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  
ornera.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 [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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Haskell ’21, August 26–27, 2021, Virtual Event, Republic of Korea  
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8615-9/21/08...$15.00
https://doi.org/10.1145/3471874.3472979
Recent works by Padovani and Novara [2015] and Kokke and Dardha [PGV, 2021a] integrate priorities [Kobayashi 2006; 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 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 [Kokke and Dardha 2021a]—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. 1996];
- in section 2.2, we use these one-shot channels to build a small library of session-typed channels [Dardha et al. 2017]; and
- in section 2.4, we decorate these session types with priorities to guarantee deadlock freedom [Kokke and Dardha 2021a].

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 → 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 used 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, andRecv1, which are two copies of the same MVar.

1https://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 (mvar₁, mvar₂) ← dup2 $ newEmptyMVar
         return (Send₁ (unur mvar₁), Recv₁ (unur mvar₂))

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 (Send₁ (mvar₁))
recv₁ :: Recv₁ a → IO a
recv₁ = Recv₁ (Recv₁ (mvar₁))

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 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, ch₄) → do send₁ ch₄; 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.”¹² Practically, consumeAndRecv throws a BlockedIndefinitelyOnMVar exception, whereas consumeAndSend does not:

consumeAndRecv = do
                  consumeAndSend = do u
                                    (ch₁, ch₂) ← new₁
                                    fork $ return (consume ch₁)
                                    fork $ return (consume ch₂)
                                    recv₁ ch₁
                                    send₁ ch₂

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 [2016]; Dardha et al. [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 (Recv 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) = do (x, s) <- recv s
                             return (mulClient (32, s))
mulClient (s :: MulClient) = do s <- send (32, s)
                              return (mulServer (s))

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

¹²https://downloads.haskell.org/~ghc/9.0.1/docs/html/libraries/base-4.15.0.0/Control-Exception.html#t:BlockedIndefinitelyOnMVar
Session. Duality must be injective—the annotation \( \text{result} \rightarrow s \) means \( \text{result} \) must uniquely determine \( s \) and involutive—\( \text{Dual} (\text{Dual} s) \sim s \) means \( \text{Dual} (\text{Dual} s) \) must equal \( s \). These constraints are all captured by the \( \text{Session} \) class, along with \( \text{new} \) for constructing channels:

\[
\begin{align*}
\text{class} \ (\text{Session} \ (\text{Dual} s), \text{Dual} \ (\text{Dual} s) \sim s) \Rightarrow \text{Session} \ s \\
\text{where} \\
\text{type} \ \text{Dual} \ s = \text{result} \ | \ \text{result} \rightarrow s \\
\text{new} :: \text{IO} \ (s, \text{Dual} s)
\end{align*}
\]

There are three primitive session types: \text{Send}, \text{Recv}, and \text{End}.

\[
\begin{align*}
\text{newtype} \ \text{Send} \ a \ s = \text{Send} \ (\text{Send}_1 \ (a, \text{Dual} s)) \\
\text{newtype} \ \text{Recv} \ a \ s = \text{Recv} \ (\text{Recv}_1 \ (a, s)) \\
\text{newtype} \ \text{End} = \text{End} \ \text{Sync}_1
\end{align*}
\]

By following Dardha et al. [2017], a channel \text{Send} wraps a one-shot channel \text{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 \text{send}. A channel \text{Recv} wraps a one-shot channel \text{Recv}_1 over which we receive some value and the channel over which we continue the session. Finally, an channel \text{End} wraps a synchronisation.

We define duality for each session type—\text{Send} is dual to \text{Recv}, \text{Recv} is dual to \text{Send}, and \text{End} is dual to itself:

\[
\begin{align*}
\text{instance} \ \text{Session} \ s \Rightarrow \text{Session} \ (\text{Send} \ a \ s) \\
\text{where} \\
\text{type} \ \text{Dual} \ (\text{Send} \ a \ s) = \text{Recv} \ (\text{Dual} s) \\
\text{new} = \text{do} \ (ch, \ ch) \leftarrow \text{new}_1 \\
\text{return} \ (\text{Send} \ ch, \ \text{Recv} \ ch) \\
\text{instance} \ \text{Session} \ s \Rightarrow \text{Session} \ (\text{Recv} \ a \ s) \\
\text{where} \\
\text{type} \ \text{Dual} \ (\text{Recv} \ a \ s) = \text{Send} \ (\text{Dual} s) \\
\text{new} = \text{do} \ (ch, \ ch) \leftarrow \text{new}_1 \\
\text{return} \ (\text{Recv} \ ch, \ \text{Send} \ ch) \\
\text{instance} \ \text{Session} \ \text{End} \\
\text{where} \\
\text{type} \ \text{Dual} \ \text{End} = \text{End} \\
\text{new} = \text{do} \ (ch, ch) \leftarrow \text{newSync}_1 \\
\text{return} \ (\text{End} \ ch, \ \text{End} \ ch)
\end{align*}
\]

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:

\[
\begin{align*}
\text{send} :: \text{Session} \ s \Rightarrow (a, \text{Send} \ a \ s) \rightarrow \text{IO} \ s \\
\text{send} \ (x, \text{Send} \ ch) = \text{do} \ (\text{here}, \text{there}) \leftarrow \text{new}_1 \\
\text{send}_1 \ ch \ (x, \text{there}) \\
\text{return} \ \text{here}
\end{align*}
\]

The \text{recv} and \text{close} operations simply wrap their corresponding one-shot operations:

\[
\begin{align*}
\text{recv} :: \text{Recv} \ a \ s \rightarrow \text{IO} \ (a, s) \\
\text{recv} \ (\text{Recv} \ ch) = \text{recv}_1 \ ch \\
\text{close} :: \text{End} \rightarrow \text{IO} () \\
\text{close} \ (\text{End} \ ch) = \text{sync}_1 \ ch
\end{align*}
\]

Cancellation. We implement session cancellation via the \text{Consumable} class. For convenience, we provide the \text{cancel} function:

\[
\begin{align*}
\text{cancel} :: \text{Session} \ s \Rightarrow s \rightarrow \text{IO} () \\
\text{cancel} \ s = \text{return} \ (\text{consume} \ s)
\end{align*}
\]

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:

\[
\begin{align*}
\text{cancelAndRecv} = \text{do} \\
\text{fork} \ (\text{send} \ ch, \text{send} \ ch) \leftarrow \text{new} \\
\text{return} \ (\text{recv} \ ch, \text{recv} \ ch)
\end{align*}
\]

\[
\begin{align*}
\text{cancelAndSend} = \text{do} \ u \\
\text{fork} \ (\text{send} \ ch, \text{send} \ ch) \leftarrow \text{new} \\
\text{return} \ (\text{recv} \ ch, \text{recv} \ ch)
\end{align*}
\]

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

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:

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

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 Either of two session continuations:

\[
\begin{align*}
\text{type} \ \text{Select} \ s_1 \ s_2 = \text{Send} \ (\text{Either} \ (\text{Dual} \ s_1) \ (\text{Dual} \ s_2)) () \\
\text{type} \ \text{Offer} \ s_1 \ s_2 = \text{Recv} \ (\text{Either} \ s_1 \ s_2) ()
\end{align*}
\]
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

![Diagram of process structure](image)

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:

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

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:

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

\[
\text{totallyFine } = \text{do } (ch_{t1}, ch_{t2}) \leftarrow \text{new}
\]

\[
\begin{align*}
\text{fork } & \text{do } (\text{void, } ()) \leftarrow \text{recv } ch_{t1} \\
& \text{send } (\text{void, } ch_{t2}) \\
& (\text{void, } ()) \leftarrow \text{recv } ch_{t2} \\
& \text{let } (\text{void, } \text{void}_{\text{copy}}) = \text{dup2 } \text{void} \\
& \text{send } (\text{void, } ch_{t1}) \\
& \text{return } \text{void}_{\text{copy}}
\end{align*}
\]

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:

\[
\text{main} \quad \text{ch}_{t1} \quad \text{ch}_{t1} \quad \text{child} \quad \text{ch}_{t2} \quad \text{ch}_{t2}
\]

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 class for them.

\[
\text{newtype SumSrv} = \text{SumSrv } (\text{Offer } (\text{Recv } \text{Int } \text{SumSrv}) \text{ (Send } \text{Int } \text{End}))
\]

We implement the summation server as a recursive function:

\[
\text{sumSrv :: Int } \rightarrow \text{SumSrv } \rightarrow \text{IO } ()
\]

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

\[
\begin{align*}
\text{Left } s & \rightarrow \text{do } (x, ) \leftarrow \text{recv } s; \text{sumSrv } (\text{tot } + x) \text{ s} \\
\text{Right } s & \rightarrow \text{do } s \leftarrow \text{send } (\text{tot, } s); \text{close } s
\end{align*}
\]

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

\[
\text{instance Session } \text{SumSrv}
\]

where

\[
\text{type Dual } \text{SumSrv} = \text{SumCnt}
\]

\[
\text{new } = \text{do } (ch_{\text{rv}}, ch_{\text{tm}}) \leftarrow \text{new}
\]

\[
\text{return } (\text{SumSrv } ch_{\text{rv}}, \text{SumCnt } ch_{\text{tm}})
\]

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:

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

\[
\text{woops } = \text{do } (ch_{t1}, ch_{t2}) \leftarrow \text{new}
\]

\[
\begin{align*}
\text{fork } & \text{do } (\text{void, } ()) \leftarrow \text{recv } ch_{t1} \\
& \text{send } (\text{void, } ch_{t2}) \\
& (\text{void, } ()) \leftarrow \text{recv } ch_{t2} \\
\end{align*}
\]

\[
\text{let } (\text{void, } \text{void}_{\text{copy}}) = \text{dup2 } \text{void} \\
\text{send } (\text{void, } ch_{t1}) \\
\text{return } \text{void}_{\text{copy}}
\]

SelectLeft :: (Session s1) \Rightarrow Select s1 s2 \rightarrow IO s1

SelectLeft s = do (here, there) \leftarrow new

\[
\text{send } (\text{Left there, s)}
\]

return here

OfferEither :: Offer s1 s2 \rightarrow (Either s1 s2 \rightarrow IO a) \rightarrow IO a

OfferEither s match = do (e, ()) \leftarrow 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 class for them.

\[
\text{newtype SumCnt} = \text{SumCnt } (\text{Select } (\text{Send } \text{Int } \text{SumCnt}) \text{ (Recv } \text{Int } \text{End}))
\]
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:

```
  recv ch₁₁  |  recv ch₁₁  |  send ch₁₁  |  recv ch₁₁
      ✡         |           ✡         |           ✡         |       ✡
     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 `send ch₁₁` and `recv ch₁₂`, and 1 to `recv ch₁₂` and `send ch₁₂`. 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 `⊥` and `⊤` for programs which don’t communicate. These are used as the identities for `∩` and `⊔` in lower and upper bounds, respectively. We let `o` range over natural numbers, `p` over lower bounds, and `q` over upper bounds.

```haskell
data Priority = ⊥ | Nat | ⊤
```

We define strict inequality (`<`), minimum (`∩`), and maximum (`⊔`) on priorities as usual.

**Channels.** We define `Send⁺`, `Recv⁺`, and `End⁺`, 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⁺⁺`, 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.

```haskell
newtype Sesh⁺⁺ a = Sesh⁺⁺ { runSeshIO :: IO a }
```

The monad operations for `Sesh⁺⁺` merely wrap those for IO, hence trivially obey the monad laws.

The `ireturn` function returns a pure computation—the type `Sesh⁺⁺` guarantees that all communications happen between `⊤` and `⊥`, hence there can be no communication at all.

```haskell
ireturn :: a → Sesh⁺⁺ a
ireturn x = Sesh⁺⁺ $ return x
```

The `⇒` operator sequences two actions with types `Sesh⁺⁺` and `Sesh⁺⁺`, and requires `q < p'`, i.e., the first action must have finished before the second starts. The resulting action has lower bound `p ∩ p'` and upper bound `q ∪ q'`.

```haskell
(⇒) :: (q < p') ⇒ Sesh⁺⁺ a → (a → Sesh⁺⁺ b) → Sesh⁺⁺ o q p' p mx⇒mf = Sesh⁺⁺ $ runSeshIO mx ⇒ λx. runSeshIO (mf x)
```

In what follows, we implicitly use `⇒` 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⁺⁺`, i.e., with exact bounds; `new` and `cancel` don’t perform any communication action, and so return a pure computation of type `Sesh⁺⁺`; `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.

```haskell
new :: Session s ⇒ Sesh⁺⁺ (s, Dual s)
fork :: Sesh⁺⁺ () ⇒ Sesh⁺⁺ ()
cancel :: Session s ⇒ Sesh⁺⁺ ()
send :: Session s ⇒ (a, Send⁺⁺ a s) → Sesh⁺⁺ o s
recv :: Recv⁺⁺ a s ⇒ Sesh⁺⁺ o (a, s)
close :: End⁺⁺ ⇒ Sesh⁺⁺ ()
```

From these, we derive decorated choice, as before:

```haskell
type Select⁺⁺ s₁ s₂ = Send⁺⁺ (Either (Dual s₁) (Dual s₂)) ()
type Offer⁺⁺ s₁ s₂ = Recv⁺⁺ (Either (s₁, s₂)) ()
selectLeft :: (Session s₁) ⇒ Select⁺⁺ s₁ s₂ ⇒ Sesh⁺⁺ o s₁
selectRight :: (Session s₂) ⇒ Select⁺⁺ s₁ s₂ ⇒ Sesh⁺⁺ o s₂
offerEither :: (o < p) ⇒ Offer⁺⁺ (Either s₁ s₂) o p
                (Either s₁ s₂ ⇒ Sesh⁺⁺ o p a) → Sesh⁺⁺ o q a
```

**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⁺⁺` monad. The idea is to index the `Sesh⁺⁺` and every session type constructor with an extra type parameter `tok`, which we’ll call the session token:

```haskell
send :: Session s ⇒ (a, Send⁺⁺ a s) → Sesh⁺⁺ o tok s
recv :: Recv⁺⁺ tok a s ⇒ Sesh⁺⁺ o tok (a, s)
close :: End⁺⁺ tok ⇒ Sesh⁺⁺ o tok ()
```

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:

```haskell
runSesh :: (∀tok. Sesh⁺⁺ tok a) → a
runSesh x = unsafePerformIO (runSeshIO x)
```
This ensures that none of the channels created in the session can escape out of the scope of \textit{runSesh}.

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

\textbf{Recursion.} We could implement recursive session via priority-polymorphic types, or via priority-shifting \cite{Padovani:2015}. For instance, we could give the \textit{summation service} from section 2.2 the following type:

\begin{verbatim}
newtype SumSrv\textsuperscript{p}
    = SumSrv (Offer\textsuperscript{p} (Recv\textsuperscript{p+1} Int (SumSrv\textsuperscript{p+2})))

(End\textsuperscript{p+1} Int (End\textsuperscript{p+2})))
\end{verbatim}

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

\begin{verbatim}
sumSrv : Int \to\!
\to\quad \textit{SumSrv} p \to\!
\to\quad \textit{Sesh} p (x)
\end{verbatim}

\begin{verbatim}
sumSrv tot (SumSrv s) = offerEither s $ \lambda e. case x of
    Left s \to\! do (x, s) \leftarrow recv s; sumSrv (tot + x) s
    Right s \to\! do s \leftarrow send (tot, s); weaken (close s)
\end{verbatim}

The upper bound for a recursive call should be \(\top\), which ensures that recursive calls are only made in \textit{tail} position \cite{Bernardi:2014, Gay:2020}. The recursive call naturally has upper bound \(\top\). However, the \textit{close} operation happens at some \textit{concrete} priority \(o + n\), which needs to be raised to \(\top\), so we’d have to add a primitive \textit{weaken : Sesh} \(\top\) \(\to\) \(\top\) \(\to\) \(\top\) \(\to\) \(\top\).

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

1. We could embrace the Hasochism \cite{Lindley: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 \textit{type-nat-solver}\textsuperscript{3} or \textit{ghc-typelists-presburger}\textsuperscript{4}. Unfortunately, \(\nabla\) and \(\square\) are beyond Presburger arithmetic, and \textit{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 \cite[Eisenberg et al. 2014, §6.1]{Eisenberg:2014} prevents us from checking exactly \textit{these cases}.

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

\textbf{Cyclic Scheduler.} Dardha and Gay \cite{Dardha:2018} and Kokke and Dardha \cite{Kokke:2021} use a \textit{finite cyclic scheduler} as an example. The cyclic scheduler has the following process structure, with the flow of information indicated by the dotted arrows:

\begin{verbatim}
\textbf{adders - sched - main}
\end{verbatim}

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

\begin{verbatim}
type SR\textsuperscript{p} a = Send\textsuperscript{p} a (Recv\textsuperscript{p} a ())
type RS\textsuperscript{p} a = Dual (SR\textsuperscript{p} a)
\end{verbatim}

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

\begin{verbatim}
sched :: RS\textsuperscript{p} a \to SR\textsuperscript{p} a \to SR\textsuperscript{p} a \to SR\textsuperscript{p} a \to Sesh\textsuperscript{p} ()

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)
\end{verbatim}

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

\begin{verbatim}
adder :: (o1 < o2) \Rightarrow RS\textsuperscript{p} Int \to Sesh\textsuperscript{p} ()
adder s = do (x, s) \leftarrow recv s; send (x + 1, s)
main :: (o1 < o2) \Rightarrow Int \to SR\textsuperscript{p} Int \to Sesh\textsuperscript{p} Int
main x s = do ; s \leftarrow send (x, s); (x, ()) \leftarrow recv s ; return x
\end{verbatim}

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.

\section{Relation to Priority GV}

The priority-sesh library is based on a variant of Priority GV \cite{Kokke:2021}, which differs in three ways:

1. it marks lower bounds \textit{explicitly} on the sequent, rather than implicitly inferring them from the typing environment;
2. it collapses the isomorphic types for session \textit{end}, \textit{end} \(\square\), and \textit{end} \(\nabla\), into \textit{end} \(\square\); and
3. it is extended with asynchronous communication and session cancellation following Fowler et al. \cite{Fowler:2019}.

\footnotesize
3\url{https://github.com/yav/type-nat-solver}
4\url{https://hackage.haskell.org/package/ghc-typelists-presburger}
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 Dardha 2021a] 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 [2021a]. For a discussion of the asynchronous semantics and session cancellation, please see Fowler et al. [2019].

As in section 2.4, we let $\alpha$ 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 ($+$), and their units (1 and 0). Linear functions ($\rightarrow$) 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:

$S := \bang S | \bang^\alpha S | ?S | ?^\alpha S | \text{end}^\alpha$

$T, U := T \times U | 1 | T + U | 0 | T \rightarrow^\alpha U | S$

The types $\bang S$ and $\bang^\alpha S$ mean “send” and “receive”, respectively, and $\text{end}^\alpha$ means, well, session end.

The term language is the standard linear $\lambda$-calculus extended with concurrency primitives $K$:

$L, M, N := x | K | \lambda x.M | M \times N | ( \cdot ) | M \cdot N | (M, N) | \text{let } (x, y) = M \text{ in } N | \text{absurd } M | \text{inl } M | \text{inr } M | \text{case } L \{ \text{inl } x \mapsto M ; \text{inr } y \mapsto N \} | \text{new } | \text{fork } | \text{send } | \text{recv } | \text{close }$

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 [2021a]).

We present the typing rules for PGV in fig. 1. A sequent $\Gamma \vdash^\alpha M : T$ 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 $\text{Sesh}_\alpha$ 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 $T + \text{exn}$ in a pure language where $\text{exn}$ is the type of exceptions.

We translate programs from PGV to Haskell programs in the $\text{Sesh}_\alpha$ monad. First, let’s look at the translation of types:

$[T \rightarrow^\alpha U] = [T] \rightarrow \text{Sesh}_\alpha [U]$ $[1] = ()$

$[\bang \bang^\alpha S] = \text{Send}^\alpha [T] [S]$ $[T \times U] = ([T],[U])$

$[?] [?] [?] = \text{Recv}^\alpha [T] [S]$ $[0] = \text{Void}$

$[\text{end}^\alpha] = \text{End}^\alpha [T + U] = \text{Either } [T] [U]$

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 $\text{Sesh}_\alpha [T]$:

$x \leadsto \text{ireturn } x$

$\lambda x. L \leadsto \text{ireturn } (\lambda x. [L])$

$K \leadsto \text{ireturn } [K]$

$L \cdot M \leadsto [L] \Rightarrow \lambda f. [M] \Rightarrow \lambda x. f \cdot x$

$() \leadsto \text{ireturn } ()$

$\text{let } (x = L \text{ in } M) \leadsto [L] \Rightarrow \lambda y. [M]$

$\text{let } (x, y = L \text{ in } M) \leadsto [L] \Rightarrow \lambda (x, y). [M]$

$\text{absurd } L \leadsto [L] \Rightarrow \lambda x. \text{absurd } x$

$\text{inl } L \leadsto [L] \Rightarrow \lambda x. \text{ireturn } (\text{Left } x)$

$\text{inr } L \leadsto [L] \Rightarrow \lambda y. \text{ireturn } (\text{Right } y)$

$\text{case } L \{ \text{inl } x \mapsto M ; \text{inr } y \mapsto N \} \leadsto [L] \Rightarrow \lambda x. \text{case } x \text{ of } \{ \text{Left } x \mapsto [M] ; \text{Right } y \mapsto [N] \}$

We translate the communication primitives from PGV to those 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 those with the same name in priority-sesh.

$\text{new} : 1 \rightarrow^\alpha S \times S$

$= \lambda(). \text{new } () \rightarrow ([S],[(\text{Dual } S)])$

$\text{fork} : (1 \rightarrow^\alpha 1) \rightarrow^\alpha 1$

$= \lambda k. \text{fork } k () : (\lambda j. \text{Sesh}_\alpha j) () \rightarrow \text{Sesh}_\alpha$

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

$\text{send} : T \times !\bang^\alpha T.S \rightarrow^\alpha S$

$= \text{send} :: \text{Session } [S] \Rightarrow ([T],[\text{send}^\alpha [T],[S]],[S]) \rightarrow \text{Sesh}_\alpha [S]$

$\text{recv} : !\bang^\alpha T.S \rightarrow^\alpha T \times S$

$= \text{recv } :: \text{Recv}^\alpha [T] [S] \rightarrow \text{Sesh}_\alpha ([T],[S])$

$\text{close} : \text{end}^\alpha \rightarrow^\alpha 1$

$= \text{close } :: \text{End}^\alpha \rightarrow \text{Sesh}_\alpha$

$\text{cancel} : S \rightarrow^\alpha 1$

$= \text{cancel } :: \text{Session } [S] \Rightarrow [S] \rightarrow \text{Sesh}_\alpha$

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 $\Gamma$:

**Theorem 3.1.** If $\Gamma \vdash^\alpha M : T$, then $[\Gamma] \vdash [M] : \text{Sesh}_\alpha [T]$. 
### Static Typing Rules.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Type Schemas for Constants.</th>
</tr>
</thead>
<tbody>
<tr>
<td>T-Var</td>
<td>( \Gamma, x: T \vdash^q x : T )</td>
</tr>
<tr>
<td>T-Lam</td>
<td>( \Gamma, \lambda x. M : T \vdash^q \lambda x. M : T )</td>
</tr>
<tr>
<td>T-Unit</td>
<td>( \emptyset \vdash^q () : 1 )</td>
</tr>
<tr>
<td>T-LetPair</td>
<td>( \Gamma \vdash^q M : T \times T' ) ( \Delta, x : T, y : T' \vdash^q N : U ) ( q &lt; p' )</td>
</tr>
<tr>
<td>T-CaseSum</td>
<td>( \Gamma \vdash^q L : T + T' ) ( \Delta, x : T \vdash^q M : U ) ( \Delta, y : T' \vdash^q N : U ) ( q &lt; p' )</td>
</tr>
<tr>
<td>T-App</td>
<td>( \Gamma \vdash^q M : T \rightarrow T' ) ( \Delta \vdash^q N : U ) ( q &lt; p' ) ( q' &lt; p'' )</td>
</tr>
<tr>
<td>T-Pair</td>
<td>( \Gamma \vdash^q M : T \times T ) ( \Delta \vdash^q N : U ) ( q &lt; p' )</td>
</tr>
<tr>
<td>T-Const</td>
<td>( \emptyset \vdash^q K : T )</td>
</tr>
<tr>
<td>T-LetUnit</td>
<td>( \Gamma \vdash^q M : 1 ) ( \Delta \vdash^q N : T ) ( q &lt; p' )</td>
</tr>
<tr>
<td>T-Inl</td>
<td>( \Gamma \vdash^q \text{inl} M : T + U )</td>
</tr>
<tr>
<td>T-Inr</td>
<td>( \Gamma \vdash^q \text{inr} M : T + U )</td>
</tr>
<tr>
<td>T-Absurd</td>
<td>( \Gamma \vdash^q \text{absurd} M : 0 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type Schemas for Constants.</th>
</tr>
</thead>
<tbody>
<tr>
<td>new : 1 (-\frac{1}{p} S \times S)</td>
</tr>
<tr>
<td>for : ((1 \rightarrow^p 1) \rightarrow_{\frac{1}{q}} 1)</td>
</tr>
<tr>
<td>cancel : (S \rightarrow_{\frac{1}{p}} 1)</td>
</tr>
<tr>
<td>send : ((T \times T) \rightarrow_{\frac{1}{p}} T \times S)</td>
</tr>
<tr>
<td>recv : ((T \times T) \rightarrow_{\frac{1}{p}} T \times S)</td>
</tr>
<tr>
<td>close : (\text{end} \rightarrow_{\frac{1}{p}} 1)</td>
</tr>
</tbody>
</table>

\[ K : T \]

**Figure 1.** Typing rules for Priority GV.

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

### 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 \(\pi\)-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 \(\lambda\)-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 sidestep 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 [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
\[
x : T \vdash \mathsf{return} x :: \text{Sesh}_T^\mathsf{p} \cdot \llbracket T \rrbracket
\]

\[
\Gamma, x : T \vdash q_p L : U \\
\vdash \text{return} (\lambda x : \llbracket L \rrbracket) :: \text{Sesh}_T^\mathsf{p} \cdot \llbracket U \rrbracket
\]

\[
\Gamma \vdash \lambda x. L : T \to q_p U = \text{return} \cdot (\llbracket T \rrbracket \to \text{Sesh}_T^\mathsf{p} \cdot \llbracket U \rrbracket)
\]

\[
\varnothing \vdash () : \mathsf{1} = \text{return} \cdot () :: \text{Sesh}_T^\mathsf{p} \cdot ()
\]

\[
\Gamma \vdash q_p L : 1 \\
\Delta \vdash q_p M : T \quad q < p'
\]

\[
\Gamma, \Delta \vdash q_p \cdot (\lambda x : M) \cdot L \cdot U = \llbracket \lambda x. [M] \Rightarrow \lambda y. \text{return} \cdot (x, y) :: (q < p') \Rightarrow \text{Sesh}_p^{q\cdot q_p \cdot q_p} \cdot \llbracket U \rrbracket
\]

\[
\Gamma \vdash q_p L : T \times T' \\
\Delta, x : T, y : T' \vdash q_p M : U \quad q < p'
\]

\[
\Gamma, \Delta \vdash q_p \cdot (\lambda x, y : M) \cdot L \cdot U = \llbracket \lambda (x, y). [M] :: (q < p') \Rightarrow \text{Sesh}_p^{q\cdot q_p \cdot q_p} \cdot \llbracket U \rrbracket
\]

\[
\Gamma \vdash q_p L : T \\
\vdash \text{return} \cdot (\lambda x : \llbracket L \rrbracket) :: \text{Sesh}_T^\mathsf{p} \cdot \llbracket T \rrbracket
\]

\[
\Gamma \vdash q_p \cdot \mathsf{inl} L : T + U = \llbracket \lambda x. \text{return} \cdot (\mathsf{Left} x) :: \text{Sesh}_T^\mathsf{p} \cdot \llbracket U \rrbracket \mathsf{Either} \cdot \llbracket T \rrbracket \cdot \llbracket U \rrbracket)
\]

\[
\Gamma \vdash q_p \cdot \mathsf{inr} L : T + U = \llbracket \lambda x. \text{return} \cdot (\mathsf{Right} x) :: \text{Sesh}_T^\mathsf{p} \cdot \llbracket U \rrbracket \mathsf{Either} \cdot \llbracket T \rrbracket \cdot \llbracket U \rrbracket)
\]

\[
\Gamma \vdash q_p L : T + T' \\
\Delta, x : T, y : T' \vdash q_p M : U \quad \Delta, y : T' \vdash q_p N : U \quad q < p'
\]

\[
\Gamma, \Delta \vdash q_p \cdot (\lambda x : M; \mathsf{inr} y \mapsto N) : U = \llbracket \lambda x. \text{return} \cdot \mathsf{Either} \cdot \llbracket T \rrbracket \cdot \llbracket T' \rrbracket \cdot \llbracket M \rrbracket :: \text{Sesh}_T^\mathsf{p} \cdot \llbracket U \rrbracket \mathsf{Either} \cdot \llbracket T' \rrbracket \cdot \llbracket N \rrbracket :: \text{Sesh}_T^\mathsf{p} \cdot \llbracket U \rrbracket
\]

\[
\Gamma \vdash q_p \cdot \mathsf{absurd} L : T = \llbracket \lambda x. \text{return} \cdot \mathsf{absurd} x :: \text{Sesh}_T^\mathsf{p} \cdot \llbracket T \rrbracket
\]

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

---

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 [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 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 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.


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 [2021a] introduce PGV–Priority GV, by following Dardha and Gay [2018] and allowing for more flexible programming in GV. Fowler et al. [2021] present Hypersequent GV (HGV), a core calculus for functional programming with session types that enjoys deadlock freedom, confluence, and strong normalisation.

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 \_ \rightarrow \_

- 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-morphologistic polymorphism 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 \citep{Brady2013, Brady2017}, which supports both linear types and complex type-level reasoning.

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


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


\(^5\)https://hackage.haskell.org/package/lhs2tex

Deadlock-Free Session Types in Linear Haskell


Renamed to Sesh.


