
There may be differences between this version and the published version. You are advised to consult the publisher’s version if you wish to cite from it.

© The Authors 2021. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in the Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell (Haskell 2021), 26-27 Aug 2021, ISBN 9781450386159 (doi:10.1145/3471874.3472979)

http://eprints.gla.ac.uk/249058/

Deposited on: 06 August 2021

Enlighten – Research publications by members of the University of Glasgow
http://eprints.gla.ac.uk
Deadlock-Free Session Types in Linear Haskell

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

Ornella 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 composable, 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 more 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

1 Introduction
Session types are a type formalism used to specify and verify communication protocols [26–28, 62]. They’ve been studied extensively in the context of the π-calculus [58], a process calculus for communication and concurrency, and in the context of concurrent λ-calculi, such as the GV family of languages ["Good Variation", 20, 23, 40, 64].

Session types have been implemented in various programming languages. We give a detailed overview in section 4, and Orchard and Yoshida [49] 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 [52, 59, 60] and static [41, 42, 56] 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: Neuhauer and Thiemann [44] only provide single session channels; Pucella and Tov [56] provide multiple channels, but only the building blocks for channel delegation; Imai et al. [30] extend Pucella and Tov [56] and provide full delegation. None of these works address deadlock freedom. Lindley and Morris [41] provide an implementation of GV into Haskell building on the work of Polakow [55]. 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 [41] 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 [53] and Kokke and Dardha [PGV, 35] integrate priorities [32, 51] 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 [4], 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 guarantees 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 more 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 [35]—the calculus upon which Priority Sesh is based—with asynchronous communication and session cancellation following Fowler et al. [20] 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 MVar [54];
- in section 2.2, we use these one-shot channels to build a small library of session-typed channels [12]; and
- in section 2.4, we decorate these session types with priorities to guarantee deadlock-freedom [35].

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 [33, 58], 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 [4]. The type → is syntactic sugar for the linear arrow %→%. Familiar definitions refer to linear variants packaged with linear-base¹ (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, Send₁ and Recv₁, which are two copies of the same MVar.

newtype Send₁ a = Send₁ (MVar a)
newtype Recv₁ a = Recv₁ (MVar a)
new₁ :: IO (Send₁ a, Recv₁ a)
new₁ = do (mvarₓ, mvarₓ) ← dup2 ($) newEmptyMVar
        return (Send₁ (unur mvarₓ), Recv₁ (unur mvarₓ))

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 send₁ and recv₁ as aliases for the corresponding MVar operations.

send₁ :: Send₁ a → a → IO ()
send₁ (Send₁ mvarₓ) x = putMVar mvarₓ x
recv₁ :: Recv₁ a → a
recv₁ (Recv₁ mvarₓ) = takeMVar mvarₓ

The MVar operations implement the correct blocking behaviour for asynchronous one-shot channels: the send₁ operation is non-blocking, and the recv₁ operations blocks until a value becomes available.

Synchronisation. We use Send₁ and Recv₁ to implement a construct for one-shot synchronisation between two processes, Sync₁, which consists of two one-shot channels. To

¹https://hackage.haskell.org/package/linear-base
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 (ch₁, ch₂) ← new₁
           (ch₃, ch₄) ← new₁
           return (Sync₁ ch₁ ch₃, Sync₁ ch₂ ch₄)

sync₁ :: Sync₁ → IO ()
sync₁ (Sync₁ ch₁ ch₃) = do send₁ ch₁ (); recv₁ ch₃

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 → (). 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 MVars. 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.” Practically, consumeAndRecv throws a BlockedIndefinitelyOnMVar exception, whereas consumeAndSend does not:

consumeAndRecv = do consumeAndSend = do u
                   (ch₁, ch₃) ← new₁ (ch₁, ch₃) ← new₁
                   fork $ return (consume ch₃) fork $ return (consume ch₃)
                   recv₁ ch₃ send₁ ch₃ ()

Where fork forks off a new thread using a linear forklO. (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 [9], Dardha et al. [12] and in line with other libraries for Scala [59, 60], OCaml [52], and Rust [34].

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 (y, s) ← recv s (z, s) ← recv s
 unfold 2 (y ∗ z) s ← send s (s, y) ← send 32 s
 close s close s 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. 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.

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 → 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. [12], a channel Send wraps a one-shot channel Send₁ 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 Recv₁ 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:

2https://downloads.haskell.org/~ghc/9.0.1/docs/html/libraries/base-4.15.0.0/Control-Exception.html#t:BlockedIndefinitelyOnMVar
instance \(\text{Session} \ s \Rightarrow \text{Session} \ (\text{Send} \ a \ s)\)
where
\[
\begin{align*}
\text{type } \text{Dual} \ (\text{Send} \ a \ s) &= \text{Recv} \ a \ (\text{Dual} \ s) \\
\text{new} &= \text{do} \ (ch, ch_r) \leftarrow \text{new}_1 \\
& \quad \text{return} \ (\text{Send} \ ch, \ \text{Recv} \ ch_r)
\end{align*}
\]
instance \(\text{Session} \ s \Rightarrow \text{Session} \ (\text{Recv} \ a \ s)\)
where
\[
\begin{align*}
\text{type } \text{Dual} \ (\text{Recv} \ a \ s) &= \text{Send} \ a \ (\text{Dual} \ s) \\
\text{new} &= \text{do} \ (ch, ch_r) \leftarrow \text{new}_1 \\
& \quad \text{return} \ (\text{Recv} \ ch_r, \ \text{Send} \ ch)
\end{align*}
\]
instance \(\text{Session} \ \text{End}\)
where
\[
\begin{align*}
\text{type } \text{Dual} \ \text{End} &= \ \text{End}\ \\
\text{new} &= \text{do} \ (ch_{\text{sync}1}, ch_{\text{sync}2}) \leftarrow \text{newSync}_1 \\
& \quad \text{return} \ (\text{End} \ ch_{\text{sync}1}, \ \text{End} \ ch_{\text{sync}2})
\end{align*}
\]

The \text{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:
\[
\text{send} :: \text{Session} \ s \Rightarrow (a, \text{Send} \ a \ s) \rightarrow \text{IO} \ s \\
\text{send} \ (x, \text{Send} \ ch_i) = \text{do} \ (\text{here}, \ \text{there}) \leftarrow \text{new} \ \\
& \quad \text{send}_1 \ ch_i \ (x, \ \text{there}) \quad \text{return} \ \text{here}
\]

The \text{recv} and \text{close} operations simply wrap their corresponding one-shot operations:
\[
\begin{align*}
\text{recv} :: \text{Recv} \ a \ s \rightarrow \text{IO} \ (a, \ s) \\
\text{recv} \ (\text{Recv} \ ch_i) &= \text{recv}_1 \ ch_i \\
\text{close} :: \text{End} \rightarrow \text{IO} \ () \\
\text{close} \ (\text{End} \ ch_{\text{sync}}) &= \text{sync}_1 \ ch_{\text{sync}}
\end{align*}
\]

**Cancellation.** We implement session cancellation via the \text{Consumable} class. For convenience, we provide the \text{cancel} function:
\[
\text{cancel} :: \text{Session} \ s \Rightarrow s \rightarrow \text{IO} \ () \\
\text{cancel} \ s = \text{return} \ \text{consume} \ s
\]

As with one-shot channels, \text{consume} simply drops the channel, and relies on the \text{BlockedIndefinitelyOnMVar} check, which means that \text{cancelAndRecv} throws an exception and \text{cancelAndSend} does not:
\[
\begin{align*}
\text{cancelAndRecv} &= \text{do} \\
& \quad (ch_i, ch_r) \leftarrow \text{new} \\
& \quad \text{fork} \$ \text{cancel} \ ch_i \\
& \quad ((\cdot), ()) \leftarrow \text{recv} \ ch_i \\
& \quad \text{return} \ ()
\end{align*}
\]
\[
\begin{align*}
\text{cancelAndSend} &= \text{do} \ u \\
& \quad (ch_i, ch_r) \leftarrow \text{new} \\
& \quad \text{fork} \$ \text{cancel} \ ch_i \\
& \quad () \leftarrow \text{send} \ ch_i \\
& \quad \text{return} \ ()
\end{align*}
\]

These semantics correspond to EGV [20].

**Asynchronous Close.** We don’t always want session-end to involve synchronisation. Unfortunately, the \text{close} operation is synchronous.

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

instance \(\text{Session} \ s \Rightarrow \text{Session} \ ()\)
where
\[
\begin{align*}
\text{type } \text{Dual} \ () &= () \\
\text{new} &= \text{return} \ (((), ()))
\end{align*}
\]

Units are naturally affine—they contain zero information, so dropping them won’t harm—and the linear \text{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 \text{Either} of two session continuations:
\[
\begin{align*}
\text{type } \text{Select} \ s_1 \ s_2 &= \text{Send} \ (\text{Either} \ (\text{Dual} \ s_1) \ (\text{Dual} \ s_2)) () \\
\text{type } \text{Offer} \ s_1 \ s_2 &= \text{Recv} \ (\text{Either} \ s_1 \ s_2) () \\
\text{selectLeft} :: (\text{Session} \ s_1) \Rightarrow \text{Select} \ s_1 \ s_2 \rightarrow \text{IO} \ s_1 \\
\text{selectLeft} \ s = \text{do} \ (\text{here}, \ \text{there}) \leftarrow \text{new} \\
& \quad \text{send} \ (\text{Left} \ \text{there}, \ s) \\
& \quad \text{return} \ \text{here}
\end{align*}
\]

\[
\begin{align*}
\text{offerEither} :: \text{Offer} \ s_1 \ s_2 \rightarrow (\text{Either} \ s_1 \ s_2 \rightarrow \text{IO} \ a) \rightarrow \text{IO} \ a \\
\text{offerEither} \ s \ \text{match} = \text{do} \ (e, ()) \leftarrow \text{recv} \ s \ \text{match} \ e
\end{align*}
\]

Differently from (), we don’t have to implement the \text{Session} class for \text{Select} and \text{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 \text{SumSrv} = \text{SumSrv} \ (\text{Offer} \ (\text{Recv} \ \text{Int} \ \text{SumSrv}) \ (\text{Send} \ \text{Int} \ \text{End}))
newtype \text{SumCnt} = \text{SumCnt} \ (\text{Select} \ (\text{Send} \ \text{Int} \ \text{SumCnt}) \ (\text{Recv} \ \text{Int} \ \text{End}))

We implement the summation server as a recursive function:
\[
\text{sumSrv} :: \text{Int} \rightarrow \text{SumSrv} \rightarrow \text{IO} \ () \\
\text{sumSrv} \ \text{tot} \ (\text{SumSrv} \ s) = \text{offerEither} \ s \$ \lambda \cdot \text{case} \ x \ \text{of}
\]
Left \( s \rightarrow \text{do} \ (x, s) \leftarrow \text{recv} \ s; \text{sumSrv} \ (\text{tot} + x) \ s \)

Right \( s \rightarrow \text{do} \ s \leftarrow \text{send} \ (\text{tot}, s); \text{close} \ s \)

As \text{sumSrv} and \text{sumCnt} are new types, we must provide instances of the \text{Session} class for them.

instance \text{Session} \text{sumSrv}
where
\[
\text{type} \ \text{Dual \ sumSrv} = \text{sumCnt} \\
\text{new} = \text{do} \ (ch_{\text{srv}}, ch_{\text{mt}}) \leftarrow \text{new} \\
\text{return} \ (\text{sumSrv} \ ch_{\text{srv}}, \text{sumCnt} \ ch_{\text{mt}})
\]

2.3 Deadlock Freedom via Process Structure

The session-typed channels presented in section 2.2 can be used to write deadlock-free programs, e.g., by receiving before sending:

\text{woops} :: \text{IO} \ \text{Void}

\text{woops} = \text{do} \ (ch_{\text{in}}, ch_{\text{r1}}) \leftarrow \text{new} \\
(ch_{\text{in}}, ch_{\text{r2}}) \leftarrow \text{new} \\
\text{fork $\text{do}$} \ (\text{void}, ()) \leftarrow \text{recv} \ ch_{\text{r1}} \\
\text{send} \ (\text{void}, ch_{\text{r2}}) \\
(\text{void}, ()) \leftarrow \text{recv} \ ch_{\text{r2}} \\
\text{let} \ (\text{void}, \text{void}_{\text{copy}}) = \text{dup2} \ \text{void} \\
\text{send} \ (\text{void}, ch_{\text{r1}}) \\
\text{return} \ \text{void}_{\text{copy}}

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

As discussed in section 1, we can \textit{structurally} guarantee deadlock freedom by ensuring that the \textit{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 \text{woops} is cyclic:

```
+-----+        +-----+        +-----+        +-----+        +-----+
|     |        |     |        |     |        |     |        |     |
| main|        |     |        | child|        |     |        |     |
|     |        |     |        |     |        |     |        |     |
| ch_{r1} |      |     |        |     |        |     |        |     |
|     |        |     |        |     |        |     |        |     |
| ch_{in} |      | ch_{r1} |      |     |     |      | ch_{r2} |      |     |
|     |        |     |        |     |        |     |        |     |
| ch_{r2} |      |     |        |     |        |     |        |     |
|     |        |     |        |     |        |     |        |     |
```

This restriction works by ensuring that between two processes there is \textit{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 \textit{new}, and only exporting \textit{connect}, which creates a new channel and, \textit{crucially}, immediately passes one endpoint to a new thread:

\text{connect :: Session} \ s \Rightarrow
( s \rightarrow \text{IO} \ ()) \rightarrow (\text{Dual} \ s \rightarrow \text{IO} \ a) \rightarrow \text{IO} \ a

connect \ k_1 \ k_2 = \text{do} \ (s_1, s_2) \leftarrow \text{new; fork} \ (k_1 \ s_1); k_2 \ s_2

You can view \text{connect} as the node constructor for a binary process tree. If the programmer only uses \text{connect}, their process structure is guaranteed to be a tree. If they also use standalone \text{fork}, their process structure is a \textit{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:

\text{totallyFine} :: \text{IO} \ \text{String}

\text{totallyFine} = \text{do} \ (ch_{\text{r1}}, ch_{\text{r2}}) \leftarrow \text{new} \\
(ch_{\text{s1}}, ch_{\text{s2}}) \leftarrow \text{new} \\
\text{fork $\text{do}$} \ (x, ()) \leftarrow \text{recv} \ ch_{\text{r1}} \\
\text{send} \ (x, ch_{\text{s2}}) \\
\text{send} \ (\text{"Hiya!"}, ch_{\text{r2}}) \\
(x, ()) \leftarrow \text{recv} \ ch_{\text{r2}} \\
\text{return} \ x

This process has \text{exactly the same} process structure as \text{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 \textit{priorities}. Priorities are an approximation of the \textit{communication graph} of a program. The communication graph of a program is a \textit{directed graph} where nodes represent \textit{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 \text{woops} and \text{totallyFine} are as follows:

```
ch_{r1} -- ch_{s1} -- ch_{r2} -- ch_{s2} -- ch_{r1}
```

If the communication graph is acyclic, then we can assign each node a number such that directed edges only ever point to nodes with \textit{bigger} numbers. For instance, for \text{totallyFine} we can assign the number 0 to \text{send} \ ch_{r1} and \text{recv} \ ch_{r2}, and 1 to \text{recv} \ ch_{r2} and \text{send} \ ch_{r2}. These numbers are \textit{priorities}.

In this section, we present a type system in which \textit{priorities} are used to ensure deadlock freedom, by tracking the time a process starts and finishes communicating using a graded monad [21, 48]. The bind operation registers the order of its actions in the type, requiring the sequentiality of their duals.

\textbf{Priorities.} The priorities assigned to \textit{communication actions} are always natural numbers, which represent, \textit{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 $\land$ and $\lor$ in lower and upper bounds, respectively. We let $\alpha$ range over natural numbers, $\beta$ over lower bounds, and $\gamma$ over upper bounds.

data Priority = $\bot$ | Nat | $\top$

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

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

**The Communication Monad.** We define a graded monad $\text{Sesh}_\beta \varphi$, which decorates IO with a lower bound $\beta$ and an upper bound $\varphi$ on the priorities of its communication actions, i.e., if you run the monad, it denotes when communication begins and ends.

newtype $\text{Sesh}_\beta \varphi \ a = \text{Sesh} \{ \text{runSeshIO} :: \text{IO} \ a \}$

The monad operations for $\text{Sesh}_\beta \varphi \ a$ merely wrap those for IO, hence trivially obey the monad laws.

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

\[
\text{return} \ a \to \text{Sesh}_\beta \varphi \ a
\]

\[
\text{return} \ x = \text{Sesh} \ S \text{ runSesh } x
\]

The $\Rightarrow$ operator sequences two actions with types $\text{Sesh}_\beta \varphi \ a$ and $\text{Sesh}_\rho \varphi \ b$, and requires $\varphi < \rho$, i.e., the first action must have finished before the second starts. The resulting action has lower bound $\beta \land \rho$ and upper bound $\varphi \lor \rho$.

\[
(\Rightarrow) :: (\varphi < \rho) \Rightarrow \text{Sesh}_\beta \varphi \ a \to (a \to \text{Sesh}_\rho \varphi \ b) \to \text{Sesh}_\beta \varphi \rho \ b
\]

\[
\text{mx} \Rightarrow \text{mf} = \text{Sesh} \ S \text{ runSeshIO} \ \text{mx} \Rightarrow \lambda x. \text{runSeshIO} \ (\text{mf} \ x)
\]

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

We define decorated variants of the concurrency and communication primitives: $\text{send}$, $\text{recv}$, and $\text{close}$ each perform a communication action with some priority $\varphi$, and return a computation of type $\text{Sesh}_\beta \varphi \ a$, i.e., with exact bounds; $\text{new}$ and $\text{cancel}$ don’t perform any communication action, and so return a pure computation of type $\text{Sesh}_\beta \top$; $\text{fork}$ takes a computation which performs communication actions as an argument, forks it off into a separate thread, and masks the upper bound in its return type.

\[
\text{new} :: \text{Session } s \Rightarrow \text{Sesh}_\beta \top \ (s, \text{Dual } s)
\]

\[
\text{fork} :: \text{Sesh}_\beta \top \ () \to \text{Sesh}_\beta \top \()
\]

\[
\text{cancel} :: \text{Session } s \Rightarrow s \to \text{Sesh}_\beta \top \()
\]

\[
\text{send} :: \text{Session } s \Rightarrow (a, \text{Send}_\beta \ a \ s) \to \text{Sesh}_\beta \ a
\]

\[
\text{recv} :: \text{Recv}_\beta \ a \ s \to \text{Sesh}_\beta \ a \ (a, s)
\]

\[
\text{close} :: \text{End}_\beta \to \text{Sesh}_\beta \ a
\]

From these, we derive decorated choice, as before:

\[
\text{type } \text{Select}_\beta \ a_1 \to \text{Select}_\beta \ a_2 = \text{Select}_\beta \ (\text{Either } \text{Dual } a_1 \ (\text{Dual } a_2))
\]

\[
\text{type } \text{Offer}_\beta \ a_1 \to \text{Offer}_\beta \ a_2 = \text{Offer}_\beta \ (\text{Either } a_1 \to a_2)
\]

\[
\text{selectLeft} :: (\text{Session } s_1) \Rightarrow \text{Select}_\beta \ a_1 \to \text{Sesh}_\beta \ a_1
\]

\[
\text{selectRight} :: (\text{Session } s_2) \Rightarrow \text{Select}_\beta \ a_2 \to \text{Sesh}_\beta \ a_2
\]

\[
\text{offerEither} :: (o < p) \Rightarrow \text{Offer}_\beta \ a_1 \to \text{Sesh}_\beta \ a_1
\]

\[(Either \ a_1 \to a_2) \Rightarrow (\text{Either } a_1 \to \text{Sesh}_\beta \ a_1 \ a_2) \to \text{Sesh}_\beta \ a_2 \ a_1 \ a
\]

\[\text{Safe IO.} \] We can use a trick from the $ST$ monad [38] to define a "pure" variant of $\text{runSesh}$, which encapsulates all use of IO within the $\text{Sesh}_\beta \varphi \ a$ monad. The idea is to index the $\text{Sesh}_\beta \varphi \ a$ and every session type constructor with an extra type parameter tok, which we’ll call the session token:

\[
\text{send} :: \text{Session } s \Rightarrow (a, \text{Send}_\beta \ a \ tok \ s) \to \text{Sesh}_\beta \ a \ tok \ s
\]

\[
\text{recv} :: \text{recv}_\beta \ a \ tok \ s \to \text{Sesh}_\beta \ a \ tok \ s
\]

\[
\text{close} :: \text{End}_\beta \ tok \ s \to \text{Sesh}_\beta \ a \ tok \ s
\]

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

\[
\text{runSesh} :: (\forall \text{tok}. \text{Sesh}_\beta \varphi \ tok \ a) \to a
\]

\[
\text{runSesh} x = \text{unsafePerformIO} \ (\text{runSeshIO } x)
\]

This ensures that none of the channels created in the session can escape out of the scope of $\text{runSesh}$.

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 [53]. For instance, we could give the summation service from section 2.2 the following type:

newtype $\text{SumSrv}_\varphi \ a = \text{SumSrv} \ (\text{Offer}_\varphi \ (\text{Recv}_\varphi \ (	ext{Int } (\text{SumSrv}_\varphi \ a) \ (\text{Int } (\text{SumSrv}_\varphi \ a)))) \ (\text{Send}_\varphi \ (\text{Int } (\text{End}_\varphi \ a)))))$

We’d then like to assign $\text{sumSrv}$ the following type:

\[
\text{sumSrv} :: \text{Int } \to \text{SumSrv}_\varphi \ a \to \text{Sesh}_\varphi \ a
\]

\[
\text{sumSrv } \text{tot} \ (\text{SumSrv } s) = \text{offerEither } \text{Sesh}_\varphi \ a \ \text{case } x \ of
\]

\[
\text{Left } s \to \text{do } (x, s) \to \text{recv } s; \text{sumSrv } (\text{tot } + x) \ s
\]

\[
\text{Right } s \to \text{do } s \to \text{send } (\text{tot }, s); \text{weaken } \text{close } s
\]

The upper bound for a recursive call should be $\top$, which ensures that recursive calls are only made in tail position [3, 22]. 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 $\text{weaken} : \text{Sesh}_\beta \ a \to \text{Sesh}_\beta \ a$.

Unfortunately, writing such priority-polymorphic code relies heavily on GHC’s ability to reason about type-level
We then define the scheduler itself, which forwards messages from one process to the next in a cycle:

1. We could embrace the Hasochism [39], 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 [16, §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 [10] and Kokke and Dardha [35] 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:

```
<table>
<thead>
<tr>
<th>main</th>
</tr>
</thead>
<tbody>
<tr>
<td>sched</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>adder</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>sched</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>adder</td>
</tr>
</tbody>
</table>
```

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

\[
type SR_{a0} a = Send_{a0} a (Recv_{a0} a ())
\]

\[
type RS_{a0} a = Dual (SR_{a0} a)
\]

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

\[
sched :: RS_{a0} a \to SR_{a0} a \to SR_{a0} a \to SR_{a0} a \to Sesh_{a0} a()
\]

\[
sched s1 s2 s3 s4 = do
  (x, s1) \leftarrow recv s1
  s2 \leftarrow send (x, s2); (x, ()) \leftarrow recv s2
  s3 \leftarrow send (x, s3); (x, ()) \leftarrow recv s3
  s4 \leftarrow send (x, s4); (x, ()) \leftarrow recv s4
  send (x, s1)
\]

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:

\[
adder :: (a0 < a1) \Rightarrow RS_{a0} a1 Int \to Sesh_{a0} a1 ()
\]

\[
adder s = do (x, s) \leftarrow recv s; send (x + 1, s)
\]

\[
main :: (a0 < a1) \Rightarrow Int \to SR_{a0} a1 Int \to Sesh_{a0} a1 Int
\]

\[
main x s = do s; s \leftarrow send (x, s); (x, ()) \leftarrow recv s; ireturn x
\]

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 [35], 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, \(\text{end}\)\(^4\) and end\(^5\), into end\(^6\);
3. It is extended with asynchronous communication and session cancellation following Fowler et al. [20].

These changes preserve subject reduction and progress properties, and give us tighter bounds on priorities. To see why, note that PCP [10] and PGV [35] 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 [35]. For a discussion of the asynchronous semantics and session cancellation, please see Fowler et al. [20].

As in section 2.4, we let \(\pi\) 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 \(\lambda\)-calculus with product types (\(\times\)), sum types (\(\oplus\)), and their units (\(1\) and \(0\)). Linear functions (\(\cdot\)) 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:

```plaintext
S ::= !T.S | ?T.S | end
T, U ::= T \times U | 1 | T + U | 0 | T \rightarrow_p U | S
```

The types \(!T.S\) and \(?T.S\) mean “send” and “receive”, respectively, and \(\text{end}\)\(^6\) means, well, session end.

\(^3\)https://github.com/yav/type-nat-solver
\(^4\)https://hackage.haskell.org/package/ghc-typelits-presburger
The term language is the standard linear \( \lambda \)-calculus extended with concurrency primitives \( K \):

\[
L, M, N := x \mid K \mid \lambda x. M \mid M N \\
| () \mid M; N \\
| (M, N) \mid \text{let } (x, y) = M \text{ in } N \\
| \text{absurd } M \\
| \text{inl } M \mid \text{inr } M \mid \text{case } L \{ \text{inl } x \mapsto M; \text{inr } y \mapsto N \}
\]

\( K := \text{new} \mid \text{fork} \mid \text{send} \mid \text{recv} \mid \text{close} \)

The concurrency primitives are uninterpreted in the term language. Rather, they are interpreted in a configuration language based on the \( \pi \)-calculus, which we omit from this paper (see Kokke and Dardha [35]).

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

**Monadic Reflection.** The graded monad \( Sesh^\_ \) arises from the monadic reflection [17] 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 [17] demonstrates the reflection from programs of type \( T \) in a language with exceptions and handlers to programs of type \( T + \text{exn} \) in a pure language where \text{exn} is the type of exceptions.

We translate programs from PGV to Haskell programs in the \( Sesh^\_ \) monad. First, let’s look at the translation of types:

\[
\begin{align*}
&T \vdash^p U \Rightarrow \text{Sesh}^\_ [U] [1] = () \\
&\forall T. S \Rightarrow \text{Send}^\_ [T] [S] [T \times U] = (\text{[T]}, [U]) \\
&\forall T. S \Rightarrow \text{Recv}^\_ [T] [S] [0] = \text{Void} \\
&\text{end}^\_ \Rightarrow \text{End}^\_ [T + U] = \text{Either} [T] [U]
\end{align*}
\]

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 \( \text{Sesh}^\_ [T] \):

\[
\begin{align*}
[ x ] &= \text{return } x \\
[ \lambda x. L ] &= \text{return } (\lambda x. [L]) \\
[ K ] &= \text{return } [K] \\
[ L M ] &= [L] \Rightarrow \lambda f. [M] \Rightarrow \lambda x. f x \\
[ () ] &= \text{return } () \\
[ \text{let } () = L \text{ in } M ] &= [L] \Rightarrow \lambda (). M \\
[ (L, M) ] &= [L] \Rightarrow \lambda x. [M] \Rightarrow \lambda y. \text{return } (x, y) \\
[ \text{let } (x, y) = L \text{ in } M ] &= [L] \Rightarrow \lambda (x, y). [M] \\
[ \text{absurd } L ] &= [L] \Rightarrow \lambda x. \text{absurd } x \\
[ \text{inl } L ] &= [L] \Rightarrow \lambda x. \text{return } (\text{Left } x) \\
[ \text{inr } L ] &= [L] \Rightarrow \lambda x. \text{return } (\text{Right } x) \\
[ \text{case } L \{ \text{inl } x \mapsto M; \text{inr } y \mapsto N \} ] &= [L] \Rightarrow \lambda x. \text{case } x \text{ of } \{ \text{Left } x \mapsto [M]; \text{Right } y \mapsto [N] \}
\end{align*}
\]

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

\[
\begin{align*}
&\text{new : } 1 \times S \times S \\
&\text{fork : } (1 \to^p 1) \to \rightarrow^p 1 \\
&\text{recv : } ? T. S \to \rightarrow^p 1 \times S
\end{align*}
\]

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

\[
\begin{align*}
&\text{send : } T \times ! T. S \to \rightarrow^p S \\
&\text{recv : } \text{Session } [S] \Rightarrow (\text{[T]}, \text{Send}^\_ [T] [S]) \Rightarrow \text{Sesh}^\_ [S] \\
&\text{recv : } \text{Recv}^\_ [T] [S] \Rightarrow \text{Sesh}^\_ (\text{[T]}, [S]) \\
&\text{cancel : } \text{End}^\_ \Rightarrow \text{Sesh}^\_ ()
\end{align*}
\]

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 \( \Gamma \vdash^x x : a \) to mean that the Haskell program \( x \) has type \( a \) in typing environment \( \Gamma \):

**Theorem 3.1.** If \( \Gamma \vdash^p M : T \), then \( \Gamma \vdash^x [M] : \text{Sesh}^\_ [T] \).

**Proof.** Figure 2 presents the translation from typing derivations in PGV to abbreviated typing derivations in Haskell with priority-sesh. \( \square \)

### 4 Related Work

**Session Types in Haskell.** Orchard and Yoshida [49] discuss various approaches to implementing session types in Haskell. Their overview is reproduced below:

- Neubauer and Thiemann [44] give an encoding of first-order single-channel session-types with recursion;
- Using parameterised monads, Pucella and Tov [56] provide multiple channels, recursion, and some building blocks for delegation, but require manual manipulation of a session typing environment;
- Sakman and Eisenbach [57] provide an alternate approach where session types are constructed via a value-level witnesses;
- Imai et al. [30] extend Pucella and Tov [56] with delegation and a more user-friendly approach to handling multiple channels;
- Orchard and Yoshida [50] use an embedding of effect systems into Haskell via graded monads based on a formal encoding of session-typed \( \pi \)-calculus into PCF with an effect system;
- Lindley and Morris [41] provide a finally tagless embedding of the GV session-typed functional calculus into Haskell, building on a linear \( \lambda \)-calculus embedding due to Polakow [55].
Deadlock-Free Session Types in Linear Haskell

Static Typing Rules.

\[
\begin{array}{lll}
\text{T-Var} & \Gamma, x : T \vdash x : T & \\
\text{T-Lam} & \Gamma, x : T \vdash \lambda x. M : T \rightarrow \omega_p T & \\
\text{T-Const} & \emptyset \vdash \omega_k K : T & \\
\text{T-Unit} & \Gamma \vdash \text{Unit} : 1 & \\
\text{T-LetUnit} & \Gamma \vdash M : T \quad \Delta \vdash N : T \quad q < p' & \\
\text{T-LetPair} & \Gamma \vdash M : T \times T' \quad \Delta \vdash N : U \quad q < p' & \\
\text{T-CaseSum} & \Gamma \vdash \text{Case} L \{ \text{inl} x \mapsto M ; \text{inr} y \mapsto N \} : U & \\
\text{T-App} & \Gamma \vdash M : T \rightarrow o_p'' U \quad \Delta \vdash N : T \quad q < p' \quad q' < p'' & \\
\text{T-Pair} & \Gamma \vdash M : T \quad \Delta \vdash N : U \quad q < p' & \\
\text{T-Inl} & \Gamma \vdash \text{inl} M : T + U & \\
\text{T-Inr} & \Gamma \vdash \text{inr} M : T + U & \\
\text{T-Absurd} & \Gamma \vdash \text{absurd} : T & \\
\end{array}
\]

\[
\begin{align*}
\Gamma & \vdash \omega_k M : T \\
\emptyset & \vdash \omega_k () : 1 \\
\Gamma & \vdash \omega_k M : T \\
\Gamma & \vdash \omega_k M : T \\
\Gamma & \vdash \omega_k M : 0 \\
\end{align*}
\]

Type Schemas for Constants.

- new : 1 \rightarrow o_+ S \times \bar{S}
- fork : (1 \rightarrow o^- 1) \rightarrow o_+ 1
- cancel : S \rightarrow o^- 1
- send : T \times o^- T.S \rightarrow o^- S
- recv : o'' T.S \rightarrow o_+ T \times S
- close : end'' \rightarrow o^- 1

Figure 1. Typing rules for Priority GV.

Table 1. Capabilities of various implementations of session types in Haskell [adapted from 49].

<table>
<thead>
<tr>
<th></th>
<th>NT04</th>
<th>PT08</th>
<th>SE08</th>
<th>IYA10</th>
<th>OY16</th>
<th>LM16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recursion</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Delegation</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Multiple channels</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Idiomatic code</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Easy-to-write session types</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Deadlock freedom</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>via process structure</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>via priorities</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

With respect to linearity, all works above—except Neubauer and Thiemann [44]—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 more idiomatic code.

With respect to deadlock freedom, none of the works above—except Lindley and Morris [41]—guarantee deadlock freedom. However, Lindley and Morris [41] 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 [49] summarise the capabilities of the various implementations of session types in Haskell in a table, which we adapted in table 1 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 [56] only provide partial delegation, Neubauer and Thiemann [44], Pucella and Tov [56], and Lindley and Morris [41] still need to use combinators instead of standard Haskell application, abstraction, or variables in
some places, and Neubauer and Thiemann [44] is only deadlock-free on the technicality that they don’t support multiple channels.

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

**Session Types in other Programming Languages.** Session types have been integrated in other programming language paradigms. Jespersen et al. [31], Padovani [52], Scalas and Yoshida [60] integrate binary session types in the native host language, without language extensions; this to
Deadlock-Free Session Types in Linear Haskell

avoid hindering session types use in practice. To obtain this integration of session types without extensions Padovani [52], Scalas and Yoshida [60] combine \textit{static} typing of input and output actions with \textit{runtime} checking of linearity of channel usage.

Implementations of multiparty session types (MPST) are less common than binary implementations. Scalas et al. [39] integrate MPST in Scala building upon Scalas and Yoshida [60] and a continuation-passing style encoding of session types into linear types Dardha et al. [11]. There are several works on implementations of MPST in Java: Sivaramakrishnan et al. [61] implement MPST leveraging an extension of Java with session primitives; Hu and Yoshida [29] develop a MPST-based API generation for Java leveraging CFSMs by Brand and Zafiropulo [7]; and Kouzapas et al. [37] implement session types in the form of \textit{typestates} in Java. Demangeon et al. [13] implement MPST leveraging an extension of Haskell with session primitives; Lopez et al. [43] adopt a dependently-typed 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:

\begin{itemize}
  \item 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 \(\texttt{\_ \rightarrow \_}\).
  \item There is no integration with base or popular Haskell packages, and that \texttt{LinearTypes} is an extension, there likely won’t be for quite a while. There’s \texttt{LinearBase}, which provides linear variants of many of the constructs in base. However, \texttt{LinearBase} relies heavily on \texttt{unsafeCoerce}, which, ironically, may affect Haskell’s performance.
  \item Generally, there is little integration with the Haskell ecosystem, \textit{e.g.}, one other contribution we made are the formatting directives for Linear Haskell in \texttt{lhs2TeX} [1].
\end{itemize}

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 \texttt{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 \texttt{Idris2} [5, 6], which supports both linear types and complex type-level reasoning.

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.
Acknowledgments
We thank Simon Fowler and April Gonçalves for comments on the manuscript. This work is supported by the EU HORIZON 2020 MSCA RISE project 778233 “Behavioural Application Program Interfaces” (BehAPI).

References
Deadlock-Free Session Types in Linear Haskell

York, NY, USA, 24–35.  https://doi.org/10.1145/178243.178246


