!}[1]\mathsf{N}.\mathbf{?}[2]\mathsf{N}.\mathsf{End} \qquad G \downharpoonright 1 = \mathbf{?}[0]\mathsf{N}.\mathbf{!}[2]\mathsf{N}.\mathsf{End} \qquad G \downharpoonright 2 = \mathbf{?}[1]\mathsf{N}.\mathbf{!}[0]\mathsf{N}.\mathsf{End}$$

The local type $\mathbf{!}[p]\tau.L$ indicates that the next action should be sending a value v of type τ to participant p, to then continue with L. Dually, $\mathbf{?}[p]\tau.L$ indicates that the next action should be receiving a value v of type τ from participant p, to then continue with L. Finally, End states that the protocol has finished and the participant's endpoint should be closed.

2.2 Combined Session and Channel Initialization

With our simple global type G at hand, we now give a program that implements this global type:

$$\begin{array}{l}
\mathbf{let}\ c_0 : \mathbf{!}[1]\mathsf{N}.\mathbf{?}[2]\mathsf{N}.\mathsf{End} = \mathbf{fork}(\mathit{service}_1, \mathit{service}_2)\ \mathbf{in} \\
\mathbf{let}\ c_0 : \mathbf{?}[2]\mathsf{N}.\mathsf{End} = \mathbf{send}[1](c_0, 99)\ \mathbf{in} \\
\mathbf{let}\ (c_0, n) : \mathsf{End} \times \mathsf{N} = \mathbf{receive}[2](c_0)\ \mathbf{in} \\
\mathbf{close}(c_0)
\end{array}$$

The **fork** operation simultaneously forks off 2 threads and creates 3 channel endpoints for the participants in the session. The **fork** returns endpoint c_0 with type $G \downharpoonright 0 = \mathbf{!}[1]\mathsf{N}.\mathbf{?}[2]\mathsf{N}.\mathsf{End}$, and runs functions $\mathit{service}_1$ and $\mathit{service}_2$ (shown below) in background threads. The main thread uses $\mathbf{send}[1](c_0, 99)$ to send the message "99" to participant 1 (*i.e., service*$_1$). As is common in functional session-typed languages, the **send** and **receive** operations of MPGV return the endpoint back to us. The returned endpoint will be at a different type, because the step has been taken in the session type. For convenience, the above code let-binds the returned endpoint to the same name. The main thread then uses the operation $\mathbf{receive}[2](c_0)$ and blocks to receive a message from endpoint 2 (*i.e., service*$_2$). After the message has been received, it closes the endpoint using **close**.

Similar to many multiparty session-type systems, MPGV uses numbers for participant names in **send** and **receive** to indicate which other participant the communication concerns. Note that also for **receive** it is necessary to indicate which participant to receive from, because multiple participants could send a message to the same participant simultaneously, and these messages may have different types. The endpoint returned from **fork** has participant number 0, and endpoints of the forked-off threads have participant numbers 1, 2, *etc.* The forked-off threads could be implemented as:

$$\mathit{service}_1 : (\mathbf{?}[0]\mathsf{N}.\mathbf{!}[2]\mathsf{N}.\mathsf{End}) \to \mathbf{1} \qquad \mathit{service}_2 : (\mathbf{?}[1]\mathsf{N}.\mathbf{!}[0]\mathsf{N}.\mathsf{End}) \to \mathbf{1}$$

$$\mathit{service}_1\ c_1 \triangleq \begin{array}{l}\mathbf{let}\ (c_1, n) = \mathbf{receive}[0](c_1)\ \mathbf{in} \\ \mathbf{let}\ c_1 = \mathbf{send}[2](c_1, n+3)\ \mathbf{in} \\ \mathbf{close}(c_1)\end{array} \qquad \mathit{service}_2\ c_2 \triangleq \begin{array}{l}\mathbf{let}\ (c_2, n) = \mathbf{receive}[1](c_2)\ \mathbf{in} \\ \mathbf{let}\ c_2 = \mathbf{send}[0](c_2, n+4)\ \mathbf{in} \\ \mathbf{close}(c_2))\end{array}$$

The arguments of **fork** are closures that take the endpoint (typed with local type $G \downharpoonright p$) as argument and return the unit value when done. The first forked-off thread *service*$_1$ tries to receive a message from participant 0 (*i.e.,* the main thread), increments the received number, and passes it on to endpoint 2 (*i.e., service*$_2$). Similarly, the second forked-off thread *service*$_2$ receives a number from participant 1 (*i.e., service*$_1$), increments it, and passes it to participant 0 (*i.e.,* the main thread).

Novel elements of MPGV. The n-ary **fork** ensures that the communication topology between sessions remains acyclic. This is in contrast to original multiparty session-type systems [Honda et al. 2008], which use service names to create new sessions between already existing, concurrently running processes, selecting the participating processes non-deterministically in case there are several attempting to participate (see §9 for an in-depth discussion). By separating session creation from thread spawning in these original systems, cyclic communication topologies can be created, and hence interleaved sessions can deadlock. Inspired by binary session-typed lambda-calculi like GV [Wadler 2012; Lindley and Morris 2015] and multi-cut with coherence proofs [Carbone et al.

2015, 2016, 2017], MPGV combines session creation with thread spawning, to maintain acyclicity of the communication topology and guarantee deadlock freedom.

2.3 Interleaving and First-Class Endpoints

We now illustrate MPGV's support for session interleaving and delegation. Similar to the original versions of GV by Gay and Vasconcelos [2010]; Wadler [2012], MPGV obtains delegation without the need for special language constructs since endpoints are first class. We modify the example from §2.2, which performs its communication actions on c_0 locally, by letting the main thread fork off yet another thread to perform the communication:

$$\begin{array}{l}
\textbf{let } c_0 : G \downharpoonright 0 = \textbf{fork}(\mathit{service}_1, \mathit{service}_2) \textbf{ in} \\
\textbf{let } d_0 : G' \downharpoonright 0 = \textbf{fork}(\lambda d_1 : G' \downharpoonright 1.\ \textbf{let } (d_1, x) : (\textbf{!}[0]\mathsf{N.End}) \times (G \downharpoonright 0) = \textbf{receive}[0](d_1) \textbf{ in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{let } x : \textbf{?}[2]\mathsf{N.End} = \textbf{send}[1](x, 99) \textbf{ in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{let } (x, n) : \mathsf{End} \times \mathsf{N} = \textbf{receive}[2](x) \textbf{ in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{let } d_1 : \mathsf{End} = \textbf{send}[0](d_1, n) \textbf{ in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{close}(x); \textbf{close}(d_1)) \textbf{ in} \\
\textbf{let } d_0 : \textbf{?}[1]\mathsf{N.End} = \textbf{send}[1](d_0, c_0) \textbf{ in} \\
\textbf{let } (d_0, n) : \mathsf{End} \times \mathsf{N} = \textbf{receive}[1](d_0) \textbf{ in} \\
\textbf{close}(d_0)
\end{array}$$

To type the second **fork**, we need to come up with a second global type that governs the communication between the third forked-off thread and the main thread:

$$G' \triangleq [0 \rightarrow 1](G \downharpoonright 0).[1 \rightarrow 0]\mathsf{N.End} \qquad \text{where } G \downharpoonright 0 = \textbf{!}[1]\mathsf{N}.\textbf{?}[2]\mathsf{N.End}$$

The projections are $G' \downharpoonright 0 = \textbf{!}[1](G \downharpoonright 0).\textbf{?}[1]\mathsf{N.End}$ and $G' \downharpoonright 1 = \textbf{?}[0](G \downharpoonright 0).\textbf{!}[0]\mathsf{N.End}$. This global type shows that participant 0 (the main thread) of G' first delegates an endpoint with local type $G \downharpoonright 0$ to participant 1 of G' (the third forked-off thread), which then sends a natural number back. In the code, the main thread sends endpoint c_0, which the third forked-off thread receives as x. The third forked-off thread then executes the communication according to local type $G \downharpoonright 0$, and sends back a natural number to the main thread.

Novel elements of MPGV. As demonstrated by the above example, MPGV's session-typed endpoints are first class and can thus be sent over channels (*i.e.*, delegated) like any other data. MPGV not only allows sending single endpoints over channels, but also lists of endpoints (§2.8) or closures, which may capture endpoints. Data types that contain endpoints are treated linearly in order to protect type safety, whereas data types that cannot contain endpoints (*e.g.*, lists of natural numbers) may be freely copied and discarded. MPGV guarantees deadlock freedom in the presence of interleaved sessions solely by linear typing and n-ary fork, and without any extrinsic mechanisms like orders/priorities [Bettini et al. 2008; Coppo et al. 2013, 2016].

2.4 Participant Redirecting

In the example from §2.2 we have two threads $\mathit{service}_1$ and $\mathit{service}_2$ that were doing more or less the same thing (adding 3 and 4, respectively). To obtain a language that enables modular programming, we would like to write a single function that generalizes both services that we could use for both threads in the **fork** operation. Let us try to make an attempt:

$$\mathit{service} : \mathsf{N} \rightarrow (\textbf{?}[0]\mathsf{N}.\textbf{!}[1]\mathsf{N.End}) \rightarrow \mathbf{1}$$

$$\begin{array}{l}
\mathit{service}\ a\ c \triangleq \textbf{let } (c, n) = \textbf{receive}[0](c) \textbf{ in} \\
\qquad\qquad\quad\ \textbf{let } c = \textbf{send}[1](c, n + a) \textbf{ in} \\
\qquad\qquad\quad\ \textbf{close}(c)
\end{array}$$

The function *service* takes a natural number a for the value that should be added. Unfortunately, *service* 3 and *service* 4 cannot readily be used because their types do not match up with $G \downharpoonright 1 = \mathbf{?}[0]\mathbf{N}.\mathbf{!}[2]\mathbf{N}.\mathsf{End}$ and $G \downharpoonright 2 = \mathbf{?}[1]\mathbf{N}.\mathbf{!}[0]\mathbf{N}.\mathsf{End}$ since the participant numbers are off.

MPGV provides a $\mathbf{redirect}[\pi](c)$ operation that allows us to locally redirect participant numbers, making it possible for a programmer to pass endpoints to destinations where different participant numbers are expected in the type signature. The informal semantics of the **redirect** operation is that any $\mathbf{send}[p]$ and $\mathbf{receive}[p]$ operations on $c' = \mathbf{redirect}[\pi](c)$ get redirected to $\mathbf{send}[\pi(p)]$ and $\mathbf{receive}[\pi(p)]$ on c. With MPGV's redirect operation at hand, we can change the **fork** in the first line of the example in §2.2 into:

$$\mathbf{fork}(\lambda c_1.\ \mathit{service}\ 3\ (\mathbf{redirect}[0 \mapsto 0, 1 \mapsto 2](c_1)), \lambda c_2.\ \mathit{service}\ 4\ (\mathbf{redirect}[0 \mapsto 1, 1 \mapsto 0](c_2)))$$

Novel elements of MPGV. Redirecting is a novel concept that has not been explored in multiparty session types to our knowledge. Redirecting is important for modularity because it allows composing a function f with a function g with compatible range and domain types even when participant numbers are at odds. Redirecting is also crucial for embedding binary sessions in MPGV; without redirecting, that would not be possible (see §4).

2.5 Choice and Recursive Session Types

Similar to traditional (multiparty) session types, MPGV supports *choice* and *recursion*. For example:

$$G'' \triangleq [0 \rightarrow 1]\{A\colon \mathbf{N}.G'', B\colon \mathbf{string}.\mathsf{End}\}$$

In this global type, participant 0 sends participant 1 a choice label $\{A, B\}$. If the choice label is A, then the payload of the message is of type $\mathbf{N}$, and the protocol recursively loops back to the initial state. If the choice label is B, then the payload of the message is of type **string**, and then the protocol ends. This gives the following local projections:

$$G'' \downharpoonright 0 \triangleq \mathbf{!}[1]\{A\colon \mathbf{N}.(G'' \downharpoonright 0), B\colon \mathbf{string}.\mathsf{End}\} \quad G'' \downharpoonright 1 \triangleq \mathbf{?}[0]\{A\colon \mathbf{N}.(G'' \downharpoonright 1), B\colon \mathbf{string}.\mathsf{End}\}$$

With choice, not all global types one can write down are valid: all the branches of a choice must have *equal* projections for participants that are neither the sender nor the receiver of the choice. This is to ensure that each participant always has enough information to determine the type of the next message that they should send or expect to receive [Honda et al. 2008, 2016]. MPGV supports recursive functions, which are crucial to provide implementations of recursive session types.

2.6 Two Buyer Protocol

The two buyer protocol is a classic example from the literature [Honda et al. 2008] with two buyers (Alice and Bob) and a Seller. The protocol has the following global type in MPGV (we use symbolic participant identifiers for readability; one can take $S = 0$, $A = 1$, $B = 2$):

$$\begin{aligned} G_{SAB} \triangleq\ & [A \rightarrow S]\mathbf{string}.[S \rightarrow A]\mathbf{N}.\ [S \rightarrow B]\mathbf{N}.[A \rightarrow B]\mathbf{N}. \\ & [B \rightarrow S]\{\mathit{Yes} : [S \rightarrow B]\mathbf{date}.\ \mathsf{End}, \mathit{No} : \mathsf{End}\} \end{aligned}$$

This global protocol has the following projections for Alice, Bob, and Seller:

$$\begin{aligned} G_{SAB} \downharpoonright A &= \mathbf{!}[S]\mathbf{string}.\mathbf{?}[S]\mathbf{N}.\mathbf{!}[B]\mathbf{N}.\mathsf{End} \\ G_{SAB} \downharpoonright B &= \mathbf{?}[S]\mathbf{N}.\mathbf{?}[A]\mathbf{N}.\mathbf{!}[S]\{\mathit{Yes} : \mathbf{?}[S]\mathbf{date}.\mathsf{End}, \mathit{No} : \mathsf{End}\} \\ G_{SAB} \downharpoonright S &= \mathbf{?}[A]\mathbf{string}.\mathbf{!}[A]\mathbf{N}.\mathbf{!}[B]\mathbf{N}.\mathbf{?}[B]\{\mathit{Yes} : \mathbf{!}[B]\mathbf{date}.\mathsf{End}, \mathit{No} : \mathsf{End}\} \end{aligned}$$

The participants perform the following interactions:

(1) Alice tells the Seller which item she wants to buy ($[A \rightarrow S]\mathbf{string}$).
(2) The Seller tells both Alice and Bob how much the item costs ($[S \rightarrow A]\mathbf{N}.\ [S \rightarrow B]\mathbf{N}$).

(3) Alice tells Bob how much money she is willing to contribute to the purchase ($[A \to B]\mathbf{N}$).
(4) Bob decides whether they can afford the item, and informs the Seller of his decision ($[B \to S]\{Yes : \ldots, No : \ldots\}$).
(5) If Bob says *Yes*, the Seller sends Bob the date at which the item will be delivered and then ends the protocol ($[S \to B]\mathbf{date}.\mathsf{End}$).
(6) If Bob says *No*, the protocol ends immediately (End).

A possible implementation of the Seller is as follows:

$seller : G_{SAB} \upharpoonright S \to \mathbf{1}$

```
seller $c_S$ $\triangleq$
  let $(c_S, item) : (![A]\mathbf{N}.![B]\mathbf{N}.?[B]\{Yes : ![B]\mathbf{date}.\mathsf{End}, No : \mathsf{End}\}) \times \mathbf{string}$ = receive$[A](c_S)$ in
  let $c_S : ![B]\mathbf{N}.?[B]\{Yes : ![B]\mathbf{date}.\mathsf{End}, No : \mathsf{End}\})$ = send$[A](c_S, cost(item))$ in
  let $c_S : ?[B]\{Yes : ![B]\mathbf{date}.\mathsf{End}, No : \mathsf{End}\})$ = send$[B](c_S, cost(item))$ in
  match receive$[B](c_S)$ with {
    $\langle Yes{:}c_S : ![B]\mathbf{date}.\mathsf{End}\rangle \mapsto$ let $c_S : \mathsf{End}$ = send$[B](c_S, date(item))$ in close$(c_S)$
    $\langle No{:}c_S : \mathsf{End}\rangle \mapsto$ close$(c_S)$
  }
```

In the case $\langle Yes{:}c_S\rangle$, we have $c_S : ![B]\mathbf{date}.\mathsf{End}$, whereas in case $\langle No{:}c_S\rangle$ we have $c_S : \mathsf{End}$, so the type of the endpoint depends on which choice was made by Bob. Assuming that we also have functions $alice : G_{SAB} \upharpoonright A \to \mathbf{1}$ and $bob : G_{SAB} \upharpoonright B \to \mathbf{1}$ for Alice and Bob, we can run the two buyer protocol with program $seller\ (\mathbf{fork}(alice,\ bob))$.

2.7 Three Buyer Protocol and Session Delegation

The two buyer example has been extended with delegation by Honda et al. [2008]. To help Alice and Bob, there is a fourth person, Carol. If Bob and Alice cannot afford the item together, then instead of replying *No* to the Seller, Bob will send the remainder of his session to Carol (*i.e.,* delegation). Carol will then respond *Yes* to the Seller, if the three of them together have enough money. This is modeled by a separate session between Bob and Carol with global type:

$$G_{BC} \triangleq [B \to C](\mathbf{N} \times ![S]\{Yes : ?[S]\mathbf{date}.\mathsf{End}, No : \mathsf{End}\}).\mathsf{End}$$

Because Bob needs access to Carol, his function is parameterized by that endpoint c_C as well as his own endpoint c_B in the two buyer protocol between him, Alice, and the Seller:

$bob_{del} : G_{BC} \upharpoonright B \to G_{SAB} \upharpoonright B \to \mathbf{1}$

```
$bob_{del}\ c_C\ c_B$ $\triangleq$ let $(c_B, cost) : (?[A]\mathbf{N}.![S]\{Yes : ?[S]\mathbf{date}.\mathsf{End}, No : \mathsf{End}\}) \times \mathbf{N}$ = receive$[S](c_B)$ in
              let $(c_B, contrib_A) : (![S]\{Yes : ?[S]\mathbf{date}.\mathsf{End}, No : \mathsf{End}\}) \times \mathbf{N}$ = receive$[A](c_B)$ in
              if $cost - contrib < max_B$ then
                let $c_B : ?[S]\mathbf{date}.\mathsf{End}$ = send$[S](c_B, \langle Yes\rangle)$ in
                let $(c_B, date) : \mathsf{End} \times \mathbf{date}$ = receive$[S](c_B)$ in
                close$(c_B)$
              else
                let $c_C : \mathsf{End}$ = send$[C](c_C, (cost - contrib_A - max_B, c_B))$ in
                close$(c_C)$
```

In the else branch, Bob sends his endpoint c_B over his connection to Carol, c_C. We can run the three buyer protocol with the following program, assuming that we have $carol : G_{BC} \upharpoonright C \to \mathbf{1}$:

```
let $c_C : G_{BC} \upharpoonright B$ = fork$(carol)$ in
$seller\ (\mathbf{fork}(alice,\ bob_{del}\ c_C))$
```

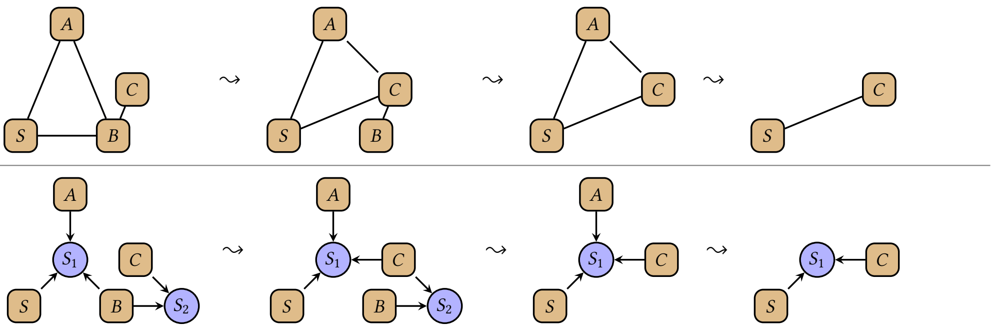

Fig. 1. Steps in three buyer protocol. Top: physical communication paths; bottom: logical connectivity.

Depending on thread scheduling, operations can be executed in a different order. One possible execution is graphically depicted in the top row of Figure 1. In the left picture, we have the session between A, B, and S, and the session between B and C. In our operational semantics, the participants are connected directly, and each participant has their own set of buffers in the heap, separate from the others. At some point Bob decides to send his session to Carol (second picture), so the connections of B get moved to C. Bob then ends his session with Carol (third picture). Alice ends her participation in the session (fourth picture). This deletes her buffers from the heap, even though the Seller and Carol may still be actively communicating. The global type ensures that whenever Alice is allowed to close her session, the other participants are guaranteed not to perform further communication with her.

2.8 Endpoints in Data Structures

Because of the functional nature of MPGV, we can freely intermix sessions and data structures. We give an example of a department store, to which we can send several buyers in a list. The department store will then let the buyers interact by applying the *seller* function for us. To illustrate recursive protocols, the department store loops around and accepts new buyers:

$$\mathit{departmentstore} : (\mu x.\, ?[C]\mathsf{List}(G_{SAB} \downharpoonleft S).x) \to \mathbf{1}$$

$$\mathit{departmentstore}\ c_D \triangleq \mathbf{let}\ (c_D, \mathit{endpoints}) = \mathbf{receive}[C](c_D)\ \mathbf{in}$$
$$\qquad\qquad \mathsf{map}\ \mathit{seller}\ \mathit{endpoints};\ \mathit{departmentstore}\ c_D$$

Given a function $\mathit{buyers} : \mathbf{string} \to G_{SAB} \downharpoonleft S$ that starts up the two or three buyers trying to buy an item of the given name and returns the seller's endpoint to interact with them, we can start a department store and send buyers to it as follows:

$\mathbf{let}\ \mathit{store} = \mathbf{fork}(\mathit{departmentstore})\ \mathbf{in}$
$\mathbf{let}\ c_1 = \mathit{buyers}\ \text{“}\mathit{hat}\text{”}\ \mathbf{in}$
$\mathbf{let}\ c_2 = \mathit{buyers}\ \text{“}\mathit{cow}\text{”}\ \mathbf{in}$
$\mathbf{let}\ \mathit{store} = \mathbf{send}[D](\mathit{store}, [c_1; c_2])\ \mathbf{in}$
$\mathbf{let}\ c_3 = \mathit{buyers}\ \text{“}\mathit{egg}\text{”}\ \mathbf{in}$
$\mathbf{let}\ c_3 = \mathit{buyers}\ \text{“}\mathit{bow}\text{”}\ \mathbf{in}$
$\mathbf{let}\ \mathit{store} = \mathbf{send}[D](\mathit{store}, [c_3; c_4])\ \mathbf{in}\ \ldots$

Novel elements of MPGV. MPGV allows *multiparty* endpoints to be stored in data structures, and captured in closures, which can then be sent as messages. This is in contrast to earlier multiparty systems, where endpoints can either not be manipulated at all [Castro-Perez et al. 2021], or where there is a separate syntactic category for endpoints, which cannot be mixed with data [Honda et al. 2008; Coppo et al. 2016; Bettini et al. 2008; Coppo et al. 2013].

2.9 Deadlock Freedom of MPGV

MPGV's deadlock freedom proof is based on two key ideas: (1) local progress *within* a session is guaranteed by the global type, and (2) global progress *between* sessions is guaranteed by our n-ary fork and linear typing, asserting that the communication topology between sessions remains acyclic (despite first-class endpoints). To reason about deadlock freedom we abstract a *logical connectivity* topology from the *physical communication* topology and prove that the logical connectivity topology remains acyclic. The logical topology of the three buyer protocol is depicted in the bottom row of Figure 1. It introduces a blue circle for each multiparty session, abstracting over the cyclic topology within a session and exposing the acyclicity of the logical topology. Figure 1 shows that the logical connectivity topology remains acyclic throughout the execution. This holds for any well-typed MPGV program—Figure 10 in § 7 shows how the logical topology is transformed and remains acyclic for each of the session operations.

Novel elements of MPGV. Similar to binary variants of GV, MPGV ensures global progress and deadlock freedom for an entire program, solely by linear typing. In contrast, earlier multiparty systems either guarantee deadlock freedom only for a single session [Castro-Perez et al. 2021; Honda et al. 2008], or for multiple sessions if types are augmented with extrinsic orders/priorities [Coppo et al. 2016; Bettini et al. 2008; Coppo et al. 2013]. Moreover, our global progress and deadlock freedom theorems are mechanized in Coq (§ 5).

3 THE SEMANTICS OF MPGV

3.1 Syntax and Operational Semantics

Each configuration in our small-step operational semantics consists of a *thread pool* and *heap*, which stores a vector of buffers for each endpoint:

$$\rho \in \mathit{Cfg} \triangleq \mathit{List}\ \mathit{Expr} \times \mathit{Heap} \qquad h \in \mathit{Heap} \triangleq \mathit{Endpoint} \xrightarrow{\text{fin}} (\mathit{Participant} \xrightarrow{\text{fin}} \mathit{List}\ (\mathit{Label} \times \mathit{Val}))$$

An endpoint $c \in \mathit{Endpoint} ::= (s, p)$ consists of a number $s \in \mathit{Session}$ identifying the session, and a number $p \in \mathit{Participant}$ identifying the participant number of the endpoint in the session.

The operational semantics has three reduction relations. Firstly, $e \rightsquigarrow_{\text{pure}} e'$ for pure reductions of expressions. Secondly, $(e, h) \rightsquigarrow_{\text{head}} (e', h', \vec{e})$ for reductions of channel operations involving the heap h, with the option to spawn a list of new threads $\vec{e}$ (a non-empty list for **fork**, and an empty list for the other operations). Thirdly, $(\vec{e}, h) \rightsquigarrow_{\text{cfg}} (\vec{e}', h')$ between configurations, which performs $\rightsquigarrow_{\text{head}}$ on some thread in the thread pool, and also handles evaluation contexts. The formal syntax and operational semantics of MPGV can be found in Figure 2. We give an informal description of the semantics of the message-passing operations **fork**, **send**, **receive**, **close**, and **redirect** next.

Fork. The fork operation $\textbf{fork}(v_1, \ldots, , v_n)$ spawns n threads and creates a new session between the $n + 1$ endpoints. The session s has $(n + 1) \times (n + 1)$ buffers in the heap h for the $n + 1$ endpoints, such that the buffer stored at $h(s, q)(p)$ queues messages sent from p to q. Session endpoints c are represented as triples $c = \#[(s, p), \pi]$ of a session address $s \in \mathit{Session}$, endpoint number $p \in \mathit{Participant}$, and translation vector $\pi : \mathit{Participant} \xrightarrow{\text{fin}} \mathit{Participant}$, which is used for redirecting and initialized by **fork** to be the identity mapping. Each of the values v_i passed as arguments to **fork** must be a closure that accepts an endpoint as its argument, so that the threads run function calls $v_i\ \#[(s, i), \text{id}]$ for $i = 1..n$. The **fork** returns endpoint $\#[(s, 0), \text{id}]$. A usage pattern is:

$$\textbf{let}\ c_0 = \textbf{fork}((\lambda c_1.\ e_1), \ldots, (\lambda c_n.\ e_n))\ \textbf{in}\ e_0$$

Expressions, values, and evaluation contexts

$$
\begin{aligned}
e \in \mathit{Expr} &::= x \mid () \mid n \mid (e,e) \mid \langle \ell : e \rangle \mid \lambda x.\, e \mid \mathbf{rec}\ f\ x.\, e \mid e\ e \mid \mathbf{fork}(e,\ldots,e) \mid \mathbf{send}[p](e, \ell : e) \mid \\
&\quad \mathbf{receive}[p](e) \mid \mathbf{close}(e) \mid \mathbf{redirect}[\pi](e) \mid \mathbf{let}\ x = e\ \mathbf{in}\ e \mid \\
&\quad \mathbf{let}\ (x_1, x_2) = e\ \mathbf{in}\ e \mid \mathbf{match}\ e\ \mathbf{with}\ \{\langle \ell : x \rangle \mapsto e;\ \ldots\}_{\ell \in I} \\
v \in \mathit{Val} &::= () \mid n \mid (v,v) \mid \langle \ell : v \rangle \mid \lambda x.\, e \mid \mathbf{rec}\ f\ x.\, e \mid \#[c, \pi] \\
K \in \mathit{Ctx} &::= \square \mid (K, e) \mid (v, K) \mid K\ e \mid v\ K \mid \mathbf{let}\ x = K\ \mathbf{in}\ e \mid \cdots
\end{aligned}
$$

Data structures

$$
\begin{aligned}
s \in \mathit{Session} &\triangleq \mathbb{N} & c \in \mathit{Endpoint} &\triangleq \mathit{Session} \times \mathit{Participant} \\
p, q \in \mathit{Participant} &\triangleq \mathbb{N} & \pi \in \mathit{Translation} &\triangleq \mathit{Participant} \xrightarrow{\text{fin}} \mathit{Participant} \\
\ell \in \mathit{Label} &\triangleq \mathbb{N} & h \in \mathit{Heap} &\triangleq \mathit{Endpoint} \xrightarrow{\text{fin}} (\mathit{Participant} \xrightarrow{\text{fin}} \mathit{List}\ (\mathit{Label} \times \mathit{Val})) \\
& & \rho \in \mathit{Cfg} &\triangleq \mathit{List}\ \mathit{Expr} \times \mathit{Heap}
\end{aligned}
$$

Small-step operational semantics

$$
\begin{aligned}
(e_1, h) &\rightsquigarrow_{\text{head}} (e_2, h, \epsilon) \quad (\text{if } e_1 \rightsquigarrow_{\text{pure}} e_2) \\
(\mathbf{fork}(v_1, \ldots, v_n), h) &\rightsquigarrow_{\text{head}} (\#[(s,0), \mathsf{id}], h \uplus \{(s,0) \mapsto \vec{\epsilon}, \ldots, (s,n) \mapsto \vec{\epsilon}\}, \\
&\qquad [v_1\ \#[(s,1), \mathsf{id}], \ldots, v_n\ \#[(s,n), \mathsf{id}]]) \\
(\mathbf{send}[q](\#[(s,p), \pi], \ell : v), h) &\rightsquigarrow_{\text{head}} (\#[(s,p), \pi], \mathsf{push}((s, \pi(q)), p, \langle \ell : v \rangle, h), \epsilon) \\
(\mathbf{receive}[p](\#[(s,q), \pi]), h) &\rightsquigarrow_{\text{head}} (\langle \ell : (v, \#[(s,q), \pi]) \rangle, h', \epsilon) \\
&\qquad\qquad (\text{if } \mathsf{pop}((s,q), \pi(p), h) = (\langle \ell : v \rangle, h')) \\
(\mathbf{close}(\#[(s,p), \pi]), h) &\rightsquigarrow_{\text{head}} ((), h \backslash \{(s,p)\}, \epsilon) \\
(\mathbf{redirect}[\pi_1](\#[(s,p), \pi_2]), h) &\rightsquigarrow_{\text{head}} (\#[(s,p), \pi_2 \circ \pi_1], h, \epsilon) \\
(\vec{e_a} +\!\!+ [K[\,e\,]] +\!\!+ \vec{e_b}, h) &\rightsquigarrow_{\text{cfg}} (\vec{e_a} +\!\!+ [K[\,e'\,]] +\!\!+ \vec{e_b} +\!\!+ \vec{e}, h') \quad (\text{if } (e,h) \rightsquigarrow_{\text{head}} (e', h', \vec{e}))
\end{aligned}
$$

Fig. 2. Syntax and operational semantics of MPGV (selected rules).

Send. The send operation $\mathbf{send}[q](c, \ell : v)$ sends the message $\langle \ell : v \rangle$ to q via the endpoint $c = \#[(s,p), \pi]$ by adding the message to the end of buffer (using the operation $\mathsf{push}((s, \pi(q)), p, \langle \ell : v \rangle, h)$ in Figure 2). The message is tagged with a label ℓ, which can influence the future actions allowed to be performed by the participant. We revisit this in detail when we introduce the typing rules. Our send operation is asynchronous. One can encode synchronous communication by inserting after each message $A \rightarrow B$ a dummy message $B \rightarrow A$ with type unit to enforce synchronization.

Receive. The receive operation $\mathbf{receive}[p](c)$ receives a message from p via endpoint $c = \#[(s,q), \pi]$. The receive operation takes the first message out of buffer (using the operation $\mathsf{pop}((s,q), \pi(p), h) = (\langle \ell : v \rangle, h')$ in Figure 2). If the buffer is empty, the operation blocks until a message becomes available.

Close. The close operation $\mathbf{close}(c)$ deletes all the buffers from which the endpoint $c = \#[(s,q), \pi]$ receives messages, that is, it simply deletes entry $h((s,q))$ of the heap.

Redirect. The redirecting operation $c' = \mathbf{redirect}[\pi](c)$ where $\pi \in \mathit{Participant} \xrightarrow{\text{fin}} \mathit{Participant}$ redirects messages so that send and receive operations to p on c' are redirected to $\pi(p)$ on c.

$$\frac{\Gamma\ \mathsf{unr} \qquad x \notin \Gamma}{\{x \mapsto \tau\} \cup \Gamma \vdash x : \tau} \qquad \frac{\Gamma_1 \perp \Gamma_2 \qquad \Gamma_1 \vdash e_1 : \tau_1 \Rightarrow \tau_2 \qquad \Gamma_2 \vdash e_2 : \tau_1 \qquad (\Rightarrow) \in \{\rightarrow, \multimap\}}{\Gamma_1 \cup \Gamma_2 \vdash e_1\ e_2 : \tau_2}$$

$$\frac{\Gamma \cup \{x \mapsto \tau_1\} \vdash e : \tau_2 \qquad x \notin \Gamma}{\Gamma \vdash \lambda x.\ e : \tau_1 \multimap \tau_2} \qquad \frac{\Gamma \cup \{x \mapsto \tau_1\} \vdash e : \tau_2 \qquad \Gamma\ \mathsf{unr} \qquad x \notin \Gamma}{\Gamma \vdash \lambda x.\ e : \tau_1 \rightarrow \tau_2}$$

$$\frac{\Gamma \cup \{f \mapsto (\tau_1 \rightarrow \tau_2), x \mapsto \tau_1\} \vdash e : \tau_2 \qquad \Gamma\ \mathsf{unr} \qquad f, x \notin \Gamma}{\Gamma \vdash \mathbf{rec}\ f\ x.\ e : \tau_1 \rightarrow \tau_2} \qquad \frac{\Gamma \vdash e : \tau_\ell}{\Gamma \vdash \langle \ell : e \rangle : \Sigma_{\ell \in I}.\ \tau_\ell}$$

$$\frac{\Gamma_1 \perp \Gamma_2 \qquad \Gamma_1 \vdash e : \Sigma_{\ell \in I}.\ \tau_\ell \qquad \forall \ell \in I.\ \ \Gamma_2 \cup \{x_\ell \mapsto \tau_\ell\} \vdash e_\ell : \tau' \quad x_\ell \notin \Gamma_2 \quad (I = \emptyset \implies \Gamma_2 = \emptyset)}{\Gamma_1 \cup \Gamma_2 \vdash \mathbf{match}\ e\ \mathbf{with}\ \{\langle \ell : x_\ell \rangle \mapsto e_\ell;\ \ldots\}_{\ell \in I} : \tau'}$$

$$\frac{\Gamma_1 \perp \cdots \perp \Gamma_n \qquad \mathsf{consistent}(L_0, L_1, \ldots, L_n) \qquad \forall p \in \{1..n\}.\ \Gamma_p \vdash e_p : L_p \multimap \mathbf{1}}{\Gamma_1 \cup \cdots \cup \Gamma_n \vdash \mathbf{fork}(e_1, \ldots, e_n) : L_0} \qquad \frac{\Gamma \vdash e : \mathsf{End}}{\Gamma \vdash \mathbf{close}(e) : \mathbf{1}}$$

$$\frac{\Gamma_1 \perp \Gamma_2 \qquad \Gamma_1 \vdash e_1 : \mathbf{!}[p]\{\ell : \tau_\ell.\ L_\ell\}_{\ell \in I} \qquad \Gamma_2 \vdash e_2 : \tau_\ell}{\Gamma_1 \cup \Gamma_2 \vdash \mathbf{send}[p](e_1, \ell : e_2) : L_\ell} \qquad \frac{\Gamma \vdash e : \mathbf{?}[p]\{\ell : \tau_\ell.\ L_\ell\}_{\ell \in I}}{\Gamma \vdash \mathbf{receive}[p](e) : \Sigma_{\ell \in I}.\ \tau_\ell \times L_\ell}$$

$$\frac{\Gamma \vdash e : \pi(L)}{\Gamma \vdash \mathbf{redirect}[\pi](e) : L}$$

Fig. 3. Selected MPGV typing rules.

Operationally, this composes the translation vector of c with π:

$$\mathbf{redirect}[\pi_1](\#[(s, p), \pi_2]) = \#[(s, p), \pi_2 \circ \pi_1]$$

For details, see Figure 2. This operation is required to make multiparty sessions formally subsume binary sessions (§4), but is independently useful for modular programming with first-class endpoints (§2.4), because it allows the programmer to pass endpoints to destinations where different endpoint numbers are expected in the type signature.

3.2 Static Type System

The functional layer of MPGV features base types, products, closures, sums, and equi-recursive types [Crary et al. 1999]. The message-passing layer of MPGV features multiparty sessions with n-ary choice. Formally the types of MPGV are given by:

$$\begin{aligned} \tau \in \mathit{Type} \underset{(\mathit{coind})}{::=} & \ \mathbf{1} \mid \mathbf{N} \mid \tau \times \tau \mid \tau \multimap \tau \mid \tau \rightarrow \tau \mid \Sigma_{\ell \in I}.\ \tau_\ell \mid L \\ L \in \mathit{LType} \underset{(\mathit{coind})}{::=} & \ \mathbf{!}[p]\{\ell : \tau_\ell.\ L_\ell\}_{\ell \in I} \mid \mathbf{?}[p]\{\ell : \tau_\ell.\ L_\ell\}_{\ell \in I} \mid \mathsf{End} \end{aligned}$$

The functional types τ and local session types L are mutually defined: functional types occur as messages in local types, and local types are functional types. To support equi-recursive types, we define *Type* and *LType coinductively*, allowing types to refer to themselves [Crary et al. 1999; Gay et al. 2020; Jacobs et al. 2022a; Castro-Perez et al. 2021; Keizer et al. 2021]. Mutually recursive functional types and local types can be constructed using corecursion in the meta logic (*i.e.,* Coq), so there is no explicit recursion operator. We use = to denote coinductive equivalence (*i.e.,* bisimulation). The typing rules for MPGV's judgment $\Gamma \vdash e : \tau$ are displayed in Figure 3.

3.2.1 Unrestricted Types. We have *linear* function types $\tau_1 \multimap \tau_2$, which must be used exactly once, and whose lambda expressions can capture linear data. We also have *unrestricted* functions $\tau_1 \rightarrow \tau_2$, which can be used any number of times (incl. zero times), but whose lambda expressions cannot capture linear data. We define the subset $\mathit{UType} \subseteq \mathit{Type}$ of unrestricted types as:

$$\tilde{\tau} \in \mathit{UType} \underset{(coind)}{::=} \mathbf{1} \mid \mathbf{N} \mid \tilde{\tau} \times \tilde{\tau} \mid \tau \rightarrow \tau \mid \Sigma_{\ell \in I}.\ \tilde{\tau}_\ell$$

Note that $\tau_1 \rightarrow \tau_2$ is always unrestricted, even if τ_1 and τ_2 are restricted, because closures of unrestricted function type cannot contain endpoints.

To support linear and unrestricted types in the typing judgment, context disjointness $\Gamma_1 \perp \Gamma_2$ is defined such that if Γ_1 and Γ_2 both contain variable x, the two contexts must assign equal types to x (*i.e.,* $\Gamma_1(x) = \Gamma_2(x)$), and the type they assign to x must be an unrestricted type. This ensures that the union operation $\Gamma_1 \cup \Gamma_2$ on contexts is well-defined whenever it is used in the typing rules (for instance, if $\Gamma_1 = \{x : \mathbf{N}; y : \mathbf{N}\}$ and $\Gamma_2 = \{y : \mathbf{N}\}$, then $\Gamma_1 \cup \Gamma_2 = \{x : \mathbf{N}; y : \mathbf{N}\}$). A context is unrestricted if all its types are unrestricted.

3.2.2 Local Types. Local types describe the protocol that an endpoint c must follow:

- If $c : \mathbf{!}[p]\{\ell : \tau_\ell.\ L_\ell\}_{\ell \in I}$ then the next action on c has to be $\mathbf{send}[p](c, \ell : v)$, and $v : \tau_\ell$ and the continuation type L_ℓ of c is determined by the sent label $\ell \in I$.
- If $c : \mathbf{?}[p]\{\ell : \tau_\ell.\ L_\ell\}_{\ell \in I}$ then the next action on c has to be $\mathbf{receive}[p](c)$, and the received label $\ell \in I$ determines the type τ_ℓ of the value received and the next type L_ℓ of c.
- If $c : \mathsf{End}$ then the next action on c must be $\mathbf{close}(c)$.

Due to linear typing of endpoints, we must use each endpoint variable exactly once. Like in other session typed languages, this is necessary for type safety.

For the typing rule of **redirect** (if $e : \pi(L)$, then $\mathbf{redirect}[\pi](e) : L$), we define the action of a renaming π (not necessarily injective) on local types:

$$\begin{aligned} \pi(\mathbf{!}[p]\{\ell : \tau_\ell.\ L_\ell\}_{\ell \in I}) &\triangleq \mathbf{!}[\pi(p)]\{\ell : \tau_\ell.\ \pi(L_\ell)\}_{\ell \in I} \\ \pi(\mathbf{?}[p]\{\ell : \tau_\ell.\ L_\ell\}_{\ell \in I}) &\triangleq \mathbf{?}[\pi(p)]\{\ell : \tau_\ell.\ \pi(L_\ell)\}_{\ell \in I} \\ \pi(\mathsf{End}) &\triangleq \mathsf{End} \end{aligned}$$

3.2.3 Global Types and Projections. The typing rule for **fork** (Figure 3) requires a session's local types $L_0, \ldots, L_n$ to be *consistent.* Consistency means, for instance, that if participant p sends a value of type τ to participant q, then q is expecting to receive a value of type τ from p at that point in the protocol. Traditionally, consistency is defined by the existence of a global type that governs the communication between all participants in a session. Global types are of the form:

$$G \in \mathit{GType} \underset{(coind)}{::=} [p_1 \rightarrow p_2]\{\ell : \tau_\ell.\ G_\ell\}_{\ell \in I} \mid \mathsf{End}$$

A global type $[p_1 \rightarrow p_2]\{\ell : \tau_\ell.\ G_\ell\}_{\ell \in I}$ expresses that the first action in the protocol is for participant p_1 to send a message to p_2, such that if the label in the message is chosen to be ℓ, then the payload of the message has to have type τ_ℓ, and then the global protocol continues as G_ℓ. Note that "global" in our use of "global type" means global with respect to a session, not the whole program—each different session started by a **fork** can have its own global type.

Local types can be extracted from global types by a *projection judgment* $G \downharpoonright p = L$, indicating that participant p's local type is L if the global type is G. The judgment is coinductively defined in Figure 4. The first two rules state how the sender and receiver of a message in the global type are

$$\frac{r \neq q \qquad \forall \ell \in I.\, G_\ell \downharpoonleft r = L_\ell}{[r \rightarrow q]\{\ell : \tau_\ell.\, G_\ell\}_{\ell \in I} \downharpoonleft r = \,![q]\{\ell : \tau_\ell.\, L_\ell\}_{\ell \in I}} \qquad \frac{r \neq p \qquad \forall \ell \in I.\, G_\ell \downharpoonleft r = L_\ell}{[p \rightarrow r]\{\ell : \tau_\ell.\, G_\ell\}_{\ell \in I} \downharpoonleft r = \,?[p]\{\ell : \tau_\ell.\, L_\ell\}_{\ell \in I}}$$

$$\frac{r \notin \{p, q\} \qquad \forall \ell \in I.\, G_\ell \downharpoonleft r = L \qquad r \text{ guards } G_\ell \qquad I \neq \emptyset}{[p \rightarrow q]\{\ell : \tau_\ell.\, G_\ell\}_{\ell \in I} \downharpoonleft r = L} \qquad \frac{r \notin \mathsf{participants}(G)}{G \downharpoonleft r = \mathsf{End}}$$

$$\frac{r \in \{p, q\}}{r \text{ guards } [p \rightarrow q]\{\ell : \tau_\ell.\, G_\ell\}_{\ell \in I}} \qquad \frac{\forall \ell \in I.\, r \text{ guards } G_\ell}{r \text{ guards } [p \rightarrow q]\{\ell : \tau_\ell.\, G_\ell\}_{\ell \in I}}$$

Fig. 4. Coinductive projection rules (dotted line) and inductive guardedness rules (solid line).

projected. The third rule states how other participants not involved in the message are projected. For participants not involved in the message, we require that participant to *guard* the rest of the global type, which means that the participant occurs in the global type at finite depth along every branch. The fourth rule states that if a participant does not occur in the global type, then it projects to End. Our projection rules are similar to those of Zooid [Castro-Perez et al. 2021].

Traditionally, consistency $\mathsf{consistent}(L_0, \ldots, L_n)$ is expressed in terms of a global type G such that $G \downharpoonleft 0 = L_0, \ldots, G \downharpoonleft n = L_n$, and $G \downharpoonleft m = \mathsf{End}$ for $m > n$. In §6 we develop, inspired by Scalas and Yoshida [2019a], a more permissive notion of consistency that is independent of a global type, permitting deadlock-free scenarios for which no appropriate global type can be found. §6.2 then shows that the traditional notion of consistency based on global types implies our new notion.

4 TRANSLATION FROM BINARY TO MULTIPARTY

We show that a GV-style *binary* session-typed language falls out as a special mode of use of our *multiparty* language MPGV by giving a *type-preserving translation* of binary channel operations into MPGV. We consider this an important benchmark, because whereas traditional multiparty systems do support such a translation—the output of the translation does not necessarily fall into the fragment of the language where the type system ensures deadlock freedom if whole programs instead of single sessions are considered. There are two main obstacles in existing systems: (1) after translation, participant numbers do not match up, and (2) in systems such as Coppo et al. [2013]; Bettini et al. [2008]; Coppo et al. [2016], deadlock-freedom mechanisms such as orders/priorities prevent programs from being translated because these orders are absent in the source program, so after translation one must come up with an order on sessions. The latter is not always possible if sessions are used in a different orders in different branches of a conditional. Finally, translation of an expressive language such as GV requires the target language to support storing endpoints in data structures, as GV supports this. MPGV overcomes all these obstacles.

We start by giving a short introduction to binary session types, and then show how they can be translated into our language, making use of **redirect**. Binary session types are equivalent to local types *without* participant annotations. The annotations are not necessary in the binary case, because there is only one other participant to communicate with:

$$B \in BType \underset{(coind)}{::=} \,!\{\ell : \tau_\ell.\, B_\ell\}_{\ell \in I} \mid \,?\{\ell : \tau_\ell.\, B_\ell\}_{\ell \in I} \mid \mathsf{End}$$

The operations for binary channels are defined in terms of multiparty operations as follows:

$$\mathbf{fork}_B(e) \triangleq \mathbf{redirect}[1 \mapsto 0](\mathbf{fork}(e)) \qquad \mathbf{send}_B(e_1, \ell{:}e_2) \triangleq \mathbf{send}[0](e_1, \ell{:}e_2)$$
$$\mathbf{close}_B(e) \triangleq \mathbf{close}(e) \qquad \mathbf{receive}_B(e) \triangleq \mathbf{receive}[0](e)$$

$$\frac{\Gamma \vdash e : [\![\overline{B}]\!]_L \multimap \mathbf{1}}{\Gamma \vdash \mathbf{fork}_B(e) : [\![B]\!]_L} \qquad \frac{\Gamma \vdash e : [\![\mathsf{End}]\!]_L}{\Gamma \vdash \mathbf{close}_B(e) : \mathbf{1}} \qquad \frac{\Gamma_1 \perp \Gamma_2 \qquad \Gamma_1 \vdash e_1 : [\![\mathbf{!}\{\ell : \tau_\ell .\ B_\ell\}_{\ell \in I}]\!]_L \quad \Gamma_2 \vdash e_2 : \tau_\ell}{\Gamma_1 \cup \Gamma_2 \vdash \mathbf{send}_B(e_1, \ell : e_2) : [\![B_\ell]\!]_L}$$

$$\frac{\Gamma \vdash e : [\![\mathbf{?}\{\ell : \tau_\ell .\ B_\ell\}_{\ell \in I}]\!]_L}{\Gamma \vdash \mathbf{receive}_B(e) : \Sigma_{\ell \in I}.\ \tau_\ell \times [\![B_\ell]\!]_L}$$

Fig. 5. Derivable typing rules for binary session types.

We do a binary spawn using the n-ary fork, then the local type of the endpoint of the spawner gets annotated with 1's (because it is communicating with endpoint 1) and the local type of the endpoint of the forked-off thread gets annotated with 0's (because it is communicating with endpoint 0). In order to implement a type-preserving translation, we redirect all annotations to 0. This enables us to canonically translate binary session types B to multiparty local types $[\![B]\!]_L$ by using $p = 0$ for every participant annotation:

$$\begin{aligned} [\![\mathbf{!}\{\ell : \tau_\ell .\ B_\ell\}_{\ell \in I}]\!]_L &\triangleq \mathbf{!}[0]\{\ell : \tau_\ell .\ [\![L_\ell]\!]_L\}_{\ell \in I} \\ [\![\mathbf{?}\{\ell : \tau_\ell .\ B_\ell\}_{\ell \in I}]\!]_L &\triangleq \mathbf{?}[0]\{\ell : \tau_\ell .\ [\![L_\ell]\!]_L\}_{\ell \in I} \\ [\![\mathsf{End}]\!]_L &\triangleq \mathsf{End} \end{aligned}$$

We then prove that the usual typing rules for binary session types are derivable in our system. For **fork**, this amounts to defining a global type $[\![B]\!]_G$ to govern the binary interaction:

$$\begin{aligned} [\![\mathbf{!}\{\ell : \tau_\ell .\ B_\ell\}_{\ell \in I}]\!]_G &\triangleq [0 \to 1]\{\ell : \tau_\ell .\ [\![B_\ell]\!]_G\}_{\ell \in I} \\ [\![\mathbf{?}\{\ell : \tau_\ell .\ B_\ell\}_{\ell \in I}]\!]_G &\triangleq [1 \to 0]\{\ell : \tau_\ell .\ [\![B_\ell]\!]_G\}_{\ell \in I} \\ [\![\mathsf{End}]\!]_G &\triangleq \mathsf{End} \end{aligned}$$

After redirecting, the projections have the right local types for B and the dual $\overline{B}$ (flips all ? with ! and vice versa):

Lemma 4.1. $[\![B]\!]_G \downharpoonright 0 = \pi^{-1}([\![B]\!]_L)$ and $[\![B]\!]_G \downharpoonright 1 = [\![\overline{B}]\!]_L$

Using this lemma and translation of types, we can prove that the binary typing rules for $\mathbf{fork}_B$, $\mathbf{send}_B$, $\mathbf{receive}_B$ and $\mathbf{close}_B$ are derivable (Figure 5).

This section shows that MPGV supports the full power of GV-style binary session types, including treatment of sessions as first-class data and dynamic spawning of sessions. Note that redirecting is crucial: without it we are not able to do a type-preserving translation, because local types $\mathbf{!}[0]$ and $\mathbf{?}[0]$ are incompatible with $\mathbf{!}[1]$ and $\mathbf{?}[1]$.

5 THE DEADLOCK AND LEAK FREEDOM THEOREM

MPGV guarantees strong properties for well-typed programs, while supporting dynamic spawning, session interleaving, and first-class endpoints. These properties are:

Type safety: The only way for a thread to get stuck is by blocking to receive from an empty buffer.
Session fidelity: The values sent to and received from buffers match the types in the protocol.
Global progress: Configurations of a well-typed initial program are either final or can take a step.
Deadlock freedom: No subset of the threads get stuck by waiting for each other.
Memory leak freedom: All data always remains reachable.

Ideally, we would like to capture these properties in a single theorem that subsumes them all. As a first step, we formulate global progress as follows:

Theorem 5.1 (Global progress). If $\emptyset \vdash e : \mathbf{1}$, and $([e], \emptyset) \rightsquigarrow^*_{\text{cfg}} (\vec{e}, h)$, then:

(1) there exists $(\vec{e}', h')$ such that $(\vec{e}, h) \rightsquigarrow_{\text{cfg}} (\vec{e}', h')$, or
(2) $\vec{e}_i = ()$ for all $i \in \text{dom}(\vec{e})$ and $h = \emptyset$.

This theorem rules out whole-program deadlocks and ensures that all buffers have been correctly deallocated when the program finishes. However, this theorem does not guarantee anything as long as there is still a single thread that can step. Thus it does not guarantee local deadlock freedom, nor memory leak freedom while the program is still running. Moreover, it does not even guarantee type safety: a situation in which a thread is stuck on a type error is not ruled out by this theorem as long as there is another thread that can still step. We therefore state partial deadlock freedom and memory leak freedom theorems, but we strengthen both so that they become equivalent. We use the definitions of partial deadlock and memory leak freedom of Jacobs et al. [2022a] and apply them to MPGV. We need the following notions:

- The set $v \in V ::= \mathsf{Thread}(i) \mid \mathsf{Session}(s)$ ranging over possible threads and sessions.
- The function $\mathsf{refs}_{(\vec{e},h)}(v) \subseteq V$ giving the set of sessions that v references.
- The predicate $\mathsf{blocked}_{(\vec{e},h)}(v_1, v_2)$ stating that thread $v_1 = \mathsf{Thread}(i)$ is blocked on session $v_2 = \mathsf{Session}(s)$.
- The function $\mathsf{active}(\vec{e}, h) \subseteq V$ giving the set of active threads and sessions in the configuration.

Using these notions, we strengthen partial deadlock freedom to incorporate aspects of memory leak freedom.

Definition 5.2 (Partial deadlock/leak). Given a configuration $(\vec{e}, h)$, a subset $S \subseteq V$ of the threads and sessions is in a partial *deadlock/leak* if the following conditions hold:

(1) We have $\emptyset \subset S \subseteq \mathsf{active}(\vec{e}, h)$.
(2) For all threads $\mathsf{Thread}(i) \in S$, the expression e_i cannot step in the heap h.
(3) If $\mathsf{Thread}(i) \in S$ and $\mathsf{blocked}_{(\vec{e},h)}(\mathsf{Thread}(i), \mathsf{Session}(s))$, then $\mathsf{Session}(s) \in S$.
(4) If $\mathsf{Session}(s) \in S$ and $\mathsf{Session}(s) \in \mathsf{refs}_{(\vec{e},h)}(v)$, then $v \in S$.

Definition 5.3 (Partial deadlock/leak freedom). A configuration $(\vec{e}, h)$ is *deadlock/leak free* if no $S \subseteq V$ is in a partial deadlock/leak in $(\vec{e}, h)$.

Conversely, we strengthen memory leak freedom (*i.e.*, full reachability) to incorporate aspects of deadlock freedom.

Definition 5.4 (Reachability). We inductively define the threads and sessions *reachable* in $(\vec{e}, h)$:

(1) $\mathsf{Thread}(i)$ is reachable if either
    - the expression e_i can step in the heap h, or
    - there exists an s such that $\mathsf{Session}(s)$ is reachable and $\mathsf{blocked}_{(\vec{e},h)}(\mathsf{Thread}(i), \mathsf{Session}(s))$.
(2) $\mathsf{Session}(s)$ is reachable if there exists a reachable v such that $\mathsf{Session}(s) \in \mathsf{refs}_{(\vec{e},h)}(v)$.

Definition 5.5 (Full reachability). A configuration $(\vec{e}, h)$ is *fully reachable* if all $v \in \mathsf{active}(\vec{e}, h)$ are reachable in $(\vec{e}, h)$.

As in Jacobs et al. [2022a]'s language for binary sessions, the strengthened versions of deadlock freedom and full reachability are equivalent, and well-typed MPGV programs satisfy both properties:

Theorem 5.6. A configuration $(\vec{e}, h)$ is deadlock/leak free if and only if it is fully reachable.

Theorem 5.7. If $\emptyset \vdash e : \mathbf{1}$ and $([e], \emptyset) \rightsquigarrow_{\text{cfg}} (\vec{e}, h)$, then $(\vec{e}, h)$ is fully reachable and deadlock/leak free.

The final theorem encompasses type safety, session fidelity, deadlock freedom, and memory leak freedom. Global progress (Theorem 5.1) also follows as a corollary from the final theorem.

6 EXTENSION: CONSISTENCY WITHOUT GLOBAL TYPES

Inspired by Scalas and Yoshida [2019a,b], we define a notion of consistency that does not rely on global types. This notion of consistency plays an important role in our proof of deadlock freedom (§7), but is also more flexible. It is more flexible in the sense that $\mathsf{consistent}(L_0, \dots, L_n)$ (premise of **fork** in Figure 3) may hold even if no global type exists whose projections are $L_1, \dots, L_n$. For example, there exists no global type for the local types $L_0 = \mathbf{!}[1]\mathbf{N}.\mathbf{?}[1]\mathbf{N}.\mathsf{End}$ and $L_1 = \mathbf{!}[0]\mathbf{N}.\mathbf{?}[0]\mathbf{N}.\mathsf{End}$ because they both start with a send. Nevertheless, it would be safe and deadlock free to allow this protocol, given an asynchronous semantics.[1] The more flexible notion of consistency we define in §6.1 does allow this protocol. In §6.2 we show that the existence of a global type for local types implies our flexible notion of consistency.

6.1 Defining Consistency without Global Types

At a high level, we define $\mathsf{consistent}(L_0, \dots, L_n)$ as follows:

> "The local types $L_0, \dots, L_n$ of a session are consistent if no deadlock can occur *within* the session when considering all possible interleavings of participant actions, assuming that each participant p follows its respective local type L_p."

Our goal is to define this notion solely as a property of the local types $L_0, \dots, L_n$, so that consistency of a session's local types can be proven without considering other sessions. To do so, we define the notion of *shadow buffers*:

$$\hat{Q} \in \mathit{ShadowBuf} \triangleq \mathit{Participant} \xrightarrow{\text{fin}} (\mathit{Participant} \xrightarrow{\text{fin}} \mathit{List}(\mathit{Label} \times \mathit{Type}))$$

Shadow buffers are similar to the physical buffers in the heap, but there are two differences. First, whereas the physical buffers contain pairs $\langle \ell : v \rangle$ of labels and values, shadow buffers contain pairs $\langle \ell : \tau \rangle$ of labels and types. Second, whereas the heap concerns all sessions, shadow buffers only concern a single session. Hence, the heap ranges over endpoints (recall that $\mathit{Endpoint} \triangleq \mathit{Session} \times \mathit{Participant}$), but shadow buffers range over mere participants.

Shadow buffers allow us to simulate the local execution of a session on the abstract level. If all the possible local executions allowed by a set of local types $\mathbf{L} : \mathit{Participant} \xrightarrow{\text{fin}} \mathit{LType}$ on a set of shadow buffers $\hat{Q}$ are type safe and deadlock free, we say that $\hat{Q}$ is consistent with $\mathbf{L}$, which we denote by $\mathsf{consistent}(\hat{Q}, \mathbf{L})$, and define as follows:

Definition 6.1. The judgment $\mathsf{consistent}(\hat{Q}, \mathbf{L})$ is defined as the most permissive relation satisfying the following properties:

(1) Consistency is preserved by sends, *i.e.,* for every participant p with $\mathbf{L}(p) = \mathbf{!}[q]\{\ell : \tau_\ell .\ L_\ell\}_{\ell \in I}$, then $\mathsf{consistent}(\mathsf{push}(q, p, \langle \ell : \tau_\ell \rangle, \hat{Q}), \mathbf{L}[p := L_\ell])$.
(2) Consistency is preserved by receives, *i.e.,* for every participant q with $\mathbf{L}(q) = \mathbf{?}[p]\{\ell : \tau_\ell .\ L_\ell\}_{\ell \in I}$, and $\mathsf{pop}(q, p, \hat{Q}) = (\langle \ell : \tau \rangle, \hat{Q}')$, then $\ell \in I$, and $\mathsf{consistent}(\hat{Q}', \mathbf{L}[q := L_\ell])$, and $\tau = \tau_\ell$.
(3) Consistency is preserved by channel closure, *i.e.,* for every participant p with $\mathbf{L}(p) = \mathsf{End}$, then $\mathsf{consistent}(\hat{Q} \setminus \{p\}, \mathbf{L} \setminus \{p\})$.
(4) Either all buffers have been deallocated ($\hat{Q} = \emptyset$) or there is a participant q such that q's local type $\mathbf{L}(q)$ is a send or a close, or $\mathbf{L}(q)$ is a receive and the corresponding buffer contains a value, *i.e.,* $\mathsf{pop}(q, p, \hat{Q}) = (\langle \ell : \tau \rangle, \hat{Q}')$ for some label ℓ, type τ, and new set of shadow buffers $\hat{Q}'$.
(5) For each participant there is a corresponding set of buffers and vice versa, *i.e.,* $\mathsf{dom}(\mathbf{L}) = \mathsf{dom}(\hat{Q})$.

Note that the cases for the preservation of $\mathsf{consistent}(\hat{Q}, \mathbf{L})$ under the sends, receives, and channel closure refer to a recursive occurrence $\mathsf{consistent}(\hat{Q}', \mathbf{L}')$ for some $\hat{Q}'$ and $\mathbf{L}'$. Since we consider

[1]There exist other extensions of (multiparty) session types that allow for a more flexible notion of consistency. In particular, session-type systems with *asynchronous subtyping* also support this example [Ghilezan et al. 2021; Mostrous et al. 2009].

the most permissive relation, these recursive occurrences should be interpreted coinductively—we use Coq's `CoInductive` keyword in the mechanization.

The first three properties are used to show that the channel operations are type safe and the resulting state is again consistent. The fourth property is used to show deadlock freedom. The fifth property is required for technical reasons because we support the possibility of some participants deallocating their buffers while other participants are continuing to communicate with each other. With this at hand, we define the new consistency predicate used in the **fork** typing rule:

Definition 6.2. We define $\mathsf{consistent}(\vec{L})$ as $\mathsf{consistent}(\mathsf{init}(\mathsf{length}(\vec{L})), \vec{L})$, where $\mathsf{init}(n)$ creates n empty buffers, and the list $\vec{L}$ is converted into a map in the natural way.

Note that we need to use the finite map representation because some participants can close their channel before others (see Item 3 in Definition 6.1), and then they disappear from $\mathbf{L}$ (lists do not allow gaps in the middle, whereas finite maps do).

6.2 Global Types Imply Consistency

The goal of this section is to show that if there is a global type for a set of local types, then the local types are consistent in the sense of the preceding section:

Theorem 6.3. *If there is a global type G with $n+1$ participants such that $G \downharpoonright 0 = L_0, \ldots, G \downharpoonright n = L_n$, then $\mathsf{consistent}(L_0, \ldots, L_n)$.*

This lemma shows that we did not lose anything by using the more flexible notion of consistency without global types—the programs we are able to type check with the more flexible notion of consistency are a superset of the programs we are able to type check using global types.

We cannot prove Theorem 6.3 directly using coinduction, because the coinductive conclusion is not general enough. We need a more general property that involves the consistency judgment $\mathsf{consistent}(\hat{Q}, \mathbf{L})$ for an arbitrary set of shadow buffers $\hat{Q}$. Our generalized property (Lemma 6.4) makes use of the notion *runtime global types*, inspired by the work of Castro-Perez et al. [2021].

Runtime global types. To model the state of a global type during an interaction in which some messages have already been sent but not yet received, we define runtime global types as:

$$R \in RType \underset{(ind)}{::=} [p_1 \xrightarrow{\ell?} p_2]\{\ell : \tau_\ell.\ R_\ell\}_{\ell \in I} \mid \mathsf{Cont}\ G$$

Runtime global types differ from ordinary global types (§3.2.3) in three aspects:

(1) Operations in runtime global types have an optional label ℓ on the arrow. If no label is present (*i.e.*, $[p_1 \to p_2]\{\ell : \tau_\ell.\ R_\ell\}_{\ell \in I}$), then both the send and receive remain to happen. If a label ℓ is present (*i.e.*, $[p_1 \xrightarrow{\ell} p_2]\{\ell : \tau_\ell.\ R_\ell\}_{\ell \in I}$), then the send portion (with label ℓ) of the operation has already happened, but the receive is still pending.
(2) Runtime global types are defined *inductively* rather than coinductively, because only finitely many messages have been sent at any given point in time.
(3) Instead of having End, they have $\mathsf{Cont}\ G$, indicating that the protocol continues as ordinary global type G.

Runtime local type projections. The projections $R \downharpoonright p = L$ of runtime global types onto local types can be found in Figure 6. These rules are inductively defined. Intuitively, when an operation $[p \xrightarrow{\ell} q]\{\ell : \tau_\ell.\ R_\ell\}_{\ell \in I} \downharpoonright r$ occurs in the runtime global type, then the projection onto p ignores the operation and continues with R_ℓ because the send by p with label ℓ has already happened. However, the projection onto q in this case still has to take the receive part of this operation into

$$\frac{q \neq r \qquad \forall \ell \in I.\, R_\ell \downharpoonleft r = L_\ell}{[r \to q]\{\ell : \tau_\ell.\, R_\ell\} \downharpoonleft r = \mathbf{!}[q]\{\ell : \tau_\ell.\, L_\ell\}} \qquad \frac{p \neq r \qquad \forall \ell \in I.\, R_\ell \downharpoonleft r = L_\ell}{[p \xrightarrow{\ell ?} r]\{\ell : \tau_\ell.\, R_\ell\} \downharpoonleft r = \mathbf{?}[p]\{\ell : \tau_\ell.\, L_\ell\}}$$

$$\frac{r \notin \{p, q\} \qquad \forall \ell \in I.\, R_\ell \downharpoonleft r = L \qquad I \neq \emptyset}{[p \to q]\{\ell : \tau_\ell.\, R_\ell\} \downharpoonleft r = L} \qquad \frac{q \neq r \qquad R_\ell \downharpoonleft r = L}{[p \xrightarrow{\ell} q]\{\ell : \tau_\ell.\, R_\ell\} \downharpoonleft r = L} \qquad \frac{G \downharpoonleft r = L}{\mathsf{Cont}\ G \downharpoonleft r = L}$$

$$\frac{\mathsf{pop}(q, p, \hat{Q}) = \bot \quad \forall \ell.\, R_\ell \downharpoonleft\!\!\downharpoonleft \hat{Q}}{[p \to q]\{\ell : \tau_\ell.\, R_\ell\} \downharpoonleft\!\!\downharpoonleft \hat{Q}} \qquad \frac{\mathsf{pop}(q, p, \hat{Q}) = (\langle \ell : \tau_\ell \rangle, \hat{Q}') \quad R_\ell \downharpoonleft\!\!\downharpoonleft \hat{Q}'}{[p \xrightarrow{\ell} q]\{\ell : \tau_\ell.\, R_\ell\} \downharpoonleft\!\!\downharpoonleft \hat{Q}} \qquad \frac{\hat{Q} = \emptyset}{\mathsf{Cont}\ G \downharpoonleft\!\!\downharpoonleft \hat{Q}}$$

Fig. 6. Projections of runtime global types: (1) local type projections $R \downharpoonleft p = L$, and (2) shadow buffer projections $R \downharpoonleft\!\!\downharpoonleft \hat{Q}$ (inductive).

account, because the receive has not happened yet. The other cases are similar to the projections for ordinary global types (Figure 3), and ensure that the protocol remains well-formed.

Runtime buffer projections. We also define the judgment $R \downharpoonleft\!\!\downharpoonleft \hat{Q}$, which says that the messages in the runtime global type R correspond to the shadow buffers $\hat{Q}$.

Runtime global types imply consistency. Using the notion of runtime global type and runtime projections, we are able to show the following lemma:

Lemma 6.4. The judgment consistent$(\hat{Q}, \mathbf{L})$ holds if there exists a runtime global type R for which the following four conditions hold: (1) $R \downharpoonleft\!\!\downharpoonleft \hat{Q}$ (2) $\forall p.\, R \downharpoonleft p = L(p)$ (3) participants$(R) \subseteq$ dom$(\mathbf{L})$ (4) $\forall p.$ if $\hat{Q}(p) = \bot$ then $p \notin$ dom$(\mathbf{L})$ else dom$(\mathbf{L}) \subseteq$ dom$(\hat{Q}(p))$.

The lemma is proved using coinduction, and relies on a series of auxiliary lemmas (see the Coq development for details [Jacobs et al. 2022b]). Once we have this lemma, Theorem 6.3 follows by relating projection of runtime global types to projection of ordinary global types.

7 PROOF OF DEADLOCK AND LEAK FREEDOM

We give an overview of the proof of our main result, Theorem 5.7. The proof is quite technical, but since all parts have been mechanized in Coq [Jacobs et al. 2022b], one can trust the theorems independent of the pen-and-paper description of the proof. We hope to provide enough insights into the proof to make our results reproducible and extensible.

The high level structure of the proof is as follows:

- We define an *invariant* on the runtime configuration, which states (1) that everything in the configuration is well-typed and that the buffer contents are consistent with respect to the local types of each channel endpoint, and (2) that the topology of the configuration is acyclic.
- We prove that the invariant is preserved by steps of the operational semantics ("preservation").
- We prove that configurations that satisfy the invariant cannot be in a deadlock ("progress").

To deal with linearity and acyclicity we use the connectivity graph framework of Jacobs et al. [2022a], which provides a couple of features to make our proof feasible. First, it provides a generic construction to define the invariant—it allows us to provide local invariants for threads and channels, which the framework then lifts to an invariant for whole runtime configurations. Second, it makes use of separation logic to hide reasoning about linearity. Third, it provides generic reasoning principles to prove the preservation (of acyclicity and typing) and progress parts of the proof. Fourth, it is implemented as a library in Coq, so it allows us to mechanize our proofs.

$$
\begin{array}{rlrl}
P, Q \in sProp & \triangleq (V \xrightarrow{\text{fin}} E) \to Prop & V & ::= \mathsf{Thread}(i) \mid \mathsf{Session}(s) \\
(\mathsf{Emp})(\Sigma) & \triangleq (\Sigma = \emptyset) & E & \triangleq Participant \times LType \\
(\mathsf{False})(\Sigma) & \triangleq \mathsf{False} & \Sigma & \in V \xrightarrow{\text{fin}} E \\
(\mathsf{True})(\Sigma) & \triangleq \mathsf{True} & (P \vee Q)(\Sigma) & \triangleq P(\Sigma) \vee Q(\Sigma) \\
(\ulcorner \phi \urcorner)(\Sigma) & \triangleq \phi \wedge (\Sigma = \emptyset) & (P \wedge Q)(\Sigma) & \triangleq P(\Sigma) \wedge Q(\Sigma) \\
(\mathsf{own}(\Sigma'))(\Sigma) & \triangleq (\Sigma = \Sigma') & (\exists x.\, P(x))(\Sigma) & \triangleq \exists x.\, P(x)(\Sigma) \\
(\Box P)(\Sigma) & \triangleq P(\emptyset) \wedge \Sigma = \emptyset & (\forall x.\, P(x))(\Sigma) & \triangleq \forall x.\, P(x)(\Sigma)
\end{array}
$$

$$(P * Q)(\Sigma) \triangleq \exists \Sigma_1 \Sigma_2.\, \mathsf{dom}(\Sigma_1) \cap \mathsf{dom}(\Sigma_2) = \emptyset \wedge \Sigma = \Sigma_1 \uplus \Sigma_2 \wedge P(\Sigma_1) \wedge Q(\Sigma_2)$$

$$(P \mathrel{-\!\!*} Q)(\Sigma) \triangleq \forall \Sigma'.\, \big(\mathsf{dom}(\Sigma) \cap \mathsf{dom}(\Sigma') = \emptyset \wedge P(\Sigma')\big) \Rightarrow Q(\Sigma \uplus \Sigma')$$

Fig. 7. The definition of the separation logic connectives.

At the high-level, the structure of our proof and our use of the connectivity framework is similar to Jacobs et al. [2022a]'s proof for binary session types. To use the framework to obtain the invariant for configurations (§7.3), we first define a *runtime type system* for expressions to express the local invariant for threads (§7.1), and define a local invariant for the buffers that back a session (§7.2). The new element of our proof is handling multiparty instead of binary sessions, for which we make use of our notion of shadow buffers (§6).

With the invariant for configurations at hand, we prove that this invariant holds for the initial configurations and is preserved by the operational semantics (§7.4). The new element is an extension of the connectivity graph framework to handle n-ary graph transformations to support the multiparty case. To complete the proof, we show that the configuration invariant implies Theorem 5.7, our main deadlock freedom theorem (§7.5).

7.1 Runtime Type System

The first step to define the invariant for configurations is to define a runtime typing judgment for expressions. The runtime judgment differs from the static typing judgment (§3.2) in the sense that it should account for channel literals $\#[c, \pi]$ that appear after the execution of a **fork**. Traditionally, this is done by extending the typing judgment $\Sigma; \Gamma \vdash e : \tau$ with an additional context Σ that keeps track of the types of the channel literals (often called a *heap typing*).[2] To avoid having to thread through such this context everywhere, and having to deal with splitting conditions of this context (due to linearity), we make use of separation logic [O'Hearn and Pym 1999; O'Hearn et al. 2001]. This follows the approach in the connectivity graph framework [Jacobs et al. 2022a], which in turn is based on Rouvoet et al. [2020]'s use of separation logic to hide heap typings in intrinsically-typed interpreters for linear languages in Agda.

Our runtime judgment $\Gamma \vDash e : \tau$ is formalized as a separation logic proposition *sProp*, *i.e.,* a predicate over heap typings Σ. The semantics of the separation logic connectives can be found in Figure 7 and the rules of our runtime type system in Figure 8. Crucially, the use of separating conjunction in the rules of n-ary constructs hides the splitting of the heap typing Σ, and the use of $\mathsf{own}(s \mapsto (p, \pi(L)))$ in the rule for endpoint literals $\#[(s, p), \pi]$ makes sure the type of each literal matches up with the heap typing Σ. Note that the runtime judgment $\Gamma \vDash e : \tau$ is defined recursively on the structure of e. To assert that $P \in sProp$ is true, means to assert that $P(\emptyset)$ holds.

[2]The actual type of Σ in Figure 7 also accounts for threads in addition to sessions. This is due to the use of the connectivity graph framework, which we discuss in §7.3.

$$\dfrac{\ulcorner \Gamma\ \mathsf{unr} \urcorner \quad * \quad \mathsf{own}(s \mapsto (p, \pi(L)))}{\Gamma \vDash \#[(s,p),\pi] : L}{*} \qquad \dfrac{\Box(\Gamma \cup \{x \mapsto \tau_1\} \vDash e : \tau_2) \quad * \quad \ulcorner \Gamma\ \mathsf{unr} \urcorner \quad * \quad \ulcorner x \notin \Gamma \urcorner}{\Gamma \vDash \lambda x.\ e : \tau_1 \to \tau_2}{*}$$

$$\dfrac{\ulcorner \Gamma_1 \perp \cdots \perp \Gamma_n \urcorner \quad * \quad \ulcorner \mathsf{consistent}(L_0, L_1, \ldots, L_n) \urcorner \quad * \quad [*]\, p \in \{1..n\}.\ \Gamma_p \vDash e_p : L_p \multimap \mathbf{1}}{\Gamma_1 \cup \cdots \cup \Gamma_n \vDash \mathbf{fork}(e_1, \ldots, e_n) : L_0}{*}$$

$$\dfrac{\Gamma \vDash e : \mathsf{End}}{\Gamma \vDash \mathbf{close}(e) : \mathbf{1}}{*} \qquad \dfrac{\ulcorner \Gamma_1 \perp \Gamma_2 \urcorner \quad * \quad \Gamma_1 \vDash e_1 : \mathbf{!}[p]\{\ell : \tau_\ell.\ L_\ell\}_{\ell \in I} \quad * \quad \Gamma_2 \vDash e_2 : \tau_\ell}{\Gamma_1 \cup \Gamma_2 \vDash \mathbf{send}[p](e_1, \ell : e_2) : L_\ell}{*}$$

$$\dfrac{\Gamma \vDash e : \mathbf{?}[p]\{\ell : \tau_\ell.\ L_\ell\}_{\ell \in I}}{\Gamma \vDash \mathbf{receive}[p](e) : \Sigma_{\ell \in I}.\ \tau_\ell \times L_\ell}{*} \qquad \dfrac{\Gamma \vDash e : \pi(L)}{\Gamma \vDash \mathbf{redirect}[\pi](e) : L}{*}$$

Fig. 8. Selected separation logic runtime typing rules (recursive).

$$\begin{aligned}
\mathsf{wf}(\vec{e}, h) &\triangleq \mathsf{wf}(\mathsf{wf}^{local}_{(\vec{e},h)}) \\
\mathsf{wf}(P) &\triangleq \exists G : \mathit{Cgraph}(V, E).\ \forall v \in V.\ P(v, \mathsf{in}(G, v))(\mathsf{out}(G, v)) \\
\mathsf{wf}^{local}_{(\vec{e},h)}(v, \Delta) &\triangleq \begin{cases} \ulcorner \Delta = \emptyset \urcorner * (\emptyset \vDash e_i : \mathbf{1}) & \text{if } v = \mathsf{Thread}(i), i < |\vec{e}| \\ \ulcorner \Delta = \emptyset \urcorner & \text{if } v = \mathsf{Thread}(i), i \geq |\vec{e}| \\ \exists \mathbf{L} \in \mathit{Participant} \xrightarrow{\text{fin}} \mathit{LType}. & \text{if } v = \mathsf{Session}(s) \\ \qquad \ulcorner \Delta = \mathsf{toMultiset}(\mathbf{L}) \urcorner * \mathsf{consistent}(h|_s, \mathbf{L}) & \end{cases} \\
\mathsf{consistent}(Q, \mathbf{L}) &\triangleq \exists \hat{Q}.\ \ulcorner \mathsf{consistent}(\hat{Q}, \mathbf{L}) \urcorner * Q \propto \hat{Q} \\
Q \propto \hat{Q} &\triangleq\ [*]\, Q_p; \hat{Q}_p \in Q; \hat{Q}.\ [*]\, Q_{pq}; \hat{Q}_{pq} \in Q_p; \hat{Q}_p. \\
&\qquad [*]\, \langle \ell_1 : v \rangle ; \langle \ell_2 : \tau \rangle \in Q_{pq}; \hat{Q}_{pq}.\ \ulcorner \ell_1 = \ell_2 \urcorner * (\emptyset \vDash v : \tau)
\end{aligned}$$

Fig. 9. Configuration invariant.

To prove the initialization lemma (Lemma 7.4), we state in separation logic that statically well-typed expressions are well-typed in the runtime type system:

Lemma 7.1. $\ulcorner \Gamma \vdash e : \tau \urcorner \mathrel{-\!\!*} \Gamma \vDash e : \tau$

7.2 The Buffer Invariant

We now define an invariant $\mathsf{consistent}(Q, \mathbf{L})$ to express that the buffers Q for a given session s are consistent with respect to a set of local types $\mathbf{L} : \mathit{Participant} \xrightarrow{\text{fin}} \mathit{LType}$. The buffer invariant is similar to the consistency judgment $\mathsf{consistent}(\hat{Q}, \mathbf{L})$ we defined in §6.1, but it operates on physical buffers Q (*i.e.,* buffers with values) instead of shadow buffers $\hat{Q}$ (*i.e.,* buffers with types):

$$Q \in \mathit{Buf} \triangleq \mathit{Participant} \xrightarrow{\text{fin}} (\mathit{Participant} \xrightarrow{\text{fin}} \mathit{List}(\mathit{Label} \times \mathit{Val}))$$

(We use the notation $h|_s$ to obtain the buffers for a session s from the heap h.)

Since MPGV allows to send arbitrary data over channels, the values in buffers can themselves contain channel literals. Hence, similar to the runtime typing judgment, the buffer invariant needs to be indexed by a heap typing Σ, which we hide again by considering $\mathsf{consistent}(Q, \mathbf{L})$ to be a separation logic proposition *sProp*. The definition of $\mathsf{consistent}(Q, \mathbf{L}) \in \mathit{sProp}$ can be found in Figure 9. This definition contains two key ingredients. First, it makes use of $\mathsf{consistent}(\hat{Q}, \mathbf{L}) \in \mathit{Prop}$ from §6 to specify that local types $\mathbf{L}$ are consistent with some (existentially quantified) shadow buffers $\hat{Q}$. Second, it makes use of the auxiliary definition $Q \propto \hat{Q} \in \mathit{sProp}$ in Figure 9 to specify that the labels in the physical buffers Q are equal to those in the shadow buffers $\hat{Q}$, and that the values in the physical buffers Q have types determined by the corresponding entry in the shadow buffers $\hat{Q}$. (The notation $[*]\ x; y \in X; Y.\ P(x, y)$ in Figure 9 is an n-ary separating conjunction: it states that the collections X, Y (lists or finite maps) have the same domain, and gives $P(X_0, Y_0) * \cdots * P(X_n, Y_n)$, where (X_i, Y_i) are corresponding elements in the collections.)

The invariant $\mathsf{consistent}(Q, \mathbf{L})$ for physical buffers has preservation and initialization properties paralleling to the rules of the consistency relation $\mathsf{consistent}(\hat{Q}, \mathbf{L})$ for shadow buffers (Definitions 6.1 and 6.2). Since $\mathsf{consistent}(Q, \mathbf{L})$ is a separation logic proposition, these properties are stated using the separation logic connectives (and thus implicitly describe the threading and splitting of the heap typing Σ).

Lemma 7.2. The buffer invariant is preserved by a sends, receives, and channel closure:

- If $\mathbf{L}(p) = ![q]\{\ell : \tau_\ell.\ L_\ell\}_{\ell \in I}$, then:
  $(\emptyset \vDash v : \tau_\ell) * \mathsf{consistent}(Q, \mathbf{L}) \twoheadrightarrow \mathsf{consistent}(\mathsf{push}(q, p, \langle \ell : \tau_\ell \rangle, Q), \mathbf{L}[p := L_\ell])$.
- If $\mathbf{L}(q) = ?[p]\{\ell : \tau_\ell.\ L_\ell\}_{\ell \in I}$, and $\mathsf{pop}(q, p, \hat{Q}) = (\langle \ell : \tau \rangle, \hat{Q}')$, then
  $\mathsf{consistent}(Q, \mathbf{L}) \twoheadrightarrow \ulcorner \ell \in I \urcorner * \mathsf{consistent}(Q', \mathbf{L}[q := L_\ell]) * (\emptyset \vDash v : \tau_\ell)$.
- If $\mathbf{L}(p) = \mathsf{End}$, then $\mathsf{consistent}(Q, \mathbf{L}) \twoheadrightarrow \mathsf{consistent}(Q \backslash \{p\}, \mathbf{L} \backslash \{p\})$.

Lemma 7.3. If $\mathsf{consistent}(\vec{L})$, then $\mathsf{Emp} \twoheadrightarrow \mathsf{consistent}(\mathsf{init}(\mathsf{length}(\vec{L})), \vec{L})$.

7.3 The Configuration Invariant

The invariant $\mathsf{wf}(\vec{e}, h)$ for configurations $(\vec{e}, h)$ ensures that every thread in $\vec{e}$ is well-typed, the contents of the buffers $h|_s$ for each session s in h are well-typed, the types of the channel literals match up with the types of the channels, and the communication topology is acyclic. To define this invariant, we instantiate the connectivity graph framework of Jacobs et al. [2022a] with the runtime typing judgment from §7.1 and the buffer invariant from §7.2.

The first ingredient of the connectivity framework is the data type $\mathit{Cgraph}(V, E)$, which represents a directed graph with vertices ranging over the set V and edge labels ranging over the set E. This graph should be acyclic in an undirected sense (*i.e.,* the undirected erasure of the graph forms an undirected unrooted forest). We instantiate V and E in $\mathit{Cgraph}(V, E)$ as follows:

$$V ::= \mathsf{Thread}(i) \mid \mathsf{Session}(s) \qquad\qquad E \triangleq \mathit{Participant} \times \mathit{LType}$$

The second ingredient of the connectivity graph framework is a generic invariant $\mathsf{wf}(P)$, which lifts a local invariant predicate $P(v, \Delta) \in \mathit{sProp}$ to whole runtime configurations. The local predicate P links the local configuration state of each vertex v (*i.e.,* the expression for a thread and the buffers for a session) to the multiset Δ of labels on the incoming edges of vertex v. Our instantiation $P(v, \Delta) \triangleq \mathsf{wf}^{\mathit{local}}_{(\vec{e},h)}(v, \Delta)$ is given in Figure 9. Intuitively, the local invariant for a thread (case $v = \mathsf{Thread}(i)$) says that the expression e_i of that thread is well-typed in the runtime type system with respect to the local types on the outgoing edges of the thread's vertex in the connectivity graph. The local invariant for a session (case $v = \mathsf{Session}(s)$) says that the buffers $h|_s$ of that session are well-typed with respect to the local types on the incoming edges of the session's vertex in the connectivity graph, where the endpoints stored in the buffers get their local types from the outgoing edges. The invariant for the whole configuration $\mathsf{wf}(\vec{e}, h)$ says that there exists an acyclic graph $G : \mathit{Cgraph}(V, E)$ such that the local invariant predicate holds for all $v \in V$.

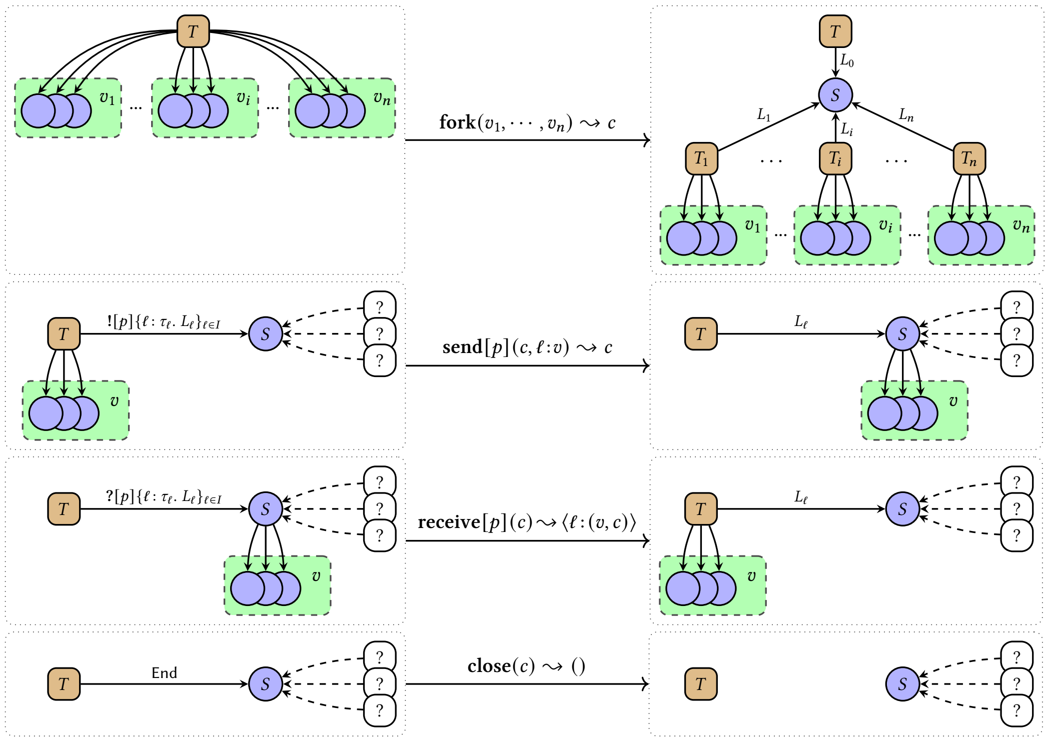
T
v_1
v_i
v_n
$\mathbf{fork}(v_1, \cdots, v_n) \rightsquigarrow c$
L_0
S
L_1
L_i
L_n
T_1
T_i
T_n
$![p]\{\ell : \tau_\ell .\ L_\ell\}_{\ell \in I}$
$\mathbf{send}[p](c, \ell : v) \rightsquigarrow c$
L_ℓ
?
v
$?[p]\{\ell : \tau_\ell .\ L_\ell\}_{\ell \in I}$
$\mathbf{receive}[p](c) \rightsquigarrow \langle \ell : (v, c) \rangle$
End
$\mathbf{close}(c) \rightsquigarrow ()$

Fig. 10. Graphical depiction of how multiparty interactions change the logical connectivity. Blue circles are multiparty sessions, brown squares are threads. A blue circle abstracts over the $n \times n$ communication paths among the n session participants, where each endpoint has buffers for incoming messages from every other endpoint. An edge from T to S indicates that thread T has an endpoint of session S. An edge from a session S_1 to a session S_2 indicates that an endpoint of S_2 is stored in one of the buffers of S_1. The figure provides a *local* viewpoint, only depicting the notions directly involved in an interaction and omitting other threads and sessions that are connected to the depicted ones as well. While the communication topology is cyclic within a multiparty session (where the global types rule out deadlock), it is acyclic *between* multiparty sessions, an invariant preserved by multiparty interactions. Acyclicity is crucial for deadlock and memory leak freedom.

7.4 Initialization and Preservation of the Invariant

The invariant holds for initial configurations and is preserved by the operational semantics:

Lemma 7.4. If $\emptyset \vdash e : \mathbf{1}$, then $\mathsf{wf}([e], \emptyset)$.

Lemma 7.5. If $(\vec{e}, h) \rightsquigarrow_{\mathrm{cfg}} (\vec{e}', h')$, then $\mathsf{wf}(\vec{e}, h)$ implies $\mathsf{wf}(\vec{e}', h')$.

The proof of the last lemma involves three aspects. First, because the configuration changes, we need to produce a connectivity graph for the new configuration as the connectivity graph is existentially quantified in the configuration invariant $\mathsf{wf}(\vec{e}, h)$. Second, we need to show that the new connectivity graph is acyclic in the appropriate sense. Third, we need to show that all the local invariant predicates $\mathsf{wf}^{local}_{(\vec{e},h)}(v, \Delta)$ still hold. The interesting cases of this proof are the steps that involve the channel operations, for which the graph transformations are depicted in Figure 10.

Proving these graph transformations by picking a new graph by hand is cumbersome (especially in a mechanized proof). The connectivity graph framework therefore provides abstract separation logic lemmas to prove the transformations without having to mention the graph or having to deal with its acyclicity explicitly. We can re-use some of these abstract transformation rules, but for the n-ary **fork** we need a new rule (which we state abstractly for arbitrary vertices V and edge labels E).

Lemma 7.6. Let $v_1, v_2 \in V$ and $w_1, \ldots, w_n \in V$. To prove $\mathsf{wf}(P)$ implies $\mathsf{wf}(P')$, it suffices to prove, for all $\Delta \in \mathit{Multiset}\ E$:

(1) $P(v, \Delta) \twoheadrightarrow P'(v, \Delta)$ for all $v \in V \setminus \{v_1, v_2, w_1, \ldots, w_n\}$
(2) $P(v, \Delta) \twoheadrightarrow \ulcorner \Delta = \emptyset \urcorner$ for all $v \in \{v_2, w_1, \ldots, w_n\}$
(3) $P(v_1, \Delta) \twoheadrightarrow \exists\, l_0, \ldots, l_n.\ (\mathsf{own}(v_2 \mapsto l_0) \twoheadrightarrow P'(v_1, \Delta)) *$
$P'(v_2, \{l_0, \ldots, l_n\}) *$
$([*] i \in \{1..n\}.\ \mathsf{own}(v_2 \mapsto l_i) \twoheadrightarrow P'(v_i, \emptyset))$

7.5 Proof of the Reachability Theorem

We give an intuitive description of the proof of our main reachability theorem (Theorem 5.7).

Waiting induction. At the top level of the proof, we apply the waiting induction principle of the connectivity graph library. Waiting induction relies on acyclicity of the graph and allows one to prove a predicate $P(v)$ for all vertices $v \in V$ of a graph $G : \mathit{Cgraph}(V, E)$, while assuming the "induction hypothesis" that $P(v')$ already holds for all vertices v' such that v is *waiting for* v', where *"waiting for"* is a binary relation chosen by us. The predicate $P(v)$ that we aim to prove for all vertices (*i.e.,* threads and sessions) is that v is reachable (see Theorem 5.7). The waiting relation we use is based on the $\mathsf{blocked}_{(\vec{e},h)}(v, v')$ relation, defined in §5. Waiting induction gives us the following induction hypotheses when proving that v is reachable:

For threads: If the thread is blocked on a session, we may assume that the session is reachable.
For sessions: The owners of the session that are not blocked on this session are reachable.

Reachability of threads. To show that a thread is reachable, we must show that it can take a step, or that it is blocked on an endpoint of a session that is reachable. By typing, either the thread can take a pure step, or is a session operation where all subexpressions are values. A session operation can proceed if the structure of the heap is valid, which we can conclude from the configuration invariant. The only possibility for a blocked operation is if we are trying to receive and the buffer we are trying to receive from is currently empty. Here, the waiting induction hypothesis applies (because $\mathsf{blocked}_{(\vec{e},h)}$ holds), using which we can show that the session that we are blocked on is reachable. Then, by the definition of reachability, the thread is also reachable.

Reachability of sessions. To show that a session s is reachable we must show that there exists a thread or session v that is (1) reachable, (2) holds an endpoint of s, and (3) is not blocked on s. We use the consistency of the buffers and local types of the session to show that there is an endpoint of s whose owner v is not blocked on this session (though v may be blocked on another session). This allows us to use the induction hypothesis to conclude that v is reachable (because $\mathsf{blocked}_{(\vec{e},h)}(\mathsf{Session}(s), v)$ does *not* hold). Then, using the definition of reachability for sessions, we conclude that s is reachable.

Main results. Theorem 5.7 is obtained by combining the reasoning above with Lemma 7.4 and Lemma 7.5. Global progress (Theorem 5.1) follows as an easy corollary. For two directions of the equivalence of partial deadlock/leak freedom with full reachability (Theorem 5.6), we show that none of the objects in a deadlock/leak are reachable, and vice versa that the set of non-reachable threads and channels forms a deadlock/leak if this set is nonempty.

8 MECHANIZATION

All our results have been mechanized in Coq using Iris Proof Mode [Krebbers et al. 2017, 2018] for the separation-logic part. The final results of our mechanization are Theorem 5.1, Theorem 5.6, and Theorem 5.7. We have also mechanized the translation from binary to multiparty in Figure 5 and have proved that it is type preserving. The mechanization is 10.4k lines of Coq code, consisting of 217 definitions, and 638 proved lemmas and theorems. Approximately half of the mechanization consists of results specific to our multiparty calculus, and the other half consists of the framework of Jacobs et al. [2022a], extended with support for n-ary graph transformations (§7.4).

Archive of the mechanization. The Coq mechanization can be found at Jacobs et al. [2022b].

Partial deadlock freedom and the empty type. A surprising technical result of the mechanization is that while global progress remains true in the presence of n-ary sum types, we discovered that partial deadlock freedom is by default **false** for languages that allow $n = 0$. The reason is that a thread can throw away endpoints by pattern matching on the empty sum type. While this pattern match will never execute because the empty type can only be produced by a looping expression (thus guaranteeing global progress), the thread can still lose an endpoint during a substitution step *before* the empty pattern match happens. This can create a partial deadlock for the threads holding the other endpoints of the session. To fix this, we amend the typing judgment $\Gamma \vdash e : \tau$ to require the variable context Γ to be empty when pattern matching on an empty sum type. This formally ensures that the thread's expression keeps track of all endpoints and does not lose any. This does not limit the expressivity of empty types because one can obtain a value of any type from an empty pattern match, including a function that can eat the remaining variables in the context.

9 RELATED WORK

To relate MPGV to the existing body of work it is helpful to consider two axes of categorization: *mechanization* and session type *philosophy*. The use of a proof assistant to mechanize correctness results has only been taken up recently by the session type community. Typeset pen-and-paper proofs and appeals to results in logic (*e.g.*, cut elimination) still constitute the status quo. We summarize mechanizations of session types below, but remark that only two works target mechanization of deadlock freedom up to date: Castro-Perez et al. [2021] for a single multiparty session and Jacobs et al. [2022a] for GV-style binary session types.

At first blush, session types can be distinguished into *binary* and *multiparty*. Whereas binary session types restrict the concurrent interaction to two participants, multiparty session types allow an arbitrary but statically determined number of participants ("roles"), by complementing the local perspective of a participant with a global type. A more foundational distinction, especially given the unifying nature of MPGV, is underlying philosophy. Session types [Honda 1993; Honda et al. 1998] have been conceived as a typing discipline for process calculi and as such preserve the fundamental characteristics of concurrent computation. Concurrent computation is inherently *non-deterministic* and may also give raise to *deadlocks*. For example, the below session-typed program from [Gay and Vasconcelos 2010] (page 38) is well-typed but deadlocks:

$$\langle \mathsf{let}\ c_1 = \mathsf{request}\ a_1\ \mathsf{in}\ \mathsf{let}\ c_2 = \mathsf{request}\ a_2\ \mathsf{in}\ \mathsf{let}\ (c_1, x) = \mathsf{receive}\ c_1\ \mathsf{in}\ \mathsf{send}\ v\ c_2 \rangle \quad || $$
$$\langle \mathsf{let}\ d_1 = \mathsf{accept}\ a_1\ \mathsf{in}\ \mathsf{let}\ d_2 = \mathsf{accept}\ a_2\ \mathsf{in}\ \mathsf{let}\ (d_1, y) = \mathsf{receive}\ d_2\ \mathsf{in}\ \mathsf{send}\ w\ d_1 \rangle$$

The program composes two threads (processes) in parallel, amounting to two binary interleaved sessions a_1 and a_2. Sessions are initiated by matching a request for a session (request a_1) with an accept for that session (accept a_1) creating two new endpoints per session (c_1 and c_2). The interleaving of the two sessions causes a deadlock: the receive on c_1 blocks the send on c_2, which is necessary for the former. The pairing of session requests with matching accepts is non-deterministic.

For example, if we compose the two threads with a third thread that is also accepting a session a_1, then only one of the two accepting threads will be chosen.

This initialization pattern carries over to multiparty sessions [Honda et al. 2008, 2016]. In the multiparty case a request is parameterized with the number of participants n and accepts with the role names ranging from 1 to $n - 1$. Like binary session types, multiparty session types that assume this initialization pattern can deadlock. In particular, deadlocks can arise if a participant simultaneously engages in several sessions. A strategy adopted by some multiparty session type work (*e.g.,* Castro-Perez et al. [2021]) is to restrict a program to a single global multiparty session, precluding dynamic session spawning and first-class sessions. Alternatively, advanced multiparty session-type systems [Coppo et al. 2013; Bettini et al. 2008; Coppo et al. 2016] employ extrinsic orders/priorities to rule out deadlocks among interleaved multiparty sessions, requiring order annotations in addition to global type declarations.

We refer to the line of session type work that adopts the initialization pattern shown above, which separates session creation from thread spawning, as *traditional* session types. We like to contrast this line of work with the one that adopts an initialization pattern based on cut, inspired by the Curry-Howard correspondence between linear logic and the session-typed π-calculus [Caires and Pfenning 2010; Wadler 2012; Lindley and Morris 2015; Kokke et al. 2019], which we refer to as *logic-based* session types. Logic-based session types come with strong guarantees out of the box. These include, besides session fidelity, *deadlock freedom*. Given our focus on deadlock freedom, MPGV adopts the initialization pattern of logic-based session types and generalizes GV's fork construct [Wadler 2012; Lindley and Morris 2015, 2016b, 2017; Fowler et al. 2019, 2021] for binary session types to the n-ary setting. The above program would thus not type check in MPGV.

A recent extension of GV by Fowler et al. [2021], Hypersequent GV (HGV), adopts hypersequents [Montesi and Peressotti 2018; Kokke et al. 2019], inspired by Avron [1991], to facilitate a tighter correspondence to the session-typed π-calculus and simplify GV's meta theory by accounting for the forest topology of runtime structures. While this account is reminiscent of our notion of logical topology, the specifics of HGV and our MPGV system are quite different. Most notably, HGV employs structural congruences for runtime typing, whereas our dynamics operates on a flat thread pool and heap (allowing an arbitrary thread to step without prior application of structural congruences) and our runtime typing relies on separation logic and connectivity graphs.

We next review the individual related work in more detail, referring to our categorization of traditional and logic-based session types as convenient. Given our focus on mechanization, we start with mechanized related work and then conclude with non-mechanized related work.

Mechanized. The only existing work on mechanizing deadlock freedom for multiparty session types is *Zooid*, a DSL by Castro-Perez et al. [2021] embedded in Coq. Although a traditional session type language in spirit, Zooid does neither support session spawning nor delegation, but restricts a program to a single global multiparty session. Zooid programs thus rule out deadlocks caused by multiparty session interleavings by construction. Thanks to a shallow embedding in Coq, Zooid programs can be extracted from Coq to OCaml via Coq's extraction mechanism. Send and receive operations are handled as monadic operations in which the endpoint is implicit (because there is a unique global session). A shallow embedding of binders works in this context because Zooid variables can only contain purely functional data, which can be represented as Coq data. Our definition of (runtime) global types and projections is inspired by Zooid.

MPGV not only differs from Zooid in its support for multiple interacting sessions, first-class endpoints, dynamic spawning, and delegation, but also in statement and precision of the deadlock freedom property. Our mechanization guarantees global progress—including the stronger notions of partial deadlock/leak freedom. Zooid's main result, on the other hand, is phrased in terms of traces, asserting that for all traces produced by a well-typed process there *exists* a matching trace in the larger system. This result relies on properties of global types from the literature and also assumes the ability to choose a favorable schedule. Our mechanization in contrast states deadlock freedom for *all* executions/schedules and gives a closed end-to-end proof in Coq.

Jacobs et al. [2022a] contribute a mechanization of deadlock freedom for a variant of GV, and thus target *binary* session types. Like our mechanization, theirs accommodates dynamic session spawning and delegation, but restricted to the binary setting. Jacobs et al. [2022a] moreover contribute the notion of a *connectivity graph*, a parametric proof method for deadlock freedom, relying on acyclicity of the communication topology. We extend this proof method with n-ary operations and support of cyclic connectivity within a session governed by consistency. Our generalization to n-ary operations also enables our encoding of binary session types in MPGV (§4). Unlike Jacobs et al.'s variant of GV, our MPGV system moreover supports choice, and thus provides the first mechanization of deadlock and leak freedom for binary session types with choice.

Moreover, there exists work on mechanizing the metatheory of binary session types. Thiemann [2019] proves type safety of a linear λ-calculus with session types inspired by GV. The result does not include deadlock nor memory leak freedom. Hinrichsen et al. [2021] prove type safety for a comprehensive session-typed language with locks, subtyping and polymorphism using Iris in Coq. Their type system is affine, which means that deadlocks are considered safe. Tassarotti et al. [2017] prove termination preservation of a compiler for an affine session-typed language using Iris in Coq.

More distantly, there exist various mechanized results involving the π-calculus. Goto et al. [2016] prove type safety for a π-calculus with a polymorphic session type system in Coq. Their type system allows dropping channels, and hence does not enjoy deadlock nor memory leak freedom. Ciccone and Padovani [2020] mechanize dependent binary session types by embedding them into a π-calculus in Agda. They prove subject reduction (*i.e.,* preservation) and that typing is preserved by structural congruence, but not deadlock or memory leak freedom. Similarly, Zalakain and Dardha [2021] mechanize subject reduction of a resource-aware π-calculus, focusing on the handling of linear resources through leftover typing. Gay et al. [2020] study various notions of duality in Agda, and show that distribution laws for duality over the recursion operator are unsound. We adopted their approach of using coinductive types for mechanizing general recursive session types. Lastly, mechanizations of choreographic languages [Montesi 2020; Cruz-Filipe et al. 2021a,b] focus on determinism, confluence, and Turing completeness, with deadlock freedom holding by design.

Non-mechanized. The work that is most closely related to ours in terms of underlying philosophy but non-mechanized is the work by Carbone et al. [2015, 2016, 2017] on coherence proofs. The authors introduce a proof theory grounded in classical linear logic via a Curry-Howard correspondence, illuminating the connection between binary and multiparty session types, in a π-calculus setting. The correspondence is due the novel notion of *coherence*, which generalizes duality known from binary session types to compatibility of local types with a global type of a multiparty session. Given a coherence derivation, an n-ary cut permits composing n participants concurrently, similar to our n-ary fork. Coherence also enables a semantic-preserving translation from multiparty sessions to corresponding binary sessions via an arbiter process [Carbone et al. 2016]. Deadlock freedom follows from cut admissibility and cut elimination, giving a normalization argument. Such an argument is made possible by using cut reductions for the semantics and restricting to a non-Turing complete calculus without loops or recursion. In contrast, we provide a complete mechanization of deadlock freedom of an n-ary session-typed functional language, with recursive types, first-class endpoints, and a realistic asynchronous operational semantics based on an unstructured thread pool. Our encoding of binary sessions in MPGV moreover does not require an arbiter process.

Similarly, Caires and Pérez [2016] embed multiparty session types in a binary calculus by a translation from a global type to a *medium process*. Instead of communicating with each other, the participants communicate with the central medium process. This approach inherits deadlock freedom from the surrounding binary calculus, but requires central coordination and sequentializes the communication. Toninho and Yoshida [2018] show that the interconnection networks of classical linear logic (CLL) are strictly less expressive than those of a multiparty session calculus. *Partial multiparty compatibility* is used to define a new binary cut rule that can form circular interconnections but preserves the deadlock-freedom of CLL, albeit for a single multiparty session.

More distantly related are works that use Kobayashi-style type systems [Kobayashi 1997, 2002, 2006; Giachino et al. 2014; Kobayashi and Laneve 2017] that enrich channel typing with usage information and partial orders to rule out cyclic dependencies among channel actions. In the traditional multiparty setting this is most notably the work by Coppo et al. [2013]; Bettini et al. [2008]; Coppo et al. [2016], which contributes an *interaction type system* that ensures deadlock freedom not only within but also between several multiparty sessions. This work not only differs from MPGV in that it requires ordering annotations in addition to global type declarations, but also in the statement of the global progress property. To account for processes that lack a communication partner, a possibility in the traditional setting, progress is stated relative to a catalyzing process that, if present, would allow the closed program to step. MPGV sets itself additionally apart in its tight integration with a functional language.

Kobayashi-style systems have also been adopted for logic-based binary session types [Dardha and Gay 2018; Kokke and Dardha 2021b,a]. The authors introduce a multicut, which allows for circular topologies within a session. To rule out deadlocks by type checking, session types must be annotated with *priorities*. Priority polymorphism has been used by Padovani [2014] to support cyclic interleavings of recursive processes. Partial order annotations, called *worlds*, are also used by Balzer et al. [2019], in a logic-based binary session type calculus that combines linear and shared [Balzer and Pfenning 2017; Balzer et al. 2018] sessions. Shared session types introduce a controlled form of aliasing, an extension we would like to consider in future work.

A somewhat orthogonal approach to ensuring progress in the presence of dynamic thread allocation is to make global types more powerful. While traditional multiparty session types involve a fixed number of participants per session, Deniélou and Yoshida [2011]; Demangeon and Honda [2012]; Hu and Yoshida [2017] proposed extensions of single-session systems to make that number dynamic. This line of work does not support sessions as first-class data, and the expressivity is orthogonal to GV and MPGV. Hence, extending MPGV with a dynamic number of participants is an interesting extension for future work.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful feedback and especially for their encouragement to explore a more permissive consistency condition than compliance with a global type, resulting in §6. We are grateful to Jorge Pérez, Bas van den Heuvel, Dan Frumin, and Bernardo Toninho for discussions on this paper and related work. Robbert Krebbers was supported by the Dutch Research Council (NWO), project 016.Veni.192.259. Stephanie Balzer was supported in part by AFOSR under grant FA9550-21-1-0385 (Tristan Nguyen, program manager) and by the National Science Foundation under award number CCF-2211996. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the AFOSR or NSF.

REFERENCES

Arnon Avron. 1991. Hypersequents, Logical Consequence and Intermediate Logics for Concurrency. *Annals of Mathematics and Artificial Intelligence* 4 (1991), 225–248. https://doi.org/10.1007/BF01531058

Stephanie Balzer and Frank Pfenning. 2017. Manifest Sharing with Session Types. *PACMPL* 1, ICFP (2017), 37:1–37:29. https://doi.org/10.1145/3110281

Stephanie Balzer, Frank Pfenning, and Bernardo Toninho. 2018. A Universal Session Type for Untyped Asynchronous Communication. In *CONCUR (LIPIcs, Vol. 118)*. 30:1–30:18. https://doi.org/10.4230/LIPIcs.CONCUR.2018.30

Stephanie Balzer, Bernardo Toninho, and Frank Pfenning. 2019. Manifest Deadlock-Freedom for Shared Session Types. In *ESOP (LNCS, Vol. 11423)*. 611–639. https://doi.org/10.1007/978-3-030-17184-1_22

Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. 2008. Global Progress in Dynamically Interleaved Multiparty Sessions. In *CONCUR (LNCS, Vol. 5201)*. 418–433. https://doi.org/10.1007/978-3-540-85361-9_33

Luís Caires and Jorge A. Pérez. 2016. Multiparty Session Types Within a Canonical Binary Theory, and Beyond. In *FORTE (LNCS, Vol. 9688)*. 74–95. https://doi.org/10.1007/978-3-319-39570-8_6

Luís Caires and Frank Pfenning. 2010. Session Types as Intuitionistic Linear Propositions. In *CONCUR (LNCS, Vol. 6269)*. 222–236. https://doi.org/10.1007/978-3-642-15375-4_16

Marco Carbone, Sam Lindley, Fabrizio Montesi, Carsten Schürmann, and Philip Wadler. 2016. Coherence Generalises Duality: A Logical Explanation of Multiparty Session Types. In *CONCUR (LIPIcs, Vol. 59)*. 33:1–33:15. https://doi.org/10.4230/LIPIcs.CONCUR.2016.33

Marco Carbone, Fabrizio Montesi, Carsten Schürmann, and Nobuko Yoshida. 2015. Multiparty Session Types as Coherence Proofs. In *CONCUR (LIPIcs, Vol. 42)*. 412–426. https://doi.org/10.4230/LIPIcs.CONCUR.2015.412

Marco Carbone, Fabrizio Montesi, Carsten Schürmann, and Nobuko Yoshida. 2017. Multiparty session types as coherence proofs. *Acta Informatica* 54, 3 (2017), 243–269. https://doi.org/10.1007/s00236-016-0285-y

David Castro-Perez, Francisco Ferreira, Lorenzo Gheri, and Nobuko Yoshida. 2021. Zooid: A DSL for Certified Multiparty Computation: From Mechanised Metatheory to Certified Multiparty Processes. In *PLDI*. 237–251. https://doi.org/10.1145/3453483.3454041

David Castro-Perez, Francisco Ferreira, and Nobuko Yoshida. 2020. EMTST: Engineering the Meta-theory of Session Types. In *TACAS (2) (LNCS, Vol. 12079)*. 278–285. https://doi.org/10.1007/978-3-030-45237-7_17

Ruofei Chen, Stephanie Balzer, and Bernardo Toninho. 2022. Ferrite: A Judgmental Embedding of Session Types in Rust. In *ECOOP (LIPIcs, Vol. 222)*. 22:1–22:28. https://doi.org/10.4230/LIPIcs.ECOOP.2022.22

Luca Ciccone and Luca Padovani. 2020. A Dependently Typed Linear π-Calculus in Agda. In *PPDP*. 8:1–8:14. https://doi.org/10.1145/3414080.3414109

Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, and Nobuko Yoshida. 2013. Inference of Global Progress Properties for Dynamically Interleaved Multiparty Sessions. In *COORDINATION*. https://doi.org/10.1007/978-3-642-38493-6_4

Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. 2016. Global Progress for Dynamically Interleaved Multiparty Sessions. *MSCS* 26, 2 (2016), 238–302. https://doi.org/10.1017/S0960129514000188

Karl Crary, Robert Harper, and Sidd Puri. 1999. What is a Recursive Module?. In *PLDI*. 50–63. https://doi.org/10.1145/301618.301641

Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. 2021a. Certifying Choreography Compilation. In *ICTAC (LNCS, Vol. 12819)*. 115–133. https://doi.org/10.1007/978-3-030-85315-0_8

Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. 2021b. Formalising a Turing-Complete Choreographic Language in Coq. In *ITP (LIPIcs, Vol. 193)*. 15:1–15:18. https://doi.org/10.4230/LIPIcs.ITP.2021.15

Ornela Dardha and Simon J. Gay. 2018. A New Linear Logic for Deadlock-Free Session-Typed Processes. In *FOSSACS (LNCS, Vol. 10803)*. 91–109. https://doi.org/10.1007/978-3-319-89366-2_5

Romain Demangeon and Kohei Honda. 2012. Nested Protocols in Session Types. In *CONCUR (LNCS, Vol. 7454)*. 272–286. https://doi.org/10.1007/978-3-642-32940-1_20

Pierre-Malo Deniélou and Nobuko Yoshida. 2011. Dynamic multirole session types. In *POPL*. 435–446. https://doi.org/10.1145/1926385.1926435

Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. 2006. Session Types for Object-Oriented Languages. In *ESOP (LNCS, Vol. 4067)*. 328–352. https://doi.org/10.1007/11785477_20

Simon Fowler, Wen Kokke, Ornela Dardha, Sam Lindley, and J. Garrett Morris. 2021. Separating Sessions Smoothly. In *CONCUR (LIPIcs, Vol. 203)*. 36:1–36:18. https://doi.org/10.4230/LIPIcs.CONCUR.2021.36

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

Simon J. Gay, Peter Thiemann, and Vasco T. Vasconcelos. 2020. Duality of Session Types: The Final Cut. In *PLACES (EPTCS, Vol. 314)*. 23–33. https://doi.org/10.4204/EPTCS.314.3

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

Silvia Ghilezan, Jovanka Pantovic, Ivan Prokic, Alceste Scalas, and Nobuko Yoshida. 2021. Precise subtyping for asynchronous multiparty sessions. *PACMPL* 5, POPL (2021), 1–28. https://doi.org/10.1145/3434297

Elena Giachino, Naoki Kobayashi, and Cosimo Laneve. 2014. Deadlock Analysis of Unbounded Process Networks. In *CONCUR (LNCS, Vol. 8704)*. 63–77. https://doi.org/10.1007/978-3-662-44584-6_6

Matthew A. Goto, Radha Jagadeesan, Alan Jeffrey, Corin Pitcher, and James Riely. 2016. An Extensible Approach to Session Polymorphism. *MSCS* 26, 3 (2016), 465–509. https://doi.org/10.1017/S0960129514000231

Jonas Kastberg Hinrichsen, Daniël Louwrink, Robbert Krebbers, and Jesper Bengtson. 2021. Machine-checked semantic session typing. In *CPP*. 178–198. https://doi.org/10.1145/3437992.3439914

Kohei Honda. 1993. Types for Dyadic Interaction. In *CONCUR (LNCS, Vol. 715)*. 509–523. https://doi.org/10.1007/3-540-57208-2_35

Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. 1998. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *ESOP (LNCS, Vol. 1381)*. 122–138. https://doi.org/10.1007/BFb0053567

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty Asynchronous Session Types. In *POPL*. 273–284. https://doi.org/10.1145/1328438.1328472

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2016. Multiparty Asynchronous Session Types. *J. ACM* 63, 1 (2016), 9:1–9:67. https://doi.org/10.1145/2827695

Raymond Hu and Nobuko Yoshida. 2017. Explicit Connection Actions in Multiparty Session Types. In *FASE (LNCS, Vol. 10202)*. 116–133. https://doi.org/10.1007/978-3-662-54494-5_7

Keigo Imai, Nobuko Yoshida, and Shoji Yuen. 2019. Session-Ocaml: A Session-Based Library with Polarities and Lenses. *Science of Computer Programming* 172 (2019), 135–159. https://doi.org/10.1016/j.scico.2018.08.005

Keigo Imai, Shoji Yuen, and Kiyoshi Agusa. 2010. Session Type Inference in Haskell. In *PLACES (EPTCS, Vol. 69)*. 74–91. https://doi.org/10.4204/EPTCS.69.6

Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. 2022a. Connectivity graphs: a method for proving deadlock freedom based on separation logic. *PACMPL* 6, POPL, 1–33. https://doi.org/10.1145/3498662

Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. 2022b. MPGV Coq development. https://doi.org/10.5281/zenodo.6883734

Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. 2015. Session Types for Rust. In *WGP*. 13–22. https://doi.org/10.1145/2808098.2808100

Alex C. Keizer, Henning Basold, and Jorge A. Pérez. 2021. Session Coalgebras: A Coalgebraic View on Session Types and Communication Protocols. In *ESOP (LNCS, Vol. 12648)*. 375–403. https://doi.org/10.1007/978-3-030-72019-3_14

Naoki Kobayashi. 1997. A Partially Deadlock-Free Typed Process Calculus. In *LICS*. 128–139. https://doi.org/10.1109/LICS.1997.614941

Naoki Kobayashi. 2002. A Type System for Lock-Free Processes. *I&C* 177, 2 (2002), 122–159. https://doi.org/10.1006/inco.2002.3171

Naoki Kobayashi. 2006. A New Type System for Deadlock-Free Processes. In *CONCUR (LNCS, Vol. 4137)*. 233–247. https://doi.org/10.1007/11817949_16

Naoki Kobayashi and Cosimo Laneve. 2017. Deadlock Analysis of Unbounded Process Networks. *Inf. Comput.* 252 (2017), 48–70. https://doi.org/10.1016/j.ic.2016.03.004

Wen Kokke. 2019. Rusty Variation: Deadlock-free Sessions with Failure in Rust. In *ICE (EPTCS, Vol. 304)*. 48–60. https://doi.org/10.4204/EPTCS.304.4

Wen Kokke and Ornela Dardha. 2021a. Deadlock-Free Session Types in Linear Haskell. In *Haskell Symposium*. 1–13. https://doi.org/10.1145/3471874.3472979

Wen Kokke and Ornela Dardha. 2021b. Prioritise the Best Variation. In *FORTE (LNCS, Vol. 12719)*. 100–119. https://doi.org/10.1007/978-3-030-78089-0_6

Wen Kokke, Fabrizio Montesi, and Marco Peressotti. 2019. Better Late Than Never: a Fully-Abstract Semantics for Classical Processes. *PACMPL* 3, POPL (2019), 24:1–24:29. https://doi.org/10.1145/3290337

Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: A General, Extensible Modal Framework for Interactive Proofs in Separation Logic. *PACMPL* 2, ICFP (2018), 77:1–77:30. https://doi.org/10.1145/3236772

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive Proofs in Higher-Order Concurrent Separation Logic. In *POPL*. 205–217. https://doi.org/10.1145/3009837.3009855

Sam Lindley and J. Garrett Morris. 2015. A Semantics for Propositions as Sessions. In *ESOP (LNCS, Vol. 9032)*. 560–584. https://doi.org/10.1007/978-3-662-46669-8_23

Sam Lindley and J. Garrett Morris. 2016a. Embedding Session Types in Haskell. In *Haskell Symposium*. 133–145. https://doi.org/10.1145/2976002.2976018

Sam Lindley and J. Garrett Morris. 2016b. Talking Bananas: Structural Recursion For Session Types. In *ICFP*. 434–447. https://doi.org/10.1145/2951913.2951921

Sam Lindley and J. Garrett Morris. 2017. Lightweight Functional Session Types. In *Behavioural Types: from Theory to Tools*. https://homepages.inf.ed.ac.uk/slindley/papers/fst.pdf

Fabrizio Montesi. 2020. Introduction to Choreographies. (2020). https://www.fabriziomontesi.com/teaching/ct-2020/files/chor-notes.pdf Draft.

Fabrizio Montesi and Marco Peressotti. 2018. Classical Transitions. *CoRR* abs/1803.01049 (2018). arXiv:1803.01049 http://arxiv.org/abs/1803.01049

Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. 2009. Global Principal Typing in Partially Commutative Asynchronous Sessions. In *ESOP (LNCS, Vol. 5502)*. 316–332. https://doi.org/10.1007/978-3-642-00590-9_23

Peter W. O'Hearn and David J. Pym. 1999. The Logic Of Bunched Implications. *Bulletin of Symbolic Logic* 5, 2 (1999), 215–244. https://doi.org/10.2307/421090

Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *CSL (LNCS, Vol. 2142)*. 1–19. https://doi.org/10.1007/3-540-44802-0_1

Luca Padovani. 2014. Deadlock and lock freedom in the linear π-calculus. In *LICS*. 72:1–72:10. https://doi.org/10.1145/2603088.2603116

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

Frank Pfenning and Dennis Griffith. 2015. Polarized Substructural Session Types. In *FoSSaCS (LNCS, Vol. 9034)*. 3–22. https://doi.org/10.1007/978-3-662-46678-0_1

Riccardo Pucella and Jesse A. Tov. 2008. Haskell Session Types with (Almost) No Class. In *Haskell Symposium*. 25–36. https://doi.org/10.1145/1411286.1411290

Arjen Rouvoet, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. 2020. Intrinsically-Typed Definitional Interpreters for Linear, Session-Typed Languages. In *CPP*. 284–298. https://doi.org/10.1145/3372885.3373818

Alceste Scalas and Nobuko Yoshida. 2016. Lightweight Session Programming in Scala. In *ECOOP (LIPIcs, 56)*. 21:1–21:28. https://doi.org/10.4230/LIPIcs.ECOOP.2016.21

Alceste Scalas and Nobuko Yoshida. 2019a. Less is more: multiparty session types revisited. *PACMPL* 3, POPL (2019), 30:1–30:29. https://doi.org/10.1145/3290343

Alceste Scalas and Nobuko Yoshida. 2019b. Less is more: multiparty session types revisited (technical report). https://www.doc.ic.ac.uk/research/technicalreports/2018/DTRS18-6.pdf

Joseph Tassarotti, Ralf Jung, and Robert Harper. 2017. A Higher-Order Logic for Concurrent Termination-Preserving Refinement. In *ESOP (LNCS, Vol. 10201)*. 909–936. https://doi.org/10.1007/978-3-662-54434-1_34

Peter Thiemann. 2019. Intrinsically-Typed Mechanized Semantics for Session Types. In *PPDP*. 19:1–19:15. https://doi.org/10.1145/3354166.3354184

Bernardo Toninho. 2015. *A Logical Foundation for Session-Based Concurrent Computation.* Ph. D. Dissertation. Carnegie Mellon University and New University of Lisbon.

Bernardo Toninho, Luís Caires, and Frank Pfenning. 2013. Higher-Order Processes, Functions, and Sessions: A Monadic Integration. In *ESOP (LNCS, Vol. 7792)*. 350–369. https://doi.org/10.1007/978-3-642-37036-6_20

Bernardo Toninho and Nobuko Yoshida. 2018. Interconnectability of Session-Based Logical Processes. *TOPLAS* 40, 4 (2018), 17:1–17:42. https://doi.org/10.1145/3242173

Philip Wadler. 2012. Propositions as Sessions. In *ICFP*. 273–286. https://doi.org/10.1145/2364527.2364568

Uma Zalakain and Ornela Dardha. 2021. π with Leftovers: A Mechanisation in Agda. In *FORTE (LNCS, Vol. 12719)*. 157–174. https://doi.org/10.1007/978-3-030-78089-0_9