Deadlock-Free Session Types in Linear Haskell

Wen Kokke
University of Edinburgh
Edinburgh, Scotland
wen.kokke@ed.ac.uk

Ornela Dardha
University of Glasgow
Glasgow, Scotland
ornela.dardha@glasgow.ac.uk

Abstract

Priority Sesh is a library for session-typed communication in Linear Haskell which offers strong compile-time correctness guarantees. Priority Sesh offers two deadlock-free APIs for session-typed communication. The first guarantees deadlock freedom by restricting the process structure to trees and forests. It is simple and composeable, but rules out cyclic structures. The second guarantees deadlock freedom via priorities, which allows the programmer to safely use cyclic structures as well.

Our library relies on Linear Haskell to guarantee linearity, which leads to easy-to-write session types and highly idiomatic code, and lets us avoid the complex encodings of linearity in the Haskell type system that made previous libraries difficult to use.

CCS Concepts: • **Theory of computation** → *Linear logic*; *Type theory*.

Keywords: session types, linear haskell, deadlock freedom

ACM Reference Format:
Wen Kokke and Ornela Dardha. 2021. Deadlock-Free Session Types in Linear Haskell. In *Proceedings of ACM Conference (Conference'17)*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Session types are a type formalism used to specify and verify communication protocols [Honda 1993; Honda et al. 1998, 2008; Takeuchi et al. 1994]. They've been studied extensively in the context of the π-calculus [Sangiorgi and Walker 2001], a process calculus for communication an concurrency, and in the context of concurrent λ-calculi, such as the GV family of languages ["Good Variation", Fowler et al. 2019; Gay and Vasconcelos 2010; Lindley and Morris 2015; Wadler 2014].

Session types have been implemented in various programming languages. We give a detailed overview in section 4, and Orchard and Yoshida [2017] provide a complete survey of session type implementations in Haskell.

The main difficulty when implementing session types in most programming languages is *linearity*, *i.e.*, the guarantee that each channel endpoint is used *exactly once*. There are several different approaches to guaranteeing linearity, but the main distinction is between *dynamic* [Padovani 2017; Scalas et al. 2017; Scalas and Yoshida 2016] and *static* [Lindley and Morris 2016, 2017; Pucella and Tov 2008] usage checks. With dynamic checks, using a channel endpoint more than once simply throws a runtime error. With static checks, usage is *somehow* encoded into the type system of the host language usually by encoding the entire linear typing environment into the type system using a parameterised or graded monad. Such encodings are only possible if the type system of the host language is expressive enough. However, such encodings are often quite complex, and result in a trade-off between easy-to-write session types and idiomatic programs.

Moreover, these implementations only focus on the most basic features of session types and often ignore more advanced ones, such as channel delegation or deadlock freedom: Neubauer and Thiemann [2004] only provide single session channels; Pucella and Tov [2008] provide multiple channels, but only the building blocks for channel delegation; Imai et al. [2010] extend Pucella and Tov [2008] and provide full delegation. None of these works address deadlock freedom. Lindley and Morris [2016] provide an implementation of GV into Haskell building on the work of Polakow [2015]. To the best of our knowledge, this is the only work that guarantees deadlock freedom of session types in Haskell, albeit in a simple form. In GV, all programs must have *tree-shaped* process structures. The process structure of a program is an undirected graph, where nodes represent processes, and edges represent the channels connecting them. (We explore this in more detail in section 2.3.) Therefore, deadlock freedom is guaranteed by design: session types rule out deadlocks over a single channel, and the tree-restriction rules out sharing multiple channels between two processes. While Lindley and Morris [2016] manage to implement more advanced properties, the tree restriction rules out many interesting programs which have *cyclic* process structure, but are deadlock free.

Recent works by Padovani and Novara [2015] and Kokke and Dardha [PGV, [n.d.]] integrate *priorities* [Kobayashi 2006;

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Padovani 2014] into functional languages. Priorities are natural numbers that abstractly represent the time at which a communication action happens. Priority-based type systems check that there are *no* cycles in the communication graph. The communication graph is a directed graph where nodes represent dual communication actions, and directed edges represent one action *must happen* before another. (We explore this in more detail in section 2.4.) Such type systems are *more* expressive, as they allow programs to have *cyclic* process structure, as long as they have an *acyclic* communication graph.

With the above in mind, our research goals are as follows:

Q1 Can we have easy-to-write session types, easy linearity checks and idiomatic code at the same time?
Q2 Can we address not only the main features of session types, but also advanced ones, such as full delegation, recursion, and deadlock freedom of programs with cyclic process structure?

Our `priority-sesh` library answers both questions *mostly* positively. We sidestep the problems with encoding linearity in Haskell by using Linear Haskell [Bernardy et al. 2018], which has native support for linear types. The resulting session type library presented in sections 2.2 and 2.3 has both easy-to-write session types, easy linearity checks, and idiomatic code. Moving to **Q2**, the library has full delegation, recursion, and the variant in section 2.3 even guarantees of deadlock freedom, albeit by restricting the process structure to trees and forests. In section 2.4, we implement another variant which uses priorities to ensure deadlock freedom of programs with cyclic processes structure. The ease-of-writing suffers a little, as the programmer has to manually write priorities, though this isn't a *huge* inconvenience. Unfortunately, GHC's ability to reason about type-level naturals currently is *not* as powerful as to allow the programmer to easily write priority-polymorphic code, which is required for *recursion*. Hence, while we address deadlock freedom for cyclic process structures, we do so only for the *finite* setting.

Contributions. In section 2, we present Priority Sesh, an implementation of deadlock free session types in Linear Haskell which is:

- the *first* implementation of session types to take advantage of Linear Haskell for linearity checking, and producing easy-to-write session types and highly idiomatic code;
- the *first* implementation of session types in Haskell to guarantee deadlock freedom of programs with cyclic process structure via *priorities*; and
- the *first* embedding of priorities into an existing mainstream programming language.

In section 3, we:

- present a variant of Priority GV [Kokke and Dardha [n.d.]]—the calculus upon which Priority Sesh is based—with asynchronous communication and session cancellation following Fowler et al. [2019] and *explicit* lower bounds on the sequent, rather than lower bounds inferred from the typing environment; and
- show that Priority Sesh is related to Priority GV via monadic reflection.

2 What is Priority Sesh?

In this section we introduce Priority Sesh in three steps:

- in section 2.1, we build a small library of *linear* or *one-shot channels* based on MVars [Peyton Jones et al. [n.d.]];
- in section 2.2, we use these one-shot channels to build a small library of *session-typed channels* [Dardha et al. 2012]; and
- in section 2.4, we decorate these session types with *priorities* to guarantee deadlock-freedom [Kokke and Dardha [n.d.]].

It is important to notice that the meaning of linearity in *one-shot channels* differs from linearity in *session channels*. A linear or one-shot channel originates from the linear π-calculus [Kobayashi et al. 1999; Sangiorgi and Walker 2001], where each endpoint of a channel must be used for *exactly one* send or receive operation, whereas linearity in the context of session-typed channels, it means that each step in the protocol is performed *exactly once*, but the channel itself is used multiple times.

Priority Sesh is written in Linear Haskell [Bernardy et al. 2018]. The type $\multimap$ is syntactic sugar for the linear arrow `%1->`. Familiar definitions refer to linear variants packaged with `linear-base`[1] (*e.g.*, *IO*, *Functor*, *Bifunctor*, *Monad*) or with Priority Sesh (*e.g.*, *MVar*).

We colour the Haskell definitions which are a part of Sesh: **red** for functions and constructors; **blue** for types and type families; and **emerald** for priorities and type families acting on priorities.

2.1 One-shot channels

We start by building a small library of *linear* or *one-shot channels*, *i.e.*, channels that must be use *exactly once* to send or receive a value.

The one-shot channels are at the core of our library, and their efficiency is crucial to the overall efficiency of Priority Sesh. However, we do not aim to present an efficient implementation here, rather we aim to present a compact implementation with the correct behaviour.

Channels. A one-shot channel has two endpoints, Send_1 and Recv_1, which are two copies of the same *MVar*.

[1] https://hackage.haskell.org/package/linear-base

```
newtype Send₁ a = Send₁ (MVar a)
newtype Recv₁ a = Recv₁ (MVar a)

new₁ :: IO (Send₁ a, Recv₁ a)
new₁ = do (mvar_s, mvar_r) ← dup2 ⟨$⟩ newEmptyMVar
          return (Send₁ (unur mvar_s), Recv₁ (unur mvar_r))
```

The *newEmptyMVar* function returns an *unrestricted MVar*, which may be used non-linearly, *i.e.*, as many times as one wants. The *dup2* function creates two (unrestricted) copies of the *MVar*. The *unur* function casts each *unrestricted* copy to a *linear* copy. Thus, we end up with two copies of an *MVar*, each of which must be used *exactly once*.

We implement $send_1$ and $recv_1$ as aliases for the corresponding *MVar* operations.

```
send₁ :: Send₁ a ⊸ a ⊸ IO ()
send₁ (Send₁ mvar_s) x = putMVar mvar_s x

recv₁ :: Recv₁ a ⊸ IO a
recv₁ (Recv₁ mvar_r) = takeMVar mvar_r
```

The *MVar* operations implement the correct blocking behaviour for asynchronous one-shot channels: the $send_1$ operation is non-blocking, and the $recv_1$ operations blocks until a value becomes available.

Synchronisation. We use $Send_1$ and $Recv_1$ to implement a construct for one-shot synchronisation between two processes, $Sync_1$, which consists of two one-shot channels. To synchronise, each process sends a unit on the one channel, then waits to receive a unit on the other channel.

```
data Sync₁ = Sync₁ (Send₁ ()) (Recv₁ ())

newSync₁ :: IO (Sync₁, Sync₁)
newSync₁ = do (ch_s1, ch_r1) ← new₁
              (ch_s2, ch_r2) ← new₁
              return (Sync₁ ch_s1 ch_r2, Sync₁ ch_s2 ch_r1)

sync₁ :: Sync₁ ⊸ IO ()
sync₁ (Sync₁ ch_s ch_r) = do send₁ ch_s (); recv₁ ch_r
```

Cancellation. We implement *cancellation* for one-shot channels. One-shot channels are created in the linear *IO* monad, so *forgetting* to use a channel results in a complaint from the type-checker. However, it is possible to *explicitly* drop values whose types implement the *Consumable* class, using *consume* :: $a \multimap ()$. The ability to cancel communications is important, as it allows us to safely throw an exception *without violating linearity*, assuming that we cancel all open channels before doing so.

One-shot channels implement *Consumable* by simply dropping their *MVar*s. The Haskell runtime throws an exception when a "thread is blocked on an *MVar*, but there are no other references to the *MVar* so it can't ever continue."[2] Practically, *consumeAndRecv* throws a *BlockedIndefinitelyOnMVar* exception, whereas *consumeAndSend* does not:

```
consumeAndRecv = do                 consumeAndSend = do u
  (ch_s, ch_r) ← new₁                 (ch_s, ch_r) ← new₁
  fork $ return (consume ch_s)        fork $ return (consume ch_r)
  recv₁ ch_r                          send₁ ch_s ()
```

Where *fork* forks off a new thread using a linear *forkIO*. (In GV, this operation is called *spawn*.)

As the *BlockedIndefinitelyOnMVar* check is performed by the runtime, it'll even happen when a channel is dropped for reasons other than consume, such as a process crashing.

2.2 Session-typed channels

We use the one-shot channels to build a small library of *session-typed channels* based on the *continuation-passing style* encoding of session types in linear types by Dardha et al. [2012, 2017] and in line with other libraries for Scala [Scalas et al. 2017; Scalas and Yoshida 2016], OCaml [Padovani 2017], and Rust [Kokke 2019].

An example. Let's look at a simple example of a session-typed channel—a multiplication service, which receives two integers, sends back their product, and then terminates:

```
type MulServer = Recv Int (Recv Int (Send Int End))
type MulClient = Send Int (Send Int (Recv Int End))
```

We define *mulServer*, which acts on a channel of type *MulServer*, and *mulClient*, which acts on a channel of the *dual* type:

```
mulServer (s :: MulServer)      mulClient (s :: MulClient)
  = do (x, s) ← recv s            = do s ← send (32, s)
       (y, s) ← recv s                 s ← send (41, s)
       s ← send (x * y, s)             (z, s) ← recv s
       close s                         close s
       return ()                       return z
```

In order to encode the *sequence* of a session type using one-shot types, each action on a session-typed channel returns a channel for the *continuation* of the session—save for *close*, which ends the session. This means that the sequence of actions specified by session types becomes a payload of one-shot channels, moving from actions *in breadth* to actions *in depth* (in a *matryoshka doll* style). Furthermore, *mulServer* and *mulClient* act on endpoints with *dual* types. *Duality* is crucial to session types as it ensures that when one process sends, the other is ready to receive, and vice versa. This is the basis for communication safety guaranteed by a session type system.

[2] https://downloads.haskell.org/~ghc/9.0.1/docs/html/libraries/base-4.15.0.0/Control-Exception.html#t:BlockedIndefinitelyOnMVar

Channels. We start by defining the *Session* type class, which has an *associated type* *Dual*. You may think of *Dual* as a type-level function associated with the *Session* class with *one* case for each instance. We encode the various restrictions on duality as constraints on the type class. Each session type must have a dual, which must itself be a session type—*Session* (*Dual* *s*) means the dual of *s* must also implement *Session*. Duality must be *injective*—the annotation $result \rightarrow s$ means *result* must uniquely determine *s* and *involutive*—*Dual* (*Dual* *s*) ∼ *s* means *Dual* (*Dual* *s*) must equal *s*. These constraints are all captured by the *Session* class, along with *new* for constructing channels:

```
class (Session (Dual s), Dual (Dual s) ~ s) ⇒ Session s
  where
    type Dual s = result | result → s
    new :: IO (s, Dual s)
```

There are three primitive session types: *Send*, *Recv*, and *End*.

```
newtype Send a s = Send (Send₁ (a, Dual s))
newtype Recv a s = Recv (Recv₁ (a, s))
newtype End      = End Sync₁
```

By following Dardha et al. [2017], a channel *Send* wraps a one-shot channel $Send_1$ over which we send some value—which is the intended value sent by the session channel, and the channel over which *the communicating partner process* continues the session—it'll make more sense once you read the definition for *send*. A channel *Recv* wraps a one-shot channel $Recv_1$ over which we receive some value and the channel over which *we* continue the session. Finally, an channel *End* wraps a synchronisation.

We define duality for each session type—*Send* is dual to *Recv*, *Recv* is dual to *Send*, and *End* is dual to itself:

```
instance Session s ⇒ Session (Send a s)
  where
    type Dual (Send a s) = Recv a (Dual s)
    new = do (ch_s, ch_r) ← new₁
             return (Send ch_s, Recv ch_r)
instance Session s ⇒ Session (Recv a s)
  where
    type Dual (Recv a s) = Send a (Dual s)
    new = do (ch_s, ch_r) ← new₁
             return (Recv ch_r, Send ch_s)
instance Session End
  where
    type Dual End = End
    new = do (ch_sync1, ch_sync2) ← newSync₁
             return (End ch_sync1, End ch_sync2)
```

The *send* operation constructs a channel for the continuation of the session, then sends one endpoint of that channel, along with the value, over its one-shot channel, and returns the other endpoint:

```
send :: Session s ⇒ (a, Send a s) ⊸ IO s
send (x, Send ch_s) = do (here, there) ← new
                         send₁ ch_s (x, there)
                         return here
```

The *recv* and *close* operations simply wrap their corresponding one-shot operations:

```
recv :: Recv a s ⊸ IO (a, s)
recv (Recv ch_r) = recv₁ ch_r
close :: End ⊸ IO ()
close (End ch_sync) = sync₁ ch_sync
```

Cancellation. We implement session *cancellation* via the *Consumable* class. For convenience, we provide the *cancel* function:

```
cancel :: Session s ⇒ s ⊸ IO
cancel s = return (consume s)
```

As with one-shot channels, *consume* simply drops the channel, and relies on the *BlockedIndefinitelyOnMVar* check, which means that *cancelAndRecv* throws an exception and *cancelAndSend* does not:

```
cancelAndRecv = do              cancelAndSend = do u
  (ch_s, ch_r) ← new              (ch_s, ch_r) ← new
  fork $ cancel ch_s              fork $ cancel ch_r
  ((), ()) ← recv ch_r            () ← send ch_s ()
  return ()                       return ()
```

These semantics correspond to EGV [Fowler et al. 2019].

Asynchronous close. We don't always *want* session-end to involve synchronisation. Unfortunately, the *close* operation is synchronous.

An advantage of defining session types via a type class is that its an *open* class, and we can add new primitives whenever. Let's make the unit type, (), a session type:

```
instance Session s ⇒ Session ()
  where
    type Dual () = ()
    new = return ((), ())
```

Units are naturally affine—they contain *zero* information, so dropping them won't harm—and the linear *Monad* class allows you to silently drop unit results of monadic computations. They're ideal for *asynchronous* session end!

Using () allows us to recover the semantics of one-shot channels while keeping a session-typed language for idiomatic protocol specification.

Choice. So far, we've only presented sending, receiving, and synchronisation. It is, however, possible to send and receive *channels* as well as values, and we leverage that to implement most other session types by using these primitives only!

For instance, we can implement *binary* choice by sending/receiving *Either* of two session continuations:

```
type Select s1 s2 = Send (Either (Dual s1) (Dual s2)) ()
type Offer s1 s2 = Recv (Either s1 s2) ()

selectLeft :: (Session s1) ⇒ Select s1 s2 ⊸ IO s1
selectLeft s = do (here, there) ← new
                  send (Left there, s)
                  return here

offerEither :: Offer s1 s2 ⊸ (Either s1 s2 ⊸ IO a) ⊸ IO a
offerEither s match = do (e, ()) ← recv s; match e
```

Differently from (), we don't have to implement the *Session* class for *Select* and *Offer*. They're already session types!

Recursion. We can write recursive session types by writing them as recursive Haskell types. Unfortunately, we cannot write recursive type synonyms, so we have to use a newtype. For instance, we can write the type for a recursive summation service, which receives numbers until the client indicates they're done, and then sends back the sum. We specify *two* newtypes:

```
newtype SumSrv
  = SumSrv (Offer (Recv Int SumSrv) (Send Int End))
newtype SumCnt
  = SumCnt (Select (Send Int SumCnt) (Recv Int End))
```

We implement the summation server as a recursive function:

```
sumSrv :: Int ⊸ SumSrv˙ ⊸ IO ()
sumSrv tot (SumSrv s) = offerEither s $ λe. case x of
  Left  s → do (x, s) ← recv s; sumSrv (tot + x) s
  Right s → do s ← send (tot, s); close s
```

As *SumSrv* and *SumCnt* are new types, we must provide instances of the *Session* class for them.

```
instance Session SumSrv
  where
    type Dual SumSrv = SumCnt
    new = do (ch_srv, ch_cnt) ← new
             return (SumSrv ch_srv, SumCnt ch_cnt)
```

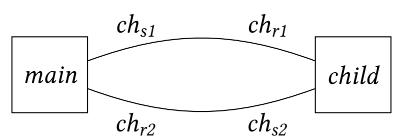
2.3 Deadlock freedom via process structure

The session-typed channels presented in section 2.2 can be used to write deadlocking programs:

```
woops :: IO Void
woops = do (ch_s1, ch_r1) ← new
           (ch_s2, ch_r2) ← new
           fork $ do (void, ()) ← recv ch_r1
                     send (void, ch_s2)
           (void, ()) ← recv ch_r2
           let (void, void_copy) = dup2 void
           send (void, ch_s1)
           return void_copy
```

Counter to what the type says, this program doesn't actually produce an inhabitant of the *uninhabited* type *Void*. Instead, it deadlocks! We'd like to help the programmer avoid such programs.

As discussed in section 1, we can *structurally* guarantee deadlock freedom by ensuring that the *process structure* is always a tree or forest. The process structure of a program is an undirected graph, where nodes represent processes, and edges represent the channels connecting them. For instance, the process structure of *woops* is cyclic:



This restriction works by ensuring that between two processes there is *at most* one (series of) channels over which the two can communicate. As duality rules out deadlocks on any one channel, such configurations must be deadlock free.

We can rule out cyclic process structures by hiding *new*, and only exporting *connect*, which creates a new channel and, *crucially*, immediately passes one endpoint to a new thread:

```
connect :: Session s ⇒
        (s ⊸ IO ()) ⊸ (Dual s ⊸ IO a) ⊸ IO a
connect k1 k2 = do (s1, s2) ← new; fork (k1 s1); k2 s2
```

You can view *connect* as the node constructor for a binary process tree. If the programmer *only* uses *connect*, their process structure is guaranteed to be a *tree*. If they also use standalone *fork*, their process structure is a *forest*. Either way, their programs are guaranteed to be deadlock free.

2.4 Deadlock freedom via priorities

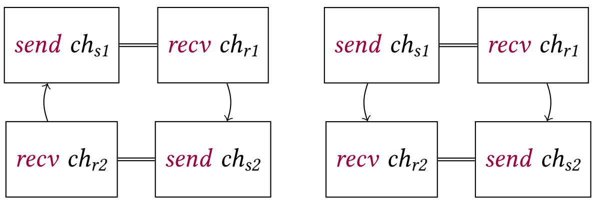
The strategy for deadlock freedom presented in section 2.3 is simple, but *very* restrictive, since it rules out *all* cyclic communication structures, even the ones which don't deadlock:

```
totallyFine :: IO String
totallyFine = do (ch_s1, ch_r1) ← new
                 (ch_s2, ch_r2) ← new
                 fork $ do (x, ()) ← recv ch_r1
                           send (x, ch_s2)
                 send ("Hiya!", ch_s1)
                 (x, ()) ← recv ch_r2
                 return x
```

This process has *exactly the same* process structure as *woops*, but it's totally fine, and returns "Hiya!" as you'd expect.

We'd like to enable the programmer to write such programs while still guaranteeing their programs don't deadlock.

As discussed in section 1, there is another way to rule out deadlocks—by using *priorities*. Priorities are an approximation of the *communication graph* of a program. The communication graph of a program is a *directed graph* where nodes represent *actions on channels*, and directed edges represent that one action happens before the other. Dual actions are connected with double undirected edges. (You may consider the graph contracted along these edges.) If the communication graph is cyclic, the program deadlocks. The communication graphs for *woops* and *totallyFine* are as follows:



woops *totallyFine*

If the communication graph is acyclic, then we can assign each node a number such that directed edges only ever point to nodes with *bigger* numbers. For instance, for *totallyFine* we can assign the number 0 to $\mathit{send}\ ch_{s1}$ and $\mathit{recv}\ ch_{r2}$, and 1 to $\mathit{recv}\ ch_{r2}$ and $\mathit{send}\ ch_{s2}$. These numbers are *priorities*.

In this section, we present a type system in which *priorities* are used to ensure deadlock freedom, by tracking the time a process starts and finishes communicating using a graded monad [Gaboardi et al. 2016; Orchard et al. 2020]. The bind operation registers the order of its actions in the type, requiring the sequentiality of their duals.

Priorities. The priorities assigned to *communication actions* are always natural numbers, which represent, *abstractly*, at which time the action happens. When tracking the start and finish times of a program, however, we also use $\bot$ and $\top$ for programs which don't communicate. These are used as the identities for $\sqcap$ and $\sqcup$ in lower and upper bounds, respectively. We let o range over natural numbers, p over *lower bounds*, and q over *upper bounds*.

$$\textbf{data}\ \mathit{Priority} = \bot \mid \mathit{Nat} \mid \top$$

We define strict inequality ($<$), minimum ($\sqcap$), and maximum ($\sqcup$) on priorities as usual.

Channels. We define Send^o, Recv^o, and End^o, which decorate the *raw* sessions from section 2.2 with the priority o of the communication action, *i.e.*, it denoted when the communication happens. Duality (*Dual*) preserves these priorities. These are implemented exactly as in section 2.2.

The communication monad. We define a graded monad Sesh_p^q, which decorates *IO* with a lower bound p and an upper bound q on the priorities of its communication actions, *i.e.*, if you run the monad, it denotes when communication begins and ends.

$$\textbf{newtype}\ \mathit{Sesh}_p^q\ a = \mathit{Sesh}\ \{\mathit{runSeshIO} :: \mathit{IO}\ a\}$$

The monad operations for Sesh_p^q merely wrap those for *IO*, hence trivially obeys the monad laws.

The *ireturn* function returns a *pure* computation—the type $\mathit{Sesh}_\top^\bot$ guarantees that all communications happen between $\top$ and $\bot$, hence there can be no communication at all.

$$\begin{array}{l} \mathit{ireturn} :: a \multimap \mathit{Sesh}_\top^\bot\ a \\ \mathit{ireturn}\ x = \mathit{Sesh}\ \$\ \mathit{return}\ x \end{array}$$

The $\ggg\!=$ operator sequences two actions with types Sesh_p^q and $\mathit{Sesh}_{p'}^{q'}$, and requires $q < p'$, *i.e.*, the first action must have finished before the second starts. The resulting action has lower bound $p \sqcap p'$ and upper bound $q \sqcup q'$.

$$\begin{array}{l} (\ggg\!=) :: (q < p') \Rightarrow \mathit{Sesh}_p^q\ a \multimap (a \multimap \mathit{Sesh}_{p'}^{q'}\ b) \multimap \mathit{Sesh}_{p \sqcap p'}^{q \sqcup q'}\ b \\ mx \ggg\!= mf = \mathit{Sesh}\ \$\ \mathit{runSeshIO}\ mx \gg\!= \lambda x.\ \mathit{runSeshIO}\ (mf\ x) \end{array}$$

In what follows, we implicitly use $\ggg\!=$ with do-notation. This can be accomplished in Haskell using `RebindableSyntax`.

We define decorated variants of the concurrency and communication primitives: *send*, *recv*, and *close* each perform a communication action with some priority o, and return a computation of type Sesh_o^o, *i.e.*, with *exact* bounds; *new*, *fork*, and *cancel* don't perform any communication action, and so return a *pure* computation of type $\mathit{Sesh}_\top^\bot$.

$$\begin{array}{ll} \mathit{new} & :: \mathit{Session}\ s \Rightarrow \mathit{Sesh}_\top^\bot\ (s, \mathit{Dual}\ s) \\ \mathit{fork} & :: \mathit{Sesh}_p^q\ () \multimap \mathit{Sesh}_\top^\bot\ () \\ \mathit{cancel} & :: \mathit{Session}\ s \Rightarrow s \multimap \mathit{Sesh}_\top^\bot\ () \\ \mathit{send} & :: \mathit{Session}\ s \Rightarrow (a, \mathit{Send}^o\ a\ s) \multimap \mathit{Sesh}_o^o\ s \\ \mathit{recv} & :: \mathit{Recv}^o\ a\ s \multimap \mathit{Sesh}_o^o\ (a, s) \\ \mathit{close} & :: \mathit{End}^o \multimap \mathit{Sesh}_o^o\ () \end{array}$$

From these, we derive decorated choice, as before:

$$\begin{array}{l} \textbf{type}\ \mathit{Select}^o\ s_1\ s_2 = \mathit{Send}^o\ (\mathit{Either}\ (\mathit{Dual}\ s_1)\ (\mathit{Dual}\ s_2))\ () \\ \textbf{type}\ \mathit{Offer}^o\ s_1\ s_2 = \mathit{Recv}^o\ (\mathit{Either}\ s_1\ s_2)\ () \end{array}$$

$$\begin{array}{l} \mathit{selectLeft} :: (\mathit{Session}\ s_1) \Rightarrow \mathit{Select}^o\ s_1\ s_2 \multimap \mathit{Sesh}_o^o\ s_1 \\ \mathit{selectRight} :: (\mathit{Session}\ s_2) \Rightarrow \mathit{Select}^o\ s_1\ s_2 \multimap \mathit{Sesh}_o^o\ s_2 \\ \mathit{offerEither} :: (o < p) \Rightarrow \mathit{Offer}^o\ s_1\ s_2 \multimap \\ \qquad (\mathit{Either}\ s_1\ s_2 \multimap \mathit{Sesh}_p^q\ a) \multimap \mathit{Sesh}_{o \sqcap p}^{o \sqcup q}\ a \end{array}$$

Safe IO. We can use a trick from the *ST* monad [Launchbury and Peyton Jones 1994] to define a "pure" variant of *runSesh*, which encapsulates all use of IO within the Sesh_p^q monad. The idea is to index the Sesh_p^q and every session type constructor with an extra type parameter *tok*, which we'll call the *session token*:

$$\begin{array}{l} \mathit{send} :: \mathit{Session}\ s \Rightarrow (a, \mathit{Send}^o\ \mathit{tok}\ a\ s) \multimap \mathit{Sesh}_o^o\ \mathit{tok}\ s \\ \mathit{recv} :: \mathit{Recv}^o\ \mathit{tok}\ a\ s \multimap \mathit{Sesh}_o^o\ \mathit{tok}\ (a, s) \\ \mathit{close} :: \mathit{End}^o\ \mathit{tok} \multimap \mathit{Sesh}_o^o\ \mathit{tok}\ () \end{array}$$

The session token should never be instantiated, except by *runSesh*, and every action under the same call to *runSesh* should use the same type variable *tok* as its session token:

$$\mathit{runSesh} :: (\forall \mathit{tok}.\ \mathit{Sesh}^q_p\ \mathit{tok}\ a) \multimap a$$
$$\mathit{runSesh}\ x = \mathit{unsafePerformIO}\ (\mathit{runSeshIO}\ x)$$

This ensures that none of the channels created in the session can escape out of the scope of *runSesh*.

We implement this encapsulation in `priority-sesh`, though the session token is the first argument, preceding the priority bounds.

Recursion. We could implement recursive session via priority-polymorphic types, or via priority-shifting [Padovani and Novara 2015]. For instance, we could give the *summation service* from section 2.2 the following type:

$$\begin{aligned}&\mathbf{newtype}\ \mathit{SumSrv}^o\\ &\quad = \mathit{SumSrv}\ (\mathit{Offer}^o\ (\mathit{Recv}^{o+1}\ \mathit{Int}\ (\mathit{SumSrv}^{o+2}))\\ &\qquad\qquad\qquad\quad (\mathit{Send}^{o+1}\ \mathit{Int}\ (\mathit{End}^{o+2})))\end{aligned}$$

We'd then like to assign *sumSrv* the following type:

$$\begin{aligned}&\mathit{sumSrv} : \mathit{Int} \multimap \mathit{SumSrv}^o \multimap \mathit{Sesh}^\top_o\ ()\\ &\mathit{sumSrv}\ \mathit{tot}\ (\mathit{SumSrv}\ s) = \mathit{offerEither}\ s\ \$\ \lambda e.\ \mathbf{case}\ x\ \mathbf{of}\\ &\quad \mathit{Left}\ \ \ s \rightarrow \mathbf{do}\ (x, s) \leftarrow \mathit{recv}\ s; \mathit{sumSrv}\ (\mathit{tot} + x)\ s\\ &\quad \mathit{Right}\ s \rightarrow \mathbf{do}\ s \leftarrow \mathit{send}\ (\mathit{tot}, s); \mathit{weaken}\ (\mathit{close}\ s)\end{aligned}$$

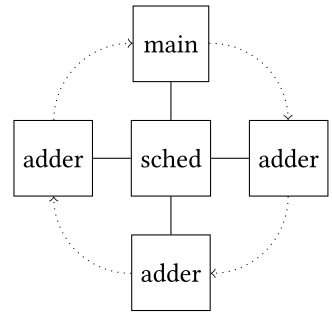
The upper bound for a recursive call should be $\top$, which ensures that recursive calls are only made in *tail* position [Bernardi et al. 2014; Gay et al. 2020]. The recursive call naturally has upper bound $\top$. However, the *close* operation happens at some *concrete* priority $o + n$, which needs to be raised to $\top$, so we'd have to add a primitive $\mathit{weaken} : \mathit{Sesh}^q_p\ a \multimap \mathit{Sesh}^\top_p\ a$.

Unfortunately, writing such priority-polymorphic code relies heavily on GHC's ability to reason about type-level naturals, and GHC rejects *sumSrv* complaining that it cannot verify that $o < o + 1$, $o + 1 < o + 2$, *etc.* There's several possible solutions for this:

1. We could embrace the Hasochism [Lindley and McBride 2013], and provide GHC with explicit evidence, though this would make `priority-sesh` more difficult to use.
2. We could delegate *some* of these problems to a GHC plugin such as `type-nat-solver`[3] or `ghc-typelits-presburger`[4]. Unfortunately, $\sqcap$ and $\sqcup$ are beyond Presburger arithmetic, and `type-nat-solver` has not been maintained in recent years.
3. We could attempt to write type families which reduce in as many cases as possible. Unfortunately, a restriction in closed type families [Eisenberg et al. 2014, §6.1] prevents us from checking *exactly these cases.*

Currently, the prioritised sessions don't support recursion, and implementing one of these solutions is future work.

Cyclic Scheduler. Dardha and Gay [2018] and Kokke and Dardha [[n.d.]] use a *finite* cyclic scheduler as an example. The cyclic scheduler has the following process structure, with the flow of information indicated by the dotted arrows:



We start by defining the types of the channels which connect each client process to the scheduler:

$$\begin{aligned}&\mathbf{type}\ \mathit{SR}^{o_2}_{o_1}\ a = \mathit{Send}^{o1}\ a\ (\mathit{Recv}^{o_2}\ a\ ())\\ &\mathbf{type}\ \mathit{RS}^{o_2}_{o_1}\ a = \mathit{Dual}\ (\mathit{SR}^{o_2}_{o1}\ a)\end{aligned}$$

We then define the scheduler itself, which forwards messages from one process to the next in a cycle:

$$\begin{aligned}&\mathit{sched} :: \mathit{RS}^7_0\ a \multimap \mathit{SR}^2_1\ a \multimap \mathit{SR}^4_3\ a \multimap \mathit{SR}^6_5\ a \multimap \mathit{Sesh}^7_0\ ()\\ &\mathit{sched}\ s1\ s2\ s3\ s4 = \mathbf{do}\\ &\quad (x, s1) \leftarrow \mathit{recv}\ s1\\ &\quad s2 \leftarrow \mathit{send}\ (x, s2); (x, ()) \leftarrow \mathit{recv}\ s2\\ &\quad s3 \leftarrow \mathit{send}\ (x, s3); (x, ()) \leftarrow \mathit{recv}\ s3\\ &\quad s4 \leftarrow \mathit{send}\ (x, s4); (x, ()) \leftarrow \mathit{recv}\ s4\\ &\quad \mathit{send}\ (x, s1)\end{aligned}$$

Finally, we define the *adder* and the *main* processes. The *adder* adds one to the value it receives, and the *main* process initiates the cycle and receives the result:

$$\begin{aligned}&\mathit{adder} :: (o_1 < o_2) \Rightarrow \mathit{RS}^{o_2}_{o_1}\ \mathit{Int} \multimap \mathit{Sesh}^{o_2}_{o_1}\ ()\\ &\mathit{adder}\ s = \mathbf{do}\ (x, s) \leftarrow \mathit{recv}\ s; \mathit{send}\ (x + 1, s)\\ &\mathit{main} :: (o_1 < o_2) \Rightarrow \mathit{Int} \multimap \mathit{SR}^{o_2}_{o_1}\ \mathit{Int} \multimap \mathit{Sesh}^{o_2}_{o_1}\ \mathit{Int}\\ &\mathit{main}\ x\ s = \mathbf{do}\ ; s \leftarrow \mathit{send}\ (x, s); (x, ()) \leftarrow \mathit{recv}\ s; \mathit{ireturn}\ x\end{aligned}$$

While the process structure of the cyclic scheduler *as presented* isn't cyclic, nothing prevents the user from adding communications between the various client processes, or from removing the scheduler and having the client processes communicate *directly* in a ring.

3 Relation to Priority GV

The `priority-sesh` library is based on a variant of Priority GV [Kokke and Dardha [n.d.]], which differs in three ways:

1. it marks lower bounds *explicitly* on the sequent, rather than implicitly inferring them from the typing environment;
2. it collapses the isomorphic types for session end, $\mathbf{end}^o_!$ and $\mathbf{end}^o_?$, into $\mathbf{end}^o$;
3. it is extended with asynchronous communication and session cancellation following Fowler et al. [2019].

These changes preserve subject reduction and progress properties, and give us *tighter* bounds on priorities. To see why, note that PCP [Dardha and Gay 2018] and PGV [Kokke and

[3] https://github.com/yav/type-nat-solver
[4] https://hackage.haskell.org/package/ghc-typelits-presburger

Dardha [n.d.]] use the *smallest* priority in the typing environment as an approximation for the lower bound. Unfortunately, this *underestimates* the lower bound in the rules T-Var and T-Lam (check fig. 1). These rules type *values*, which are pure and could have lower bound $\top$, but the smallest priority in their typing environment is not necessarily $\top$.

Priority GV. We briefly revisit the syntax and type system of PGV, but a full discussion of PGV is out of scope for this paper. For a discussion of the *synchronous* semantics for PGV, and the proofs of subject reduction, progress, and deadlock freedom, please see Kokke and Dardha [[n.d.]]. For a discussion of the *asynchronous* semantics and session cancellation, please see Fowler et al. [2019].

As in section 2.4, we let o range over priorities, which are natural numbers, and p and q over priority bounds, which are either natural numbers, $\top$, or $\bot$.

PGV is based on the standard linear λ-calculus with product types $(\cdot \times \cdot)$, sum types $(\cdot + \cdot)$, and their units ($\mathbf{1}$ and $\mathbf{0}$). Linear functions $(\cdot \multimap^q_p \cdot)$ are annotated with priority bounds which tell us–when the function is applied–when communication begins and ends.

Types and session types are defined as follows:

$$\begin{array}{lcl} S & ::= & !^o T.S \mid ?^o T.S \mid \mathbf{end}^o \\ T, U & ::= & T \times U \mid \mathbf{1} \mid T + U \mid \mathbf{0} \mid T \multimap^q_p U \mid S \end{array}$$

The types $!^o T.S$ and $?^o T.S$ mean "send" and "receive", respectively, and $\mathbf{end}^o$ means, well, session end.

The term language is the standard linear λ-calculus extended with concurrency primitives K:

$$\begin{array}{lcl} L, M, N & & \\ & ::= & x \mid K \mid \lambda x.M \mid M\ N \\ & \mid & () \mid M; N \\ & \mid & (M, N) \mid \mathbf{let}\ (x, y) = M\ \mathbf{in}\ N \\ & \mid & \mathbf{absurd}\ M \\ & \mid & \mathbf{inl}\ M \mid \mathbf{inr}\ M \mid \mathbf{case}\ L\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ y \mapsto N\} \\ K & ::= & \mathbf{new} \mid \mathbf{fork} \mid \mathbf{send} \mid \mathbf{recv} \mid \mathbf{close} \end{array}$$

The concurrency primitives are uninterpreted in the term language. Rather, they are interpreted in a configuration language based on the π-calculus, which we omit from this paper (see Kokke and Dardha [[n.d.]]).

We present the typing rules for PGV in fig. 1. A sequent $\Gamma \vdash^q_p M : T$ should be read as "M is well-typed PGV program with type T in typing environment Γ, and when run it starts communicating at time p and stops at time q."

Monadic Reflection. The graded monad $Sesh^q_p$ arises from the *monadic reflection* [Filinski 1994] of the typing rules in fig. 1. Monadic reflection is a technique for translating programs in an effectful language to *monadic* programs in a pure language. For instance, Filinski [1994] demonstrates the reflection from programs of type T in a language with exceptions and handlers to programs of type $T + \mathbf{exn}$ in a pure language where $\mathbf{exn}$ is the type of exceptions.

We translate programs from PGV to Haskell programs in the $Sesh^q_p$ monad. First, let's look at the translation of types:

$$\begin{array}{lcl@{\quad}lcl} [\![T \multimap^q_p U]\!] & = & [\![T]\!] \multimap Sesh^q_p\ [\![U]\!] & [\![\mathbf{1}]\!] & = & () \\ [\![!^o T.S]\!] & = & Send^o\ [\![T]\!]\ [\![S]\!] & [\![T \times U]\!] & = & ([\![T]\!], [\![U]\!]) \\ [\![?^o T.S]\!] & = & Recv^o\ [\![T]\!]\ [\![S]\!] & [\![\mathbf{0}]\!] & = & Void \\ [\![\mathbf{end}^o]\!] & = & End^o & [\![T + U]\!] & = & Either\ [\![T]\!]\ [\![U]\!] \end{array}$$

Now, let's look at the translation of terms. A term of type T with lower bound p and upper bound q is translated to a Haskell program of type $Sesh^q_p\ [\![T]\!]$:

$$\begin{array}{lcl} [\![x]\!] & = & ireturn\ x \\ [\![\lambda x.L]\!] & = & ireturn\ (\lambda x.\ [\![L]\!]) \\ [\![K]\!] & = & ireturn\ [\![K]\!] \\ [\![L\ M]\!] & = & [\![L]\!] \ggg \lambda f.\ [\![M]\!] \ggg \lambda x. f\ x \\ [\![()]\!] & = & ireturn\ () \\ [\![\mathbf{let}\ () = L\ \mathbf{in}\ M]\!] & = & [\![L]\!] \ggg \lambda().\ M \\ [\![(L, M)]\!] & = & [\![L]\!] \ggg \lambda x.\ [\![M]\!] \ggg \lambda y.\ ireturn\ (x, y) \\ [\![\mathbf{let}\ (x, y) = L\ \mathbf{in}\ M]\!] & = & [\![L]\!] \ggg \lambda(x, y).\ [\![M]\!] \\ [\![\mathbf{absurd}\ L]\!] & = & [\![L]\!] \ggg \lambda x.\ absurd\ x \\ [\![\mathbf{inl}\ L]\!] & = & [\![L]\!] \ggg \lambda x.\ ireturn\ (Left\ x) \\ [\![\mathbf{inr}\ L]\!] & = & [\![L]\!] \ggg \lambda x.\ ireturn\ (Right\ x) \\ \multicolumn{3}{l}{[\![\mathbf{case}\ L\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ y \mapsto N\}]\!] =} \\ \multicolumn{3}{l}{\qquad [\![L]\!] \ggg \lambda x.\ \mathbf{case}\ x\ \mathbf{of}\ \{Left\ x \rightarrow [\![M]\!]; Right\ y \rightarrow [\![N]\!]\}} \end{array}$$

We translate the communication primitives from PGV to those with the same name in `priority-sesh`, with some minor changes in the translations of **new** and **fork**, where PGV needs some unit arguments to create thunks in PGV, as it's call-by-value, which aren't needed in Haskell:

$$\begin{array}{l} [\![\mathbf{new} : \mathbf{1} \multimap^\bot_\top S \times \overline{S}]\!] \\ \quad = \lambda().\ new :: () \multimap ([\![S]\!], [\![(Dual\ S)]\!]) \\ [\![\mathbf{fork} : (\mathbf{1} \multimap^q_p \mathbf{1}) \multimap^\bot_\top \mathbf{1}]\!] \\ \quad = \lambda k.\ fork\ (k\ ()) :: (() \multimap Sesh^q_p\ ()) \multimap Sesh^\bot_\top\ () \end{array}$$

The rest of PGV's communication primitives line up exactly with those of `priority-sesh`:

$$\begin{array}{l} [\![\mathbf{send} : T \times !^o T.S \multimap^o_o S]\!] \\ \quad = send :: Session\ [\![S]\!] \Rightarrow ([\![T]\!], Send^o\ [\![T]\!]\ [\![S]\!]) \multimap Sesh^o_o\ [\![S]\!] \\ [\![\mathbf{recv} : ?^o T.S \multimap^o_o T \times S]\!] \\ \quad = recv :: Recv^o\ [\![T]\!]\ [\![S]\!] \multimap Sesh^o_o\ ([\![T]\!], [\![S]\!]) \\ [\![\mathbf{close} : \mathbf{end}^o \multimap^o_o \mathbf{1}]\!] \\ \quad = close :: End^o \multimap Sesh^o_o\ () \\ [\![\mathbf{cancel} : S \multimap^\bot_\top \mathbf{1}]\!] \\ \quad = cancel :: Session\ [\![S]\!] \Rightarrow [\![S]\!] \multimap Sesh^\bot_\top\ () \end{array}$$

These two translations, on types and terms, comprise a *monadic reflection* from PGV into `priority-sesh`, which preserves typing. We state this theorem formally, using $\Gamma \vdash x :: a$ to mean that the Haskell program x has type a in typing environment Γ:

Theorem 3.1. *If* $\Gamma \vdash^q_p M : T$, *then* $[\![\Gamma]\!] \vdash [\![M]\!] :: Sesh^q_p\ [\![T]\!]$.

Proof. Figure 2 presents the translation from typing derivations in PGV to abbreviated typing derivations in Haskell with `priority-sesh`. □

Static Typing Rules. $\boxed{\Gamma \vdash^{q}_{p} M : T}$

$$\textsc{T-Var}\ \frac{}{x : T \vdash^{\bot}_{\top} x : T} \qquad \textsc{T-Lam}\ \frac{\Gamma, x : T \vdash^{q}_{p} M : U}{\Gamma \vdash^{\bot}_{\top} \lambda x.M : T \multimap^{q}_{p} U} \qquad \textsc{T-Const}\ \frac{}{\varnothing \vdash^{\bot}_{\top} K : T}$$

$$\textsc{T-App}\ \frac{\Gamma \vdash^{q}_{p} M : T \multimap^{q''}_{p''} U \qquad \Delta \vdash^{q'}_{p'} N : T \qquad q < p' \qquad q' < p''}{\Gamma, \Delta \vdash^{q \sqcup q' \sqcup q''}_{p \sqcap p' \sqcap p''} M\ N : U}$$

$$\textsc{T-Unit}\ \frac{}{\varnothing \vdash^{\bot}_{\top} () : \mathbf{1}} \qquad \textsc{T-LetUnit}\ \frac{\Gamma \vdash^{q}_{p} M : \mathbf{1} \qquad \Delta \vdash^{q'}_{p'} N : T \qquad q < p'}{\Gamma, \Delta \vdash^{q \sqcup q'}_{p \sqcap p'} \mathbf{let}\ () = M\ \mathbf{in}\ N : T}$$

$$\textsc{T-Pair}\ \frac{\Gamma \vdash^{q}_{p} M : T \qquad \Delta \vdash^{q'}_{p'} N : U \qquad q < p'}{\Gamma, \Delta \vdash^{q \sqcup q'}_{p \sqcap p'} (M, N) : T \times U}$$

$$\textsc{T-LetPair}\ \frac{\Gamma \vdash^{q}_{p} M : T \times T' \qquad \Delta, x : T, y : T' \vdash^{q'}_{p'} N : U \qquad q < p'}{\Gamma, \Delta \vdash^{q \sqcup q'}_{p \sqcap p'} \mathbf{let}\ (x, y) = M\ \mathbf{in}\ N : U}$$

$$\textsc{T-Inl}\ \frac{\Gamma \vdash^{q}_{p} M : T}{\Gamma \vdash^{q}_{p} \mathbf{inl}\ M : T + U} \qquad \textsc{T-Inr}\ \frac{\Gamma \vdash^{q}_{p} M : T}{\Gamma \vdash^{q}_{p} \mathbf{inr}\ M : T + U}$$

$$\textsc{T-CaseSum}\ \frac{\Gamma \vdash^{q}_{p} L : T + T' \qquad \Delta, x : T \vdash^{q'}_{p'} M : U \qquad \Delta, y : T' \vdash^{q'}_{p'} N : U \qquad q < p'}{\Gamma, \Delta \vdash^{q \sqcup q'}_{p \sqcup p'} \mathbf{case}\ L\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ y \mapsto N\} : U}$$

$$\textsc{T-Absurd}\ \frac{\Gamma \vdash^{q}_{p} M : \mathbf{0}}{\Gamma \vdash^{q}_{p} \mathbf{absurd}\ M : T}$$

Type Schemas for Constants. $\boxed{K : T}$

$$\mathbf{new} : \mathbf{1} \multimap^{\bot}_{\top} S \times \overline{S} \qquad \mathbf{fork} : (\mathbf{1} \multimap^{q}_{p} \mathbf{1}) \multimap^{\bot}_{\top} \mathbf{1} \qquad \mathbf{cancel} : S \multimap^{\bot}_{\top} \mathbf{1}$$

$$\mathbf{send} : T {\times} !^{o} T.S \multimap^{o}_{o} S \qquad \mathbf{recv} : ?^{o} T.S \multimap^{o}_{o} T \times S \qquad \mathbf{close} : \mathbf{end}^{o} \multimap^{o}_{o} \mathbf{1}$$

Figure 1. Typing rules for Priority GV.

4 Related work

Session types in Haskell. Orchard and Yoshida [2017] discuss various approaches to implementing session types in Haskell. Their overview is reproduced below:

- Neubauer and Thiemann [2004] give an encoding of first-order single-channel session-types with recursion;
- Using *parameterised monads*, Pucella and Tov [2008] provide multiple channels, recursion, and some building blocks for delegation, but require manual manipulation of a session typing environment;
- Sackman and Eisenbach [2008] provide an alternate approach where session types are constructed via a value-level witnesses;
- Imai et al. [2010] extend Pucella and Tov [2008] with delegation and a more user-friendly approach to handling multiple channels;
- Orchard and Yoshida [2016] use an embedding of effect systems into Haskell via graded monads based on a formal encoding of session-typed π-calculus into PCF with an effect system;
- Lindley and Morris [2016] provide a *finally tagless* embedding of the GV session-typed functional calculus into Haskell, building on a linear λ-calculus embedding due to Polakow [2015].

With respect to linearity, all works above—except Neubauer and Thiemann [2004]—guarantee linearity by encoding a linear typing environment in the Haskell type system, which leads to a trade-off between having easy-to-write session types and having idiomatic programs. We side-step this trade-off by relying on Linear Haskell to check linearity. Furthermore, our implementation supports all relevant features, including multiple channels, full delegation, recursion, and highly idiomatic code.

With respect to deadlock freedom, none of the works above—except Lindley and Morris [2016]—guarantee deadlock freedom. However, Lindley and Morris [2016] guarantee deadlock freedom *structurally*, by implementing GV. As discussed in section 1, structure-based deadlock freedom is more restrictive than priority-based deadlock freedom, as it restricts communication graphs to *trees*, whereas the priority-based approach allows programs to have *cyclic* process structures.

Orchard and Yoshida [2017] summarise the capabilities of the various implementations of session types in Haskell in a table, which we adapted in fig. 3 by adding columns for the various versions of `priority-sesh`. In general, you may read ✓ as "Kinda" and ✓ as a resounding "Yes!" For instance, Pucella and Tov [2008] only provide *partial* delegation, Neubauer and Thiemann [2004], Pucella and Tov

$$\frac{}{x : T \vdash^{\bot}_{\top} x : T} = \frac{x :: \llbracket T \rrbracket \vdash x :: \llbracket T \rrbracket}{\mathit{ireturn}\ x :: \mathit{Sesh}^{\bot}_{\top}\ \llbracket T \rrbracket}$$

$$\frac{\Gamma, x : T \vdash^{q}_{p} L : U}{\Gamma \vdash^{\bot}_{\top} \lambda x.L : T \multimap^{q}_{p} U} = \frac{\llbracket \Gamma \rrbracket, x :: \llbracket T \rrbracket \vdash \llbracket L \rrbracket :: \mathit{Sesh}^{q}_{p}\ \llbracket U \rrbracket}{\mathit{ireturn}\ (\lambda x.\ \llbracket L \rrbracket) :: \mathit{Sesh}^{\bot}_{\top}\ (\llbracket T \rrbracket \multimap \mathit{Sesh}^{q}_{p}\ \llbracket U \rrbracket)}$$

$$\frac{}{\varnothing \vdash^{\bot}_{\top} K : T} = \frac{}{\mathit{ireturn}\ \llbracket K \rrbracket :: \mathit{Sesh}^{\bot}_{\top}\ \llbracket T \rrbracket}$$

$$\frac{\Gamma \vdash^{q}_{p} L : T \multimap^{q''}_{p''} U \quad \Delta \vdash^{q'}_{p'} M : T \quad q < p' \quad q' < p''}{\Gamma, \Delta \vdash^{q \sqcup q' \sqcup q''}_{p \sqcap p' \sqcap p''} L\ M : U} = \frac{\llbracket \Gamma \rrbracket \vdash \llbracket L \rrbracket :: \mathit{Sesh}^{q}_{p}\ (\llbracket T \rrbracket \multimap \mathit{Sesh}^{q''}_{p''}\ \llbracket U \rrbracket) \quad \llbracket \Delta \rrbracket \vdash \llbracket M \rrbracket :: \mathit{Sesh}^{q'}_{p'}\ \llbracket T \rrbracket}{\llbracket L \rrbracket \ggg= \lambda f.\ \llbracket M \rrbracket \ggg= \lambda x.\ f\ x :: (q < p', q' < p'') \Rightarrow \mathit{Sesh}^{q \sqcup q' \sqcup q''}_{p \sqcap p' \sqcap p''}\ \llbracket U \rrbracket}$$

$$\frac{}{\varnothing \vdash^{\bot}_{\top} () : \mathbf{1}} = \frac{}{\mathit{ireturn}\ () :: \mathit{Sesh}^{\bot}_{\top}\ ()}$$

$$\frac{\Gamma \vdash^{q}_{p} L : \mathbf{1} \quad \Delta \vdash^{q'}_{p'} M : T \quad q < p'}{\Gamma, \Delta \vdash^{q \sqcup q'}_{p \sqcap p'} \mathbf{let}\ () = L\ \mathbf{in}\ M : T} = \frac{\llbracket \Gamma \rrbracket \vdash \llbracket L \rrbracket :: \mathit{Sesh}^{q}_{p}\ () \quad \llbracket \Delta \rrbracket \vdash \llbracket M \rrbracket :: \mathit{Sesh}^{q'}_{p'}\ \llbracket T \rrbracket}{\llbracket L \rrbracket \ggg= \lambda ().\ M :: (p < q') \Rightarrow \mathit{Sesh}^{q \sqcup q'}_{p \sqcap p'}\ \llbracket T \rrbracket}$$

$$\frac{\Gamma \vdash^{q}_{p} L : T \quad \Delta \vdash^{q'}_{p'} M : U \quad q < p'}{\Gamma, \Delta \vdash^{q \sqcup q'}_{p \sqcap p'} (L, M) : T \times U} = \frac{\llbracket \Gamma \rrbracket \vdash \llbracket L \rrbracket :: \mathit{Sesh}^{q}_{p}\ \llbracket T \rrbracket \quad \llbracket \Delta \rrbracket \vdash \llbracket M \rrbracket :: \mathit{Sesh}^{q'}_{p'}\ \llbracket U \rrbracket}{\llbracket L \rrbracket \ggg= \lambda x.\ \llbracket M \rrbracket \ggg= \lambda y.\ \mathit{ireturn}\ (x, y) :: (q < p') \Rightarrow \mathit{Sesh}^{q \sqcup q'}_{p \sqcap p'}\ (\llbracket T \rrbracket, \llbracket U \rrbracket)}$$

$$\frac{\Gamma \vdash^{q}_{p} L : T \times T' \quad \Delta, x : T, y : T' \vdash^{q'}_{p'} M : U \quad q < p'}{\Gamma, \Delta \vdash^{q \sqcup q'}_{p \sqcap p'} \mathbf{let}\ (x, y) = L\ \mathbf{in}\ M : U} = \frac{\llbracket \Gamma \rrbracket \vdash \llbracket L \rrbracket :: \mathit{Sesh}^{q}_{p}\ (\llbracket T \rrbracket, \llbracket T' \rrbracket) \quad \llbracket \Delta \rrbracket, x :: \llbracket T \rrbracket, y :: \llbracket T' \rrbracket \vdash \llbracket M \rrbracket :: \mathit{Sesh}^{q'}_{p'}\ \llbracket U \rrbracket}{\llbracket L \rrbracket \ggg= \lambda (x, y).\ \llbracket M \rrbracket :: (q < p') \Rightarrow \mathit{Sesh}^{q \sqcup q'}_{p \sqcap p'}\ \llbracket U \rrbracket}$$

$$\frac{\Gamma \vdash^{q}_{p} L : T}{\Gamma \vdash^{q}_{p} \mathbf{inl}\ L : T + U} = \frac{\llbracket \Gamma \rrbracket \vdash \llbracket L \rrbracket :: \mathit{Sesh}^{q}_{p}\ \llbracket T \rrbracket}{\llbracket \Gamma \rrbracket \vdash \llbracket L \rrbracket \ggg= \lambda x.\ \mathit{ireturn}\ (\mathit{Left}\ x) :: \mathit{Sesh}^{q}_{p}\ (\mathit{Either}\ \llbracket T \rrbracket\ \llbracket U \rrbracket)}$$

$$\frac{\Gamma \vdash^{q}_{p} L : T}{\Gamma \vdash^{q}_{p} \mathbf{inr}\ L : T + U} = \frac{\llbracket \Gamma \rrbracket \vdash \llbracket L \rrbracket :: \mathit{Sesh}^{q}_{p}\ \llbracket U \rrbracket}{\llbracket \Gamma \rrbracket \vdash \llbracket L \rrbracket \ggg= \lambda x.\ \mathit{ireturn}\ (\mathit{Right}\ x) :: \mathit{Sesh}^{q}_{p}\ (\mathit{Either}\ \llbracket T \rrbracket\ \llbracket U \rrbracket)}$$

$$\frac{\Gamma \vdash^{q}_{p} L : T + T' \quad \Delta, x : T \vdash^{q'}_{p'} M : U \quad \Delta, y : T' \vdash^{q'}_{p'} N : U \quad q < p'}{\Gamma, \Delta \vdash^{q \sqcup q'}_{p \sqcup p'} \mathbf{case}\ L\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ y \mapsto N\} : U} =$$

$$\frac{\llbracket \Gamma \rrbracket \vdash \llbracket L \rrbracket :: \mathit{Sesh}^{q}_{p}\ (\mathit{Either}\ \llbracket T \rrbracket\ \llbracket T' \rrbracket) \quad \llbracket \Delta \rrbracket \vdash x :: \llbracket T \rrbracket \vdash \llbracket M \rrbracket :: \mathit{Sesh}^{q'}_{p'}\ \llbracket U \rrbracket \quad \llbracket \Delta \rrbracket \vdash y :: \llbracket T' \rrbracket \vdash \llbracket N \rrbracket :: \mathit{Sesh}^{q'}_{p'}\ \llbracket U \rrbracket}{\llbracket \Gamma \rrbracket, \llbracket \Delta \rrbracket \vdash \llbracket L \rrbracket \ggg= \lambda x.\ \mathbf{case}\ x\ \{\mathit{Left}\ x \rightarrow \llbracket M \rrbracket; \mathit{Right}\ y \rightarrow \llbracket N \rrbracket\} :: \mathit{Sesh}^{q \sqcup q'}_{p \sqcap p'}\ \llbracket U \rrbracket}$$

$$\frac{\Gamma \vdash^{q}_{p} L : \mathbf{0}}{\Gamma \vdash^{q}_{p} \mathbf{absurd}\ L : T} = \frac{\llbracket \Gamma \rrbracket \vdash \llbracket L \rrbracket :: \mathit{Sesh}^{q}_{p}\ \mathit{Void}}{\llbracket \Gamma \rrbracket \vdash \llbracket L \rrbracket \ggg= \lambda x.\ \mathit{absurd}\ x :: \mathit{Sesh}^{q}_{p}\ \llbracket T \rrbracket}$$

Figure 2. Translation from Priority GV to Sesh preserves types.

[2008], and Lindley and Morris [2016] still need to use combinators instead of standard Haskell application, abstraction, or variables in *some* places, and Neubauer and Thiemann [2004] is only deadlock free on the technicality that they don't support multiple channels.

Session types in other programming languages. Session types have been integrated in other programming language paradigms. Jespersen et al. [2015]; Padovani [2017]; Scalas and Yoshida [2016] integrate *binary* session types in the *native* host language, without language extensions; this

	NT04	PT08	SE08	IYA10	OY16	LM16	priority-sesh section 2.2	section 2.3	section 2.4
Recursion	✓	✓	✓	✓	✓		✓	✓	
Delegation		✓	✓	✓	✓	✓	✓	✓	✓
Multiple channels		✓	✓	✓	✓	✓	✓	✓	✓
Idiomatic code	✓	✓		✓	✓	✓	✓	✓	✓
Easy-to-write session types	✓	✓	✓		✓	✓	✓	✓	✓
Deadlock freedom	✓					✓		✓	✓
via process structure	✓					✓		✓	
via priorities									✓

Figure 3. Capabilities of various implementations of session types in Haskell [adapted from Orchard and Yoshida 2017].

to avoid hindering session types use in practice. To obtain this integration of session types without extensions Padovani [2017]; Scalas and Yoshida [2016]) combine *static* typing of input and output actions with *runtime* checking of linearity of channel usage.

Implementations of *multiparty* session types (MPST) are less common than binary implementations. Scalas et al. [2017] integrate MPST in Scala building upon Scalas and Yoshida [2016] and a continuation-passing style encoding of session types into linear types Dardha et al. [2012]. There are several works on implementations of MPST in Java: Sivaramakrishnan et al. [2010] implement MPST leveraging an extension of Java with session primitives; Hu and Yoshida [2016] develops a MPST-based API generation for Java leveraging CFSMs by Brand and Zafiropulo [1983]; and Kouzapas et al. [2016] implement session types in the form of *typestates* in Java. Demangeon et al. [2015] implement MPST in Python and Fowler [2016]; Neykova and Yoshida [2017a] in Erlang, focusing on purely dynamic MPST verification via runtime monitoring. Neykova et al. [2017]; Neykova and Yoshida [2017b] extend the work by Demangeon et al. [2015] with actors and timed specifications. Lopez et al. [2015] adopt a dependently-typed MPST theory to verify MPI programs.

Session types, linear logic and deadlock freedom. The main line of work regarding deadlock freedom in session-typed systems is that of Curry-Howard correspondences with linear logic [Girard 1987]. Caires and Pfenning [2010] defined a correspondence between session types and dual intuitionistic linear logic and Wadler [2014] between session types and classical linear logic. These works guarantee deadlock freedom *by design* as the communication structures are restricted to trees and due to the *cut* rule, processes share *only* one channel between them. Dardha and Gay [2018] extend Wadler [2014] with *priorities* following Kobayashi [2006]; Padovani [2014], thus allowing processes to share more than one channel in parallel, while guaranteeing deadlock freedom. Balzer et al. [2019] introduce *sharing* and guarantee deadlock freedom via priorities. All the above works deal with deadlock freedom in a session-typed π-calculus. With regards to function languages, the original works on GV [Gay and Vasconcelos 2010, 2012] did not guarantee deadlock freedom. This was later addressed by Lindley and Morris [2015]; Wadler [2015] via syntactic restrictions where communication once again follows a tree structure. Kokke and Dardha [[n.d.]] introduce PGV–Priority GV, by following Dardha and Gay [2018] and allowing for more flexible programming in GV.

Other works on deadlock freedom in session-typed systems include the works by Dezani-Ciancaglini et al. [2006], where deadlock freedom is guaranteed by allowing only one active session at a time and by Dezani-Ciancaglini et al. [2009], where priorities are used for correct interleaving of channels. Honda et al. [2008] guarantee deadlock freedom *within a single* session of MPST, but not for session interleaving. Kokke [2019] guarantees deadlock freedom of session types in Rust by enforcing a tree structure of communication actions.

5 Discussion and future work

We presented `priority-sesh`, an implementation of deadlock-free session types in Linear Haskell. Using Linear Haskell allows us to check linearity—or more accurately, have linearity guaranteed for us—without relying on complex type-level machinery. Consequently, we have easy-to-write session types and idiomatic code—in fact, probably *the most* idiomatic code when compared with previous work, though in fairness, all previous work predates Linear Haskell. Unfortunately, there are some drawbacks to using Linear Haskell. Most importantly, Linear Haskell is not very mature at this stage. For instance:

- Anonymous functions are assumed to be unrestricted rather than linear, meaning anonymous functions must be factored out into a let-binding or where-clause with *at least* a minimal type signature such as $\_ \multimap \_$.
- There is no integration with `base` or popular Haskell packages, and given that `LinearTypes` is an extension, there likely won't be for quite a while. There's `linear-base`, which provides linear variants of many

of the constructs in `base`. However, `linear-base` relies heavily on `unsafeCoerce`, which, *ironically*, may affect Haskell's performance.
- Generally, there is little integration with the Haskell ecosystem, *e.g.*, one other contribution we made are the formatting directives for Linear Haskell in lhs2TEX[5].

However, we believe that many of these drawbacks will disappear as the Linear Haskell ecosystem matures.

Our work also provides a library which guarantees deadlock freedom via *priorities*, which allows for more flexible typing than previous work on deadlock freedom via a *tree process structure*.

In the future, we plan to address the issue of priority-polymorphic code and recursion session types in our implementation. (While the versions of our library in sections 2.2 and 2.3 support recursion, that is not yet the case for the priority-based version in section 2.4.) This is a challenging task, as it requires complex reasoning about type-level naturals. We outlined various approaches in section 2.4. However, an alternative we would like to investigate, would be to implement `priority-sesh` in Idris2 [Brady 2013, 2017], which supports *both* linear types *and* complex type-level reasoning.

References

Stephanie Balzer, Bernardo Toninho, and Frank Pfenning. 2019. Manifest Deadlock-Freedom for Shared Session Types. In *Proc. of ESOP (Lecture Notes in Computer Science, Vol. 11423)*. Springer, 611–639.

Giovanni Bernardi, Ornela Dardha, Simon J. Gay, and Dimitrios Kouzapas. 2014. On Duality Relations for Session Types. In *Trustworthy Global Computing*. Springer Berlin Heidelberg, 51–66. https://doi.org/10.1007/978-3-662-45917-1_4

Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2018. Linear Haskell: practical linearity in a higher-order polymorphic language. *Proc. of POPL* 2 (2018), 1–29. https://doi.org/10.1145/3158093

Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 5 (2013), 552–593. https://doi.org/10.1017/S095679681300018X

Edwin Brady. 2017. TYPE-DRIVEN DEVELOPMENT OF CONCURRENT COMMUNICATING SYSTEMS. *Computer Science* 18, 3 (2017), 219. https://doi.org/10.7494/csci.2017.18.3.1413

Daniel Brand and Pitro Zafiropulo. 1983. On Communicating Finite-State Machines. *J. ACM* 30, 2 (April 1983), 323–342. https://doi.org/10.1145/322374.322380

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

Ornela Dardha and Simon J. Gay. 2018. A New Linear Logic for Deadlock-Free Session-Typed Processes. In *Proc. of FoSSaCS (LNCS, Vol. 10803)*. Springer, 91–109.

Ornela Dardha, Elena Giachino, and Davide Sangiorgi. 2012. Session types revisited. In *Proc. of PPDP*. ACM, 139–150.

Ornela Dardha, Elena Giachino, and Davide Sangiorgi. 2017. Session types revisited. *Inf. Comput.* 256 (2017), 253–286. https://doi.org/10.1016/j.ic.2017.06.002

Romain Demangeon, Kohei Honda, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. 2015. Practical Interruptible Conversations: Distributed Dynamic Verification with Multiparty Session Types and Python. *Formal Methods in System Design* (2015). https://doi.org/10.1007/s10703-014-0218-8

Mariangiola Dezani-Ciancaglini, Ugo de'Liguoro, and Nobuko Yoshida. 2009. On Progress for Structured Communications. In *Proc. of TGC (LNCS, Vol. 4912)*. Springer, 257–275.

Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. 2006. Session Types for Object-Oriented Languages. In *Proc. of ECOOP (LNCS, Vol. 4067)*. Springer, 328–352.

Richard A. Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. 2014. Closed type families with overlapping equations. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM. https://doi.org/10.1145/2535838.2535856

Andrzej Filinski. 1994. Representing monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '94*. ACM Press. https://doi.org/10.1145/174675.178047

Simon Fowler. 2016. An Erlang Implementation of Multiparty Session Actors. In *ICE*. https://doi.org/10.4204/EPTCS.223.3

Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. 2019. Exceptional Asynchronous Session Types: Session Types without Tiers. *Proc. of POPL* 3, Article 28 (2019), 29 pages. https://doi.org/10.1145/3290341

Marco Gaboardi, Shin ya Katsumata, Dominic Orchard, Flavien Breuvart, and Tarmo Uustalu. 2016. Combining effects and coeffects via grading. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. ACM. https://doi.org/10.1145/2951913.2951939

Simon J. Gay, Peter Thiemann, and Vasco T. Vasconcelos. 2020. Duality of Session Types: The Final Cut. *Electronic Proceedings in Theoretical Computer Science* 314 (April 2020), 23–33. https://doi.org/10.4204/eptcs.314.3

Simon J. Gay and Vasco T. Vasconcelos. 2010. Linear type theory for asynchronous session types. *Journal of Functional Programming* 20, 1 (2010), 19–50.

Simon J. Gay and Vasco T. Vasconcelos. 2012. Linear type theory for asynchronous session types. *JFP* 20, 1 (2012), 19–50. Extended version of [Gay and Vasconcelos 2010].

Jean-Yves Girard. 1987. Linear Logic. *Theoretical Computer Science* 50 (1987), 1–102.

Kohei Honda. 1993. Types for Dyadic Interaction. In *Proc. of CONCUR (LNCS, Vol. 715)*. Springer, 509–523.

Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. 1998. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *Proc. of ESOP (LNCS, Vol. 1381)*. Springer, 122–138.

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty asynchronous session types. In *Proc. of POPL*, Vol. 43(1). ACM, 273–284.

Raymond Hu and Nobuko Yoshida. 2016. Hybrid Session Verification Through Endpoint API Generation. In *Proc. of FASE*. https://doi.org/10.1007/978-3-662-49665-7_24

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

Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. 2015. Session types for Rust. In *Proc. of WGP@ICFP*. https://doi.org/10.1145/2808098.2808100

Naoki Kobayashi. 2006. A New Type System for Deadlock-Free Processes. In *Proc. of CONCUR (LNCS, Vol. 4137)*. Springer, 233–247.

Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. 1999. Linearity and the pi-calculus. *ACM Trans. Program. Lang. Syst.* 21, 5 (1999), 914–947. https://doi.org/10.1145/330249.330251

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

[5] https://hackage.haskell.org/package/lhs2tex

Renamed to Sesh.

Wen Kokke and Ornela Dardha. [n.d.]. Prioritise the Best Variation. ([n. d.]). https://wen.works/public/drafts/pgv.pdf Forthcoming.

Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. 2016. Typechecking protocols with Mungo and StMungo. In *PPDP*. 146–159.

John Launchbury and Simon L. Peyton Jones. 1994. Lazy Functional State Threads. In *Proc. of PLDI* (Orlando, Florida, USA). ACM, New York, NY, USA, 24–35. https://doi.org/10.1145/178243.178246

Sam Lindley and Conor McBride. 2013. Hasochism. In *Proceedings of the 2013 ACM SIGPLAN symposium on Haskell - Haskell '13*. ACM Press. https://doi.org/10.1145/2503778.2503786

Sam Lindley and J. Garrett Morris. 2015. A Semantics for Propositions as Sessions. In *Proc. of ESOP*. 560–584.

Sam Lindley and J. Garrett Morris. 2016. Embedding session types in Haskell. In *Proc. of Haskell*. ACM, 133–145. https://doi.org/10.1145/2976002.2976018

Sam Lindley and J Garrett Morris. 2017. Lightweight Functional Session Types. In *Behavioural Types: from Theory to Tools*. River Publishers, 265–286.

Hugo A. Lopez, Eduardo R. B. Marques, Francisco Martins, Nicholas Ng, Casar Santos, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. 2015. Protocol-Based Verification of Message-Passing Parallel Programs. In *OOPSLA*. https://doi.org/10.1145/2814270.2814302

Matthias Neubauer and Peter Thiemann. 2004. An Implementation of Session Types. In *Proc. of PADL (Lecture Notes in Computer Science, Vol. 3057)*. Springer, 56–70. https://doi.org/10.1007/978-3-540-24836-1_5

Rumyana Neykova, Laura Bocchi, and Nobuko Yoshida. 2017. Timed Runtime Monitoring for Multiparty Conversations. *Formal Aspects of Computing* (2017). https://doi.org/10.1007/s00165-017-0420-8

Rumyana Neykova and Nobuko Yoshida. 2017a. Let It Recover: Multiparty Protocol-Induced Recovery. In *CC*. https://doi.org/10.1145/3033019.3033031

Rumyana Neykova and Nobuko Yoshida. 2017b. Multiparty Session Actors. *Logical Methods in Computer Science* 13, 1 (March 2017). https://doi.org/10.23638/LMCS-13(1:17)2017

Dominic Orchard, Philip Wadler, and Harley Eades. 2020. Unifying graded and parameterised monads. *Electronic Proceedings in Theoretical Computer Science* 317 (May 2020), 18–38. https://doi.org/10.4204/eptcs.317.2

Dominic Orchard and Nobuko Yoshida. 2017. Session types with linearity in Haskell. *Behavioural Types: from Theory to Tools* (2017), 219.

Dominic A. Orchard and Nobuko Yoshida. 2016. Effects as sessions, sessions as effects. In *Proc. of POPL*. ACM, 568–581. https://doi.org/10.1145/2837614.2837634

Luca Padovani. 2014. Deadlock and Lock Freedom in the Linear π-Calculus. In *Proc. of CSL-LICS*. ACM, 72:1–72:10.

Luca Padovani. 2017. A simple library implementation of binary sessions. *Journal of Functional Programming* 27 (2017). https://doi.org/10.1017/S0956796816000289 Website: http://www.di.unito.it/~padovani/Software/FuSe/FuSe.html.

Luca Padovani and Luca Novara. 2015. Types for Deadlock-Free Higher-Order Programs. In *Proc. of FORTE (LNCS, Vol. 9039)*. Springer, 3–18.

Simon L. Peyton Jones, Andrew D. Gordon, and Sigbjørn Finne. [n.d.]. Concurrent Haskell. In *Proc. of POPL*. ACM, 295–308. https://doi.org/10.1145/237721.237794

Jeff Polakow. 2015. Embedding a full linear Lambda calculus in Haskell. In *Proc/ of the Symposium on Haskell*. ACM. https://doi.org/10.1145/2804302.2804309

Riccardo Pucella and Jesse A. Tov. 2008. Haskell session types with (almost) no class. In *Proc. of Haskell*. ACM. https://doi.org/10.1145/1411286.1411290

Matthew Sackman and Susan Eisenbach. 2008. Session Types in Haskell Updating Message Passing for the 21st Century. (01 2008).

Davide Sangiorgi and David Walker. 2001. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press.

Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. 2017. A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming. In *Proc. of ECOOP (LIPIcs, Vol. 74)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 24:1–24:31. https://doi.org/10.4230/LIPIcs.ECOOP.2017.24

Alceste Scalas and Nobuko Yoshida. 2016. Lightweight Session Programming in Scala. In *ECOOP*. https://doi.org/10.4230/LIPIcs.ECOOP.2016.21

K. C. Sivaramakrishnan, Karthik Nagaraj, Lukasz Ziarek, and Patrick Eugster. 2010. Efficient Session Type Guided Distributed Interaction. In *Proc. of COORDINATION*, Vol. 6116. https://doi.org/10.1007/978-3-642-13414-2_11

Kaku Takeuchi, Kohei Honda, and Makoto Kubo. 1994. An Interaction-Based Language and its Typing System. In *Proc. of PARLE (LNCS, Vol. 817)*. Springer, 398–413.

Philip Wadler. 2012. Propositions as sessions. In *Proc. of ICFP*. 273–286.

Philip Wadler. 2014. Propositions as sessions. *Journal of Functional Programming* 24, 2-3 (Jan. 2014), 384–418. Extended version of [Wadler 2012].

Philip Wadler. 2015. Propositions as types. *Commun. ACM* 58, 12 (2015), 75–84.