10

Session Types with Linearity in Haskell

Dominic Orchard[1] and Nobuko Yoshida[2]

[1]University of Kent, UK
[2]Imperial College London, UK

Abstract

Type systems with parametric polymorphism can encode a significant proportion of the information contained in session types. This allows concurrent programming with session-type-like guarantees in languages like ML and Java. However, statically enforcing the linearity properties of session types, in a way that is also natural to program with, is more challenging. Haskell provides various language features that can capture concurrent programming with session types, with full linearity invariants and in a mostly idiomatic style. This chapter overviews various approaches in the literature for session typed programming in Haskell.

As a starting point, we use polymorphic types and simple type-level functions to provide session-typed communication in Haskell without linearity. We then overview and compare the varying approaches to implementing session types with static linearity checks. We conclude with a discussion of the remaining open problems.

The code associated with this chapter can be found at `http://github.com/dorchard/betty-book-haskell-sessions`.

10.1 Introduction

Session types are a kind of behavioural type capturing the communication behaviour of concurrent processes. While there are many variants of session types, they commonly capture the sequence of sends and receives performed over a channel and the types of the messages carried by these interactions. A significant aspect of session types is that they enforce *linear* use of channels:

every send must have exactly one receive (no orphan messages), and vice versa (no hanging receives). These properties are often referred to together as *communication safety*. A channel cannot be reused once it has "used up" its capability to perform sends and receives. This aspect of session types makes them hard to implement in languages which do not have built-in notions of linearity and resource consumption in the type system.

The following two example interactions will be used throughout.

Example 1 (Integer equality server and client)**.** Consider a simple server which provides two modes of interaction (services) to clients. If a client chooses the first service, the server can then receive two integers, compare these for equality, send the result back as a boolean, and then return to the start state. The second service tells the server to stop hence it does not return to providing the initial two services.

A potential client requests the first behaviour, sends two integers, receives a boolean, and then requests that the server stop. These server and client behaviours are captured by the following session types, using the notation of Yoshida and Vasconcelos [18], which describe the interaction from the perspective of opposite channel endpoints:

$$\begin{aligned} \mathit{Server} &:= \mu\alpha.\&\{\mathsf{eq} : ?\mathbb{Z}.?\mathbb{Z}.!\mathbb{B}.\alpha,\ \mathsf{nil} : \mathbf{end}\} \\ \mathit{Client} &:= \oplus\{\mathsf{eq} : !\mathbb{Z}.!\mathbb{Z}.?\mathbb{B}.\ \oplus\{\mathsf{nil} : \mathbf{end}\}\} \end{aligned}$$

The server has a *recursive* session type, denoted $\mu\alpha.S$ which binds the variable α in scope of a session type S. Session types are typically equi-recursive, such that $\mu\alpha.S \equiv S[\mu\alpha.S/\alpha]$. The operator $\&$ denotes a choice offered between branches, labelled here as eq and nil. In the eq case, two integers are received and a boolean is sent before recursing with α. In the nil case the interaction finishes, denoted by **end**.

The client selects the eq service, denoted by $\oplus$. Two integers are sent and a boolean is received. Then the nil behaviour is selected via $\oplus$, ending the interaction. Session types thus abstract communication over a channel, or equivalently, they describe a channel's *capabilities*.

The two types are *dual*: they describe complementary communication behaviour on opposite end-points of a channel. Duality can be defined inductively as a function on session types:

$$\begin{array}{llll} \overline{!\tau.S} = ?\tau.\overline{S} & \overline{\&\{l_i : S_i\}_{i\in I}} = \oplus\{l_i : \overline{S_i}\}_{i\in I} & \overline{\mu\alpha.S} = \mu\alpha.\overline{S[\overline{\alpha}/\alpha]} \\ \overline{?\tau.S} = !\tau.\overline{S} & \overline{\oplus\{l_i : S_i\}_{i\in I}} = \&\{l_i : \overline{S_i}\}_{i\in I} & \overline{\mathbf{end}} = \mathbf{end} \end{array}$$

Recursion variables come in two flavours: α and their dual $\overline{\alpha}$. The dual of a dualised variable $\overline{(\overline{\alpha})} = \alpha$ is the undualised α. This formulation of duality with recursive types is due to Lindley and Morris [7].

Duality enforces communication safety. If the communication patterns of the server and client do not match then duality does not hold. Duality also encompasses linearity, as any repetition of actions by the server or client leads to non-matching communication behaviour.

Example 2 (Delegating integer equality)**.** Following the expressive power of the π-calculus, session types can also capture *delegation*, where channels are passed over channels. Thus, the types of communicated values τ include session types of communicated channels, written $\langle S \rangle$.

As a permutation on the previous example, we introduce a layer of indirection through delegation. The server, after receiving two integers, now receives a channel over which the resulting boolean should be sent. Dually, the client sends a channel which has the capability of sending a boolean. This is captured by the session types:

$$\begin{aligned} \mathit{Server} &:= \mu\alpha.\&\{\mathsf{eq} : ?\mathbb{Z}.?\mathbb{Z}.?\langle !\mathbb{B}\rangle.\alpha, \mathsf{nil} : \mathbf{end}\} \\ \mathit{Client} &:= \oplus\{\mathsf{eq} : !\mathbb{Z}.!\mathbb{Z}.!\langle !\mathbb{B}\rangle.\ \oplus\{\mathsf{nil} : \mathbf{end}\}\} \end{aligned}$$

The server's capability to receive a channel, over which a boolean is sent, is denoted $?\langle !\mathbb{B}\rangle$ whose dual in the client is $!\langle !\mathbb{B}\rangle$: the sending of a channel over which a boolean can be sent.

The reader is referred to the work of Yoshida and Vasconcelos [18] for a full description of a session type theory for the π-calculus on which our more informal presentation is based here.

To unpack the problem of encoding session type linearity in Haskell, we first introduce a relatively simple encoding of session types capturing sequences of send and receive actions on channels and some notion of session duality. However, this approach does not fully enforce linearity (Section 10.2). We then overview the various approaches in the literature for encoding session types in Haskell, focusing on their approach to linearity (Section 10.3). Outstanding problems and open questions in this area are discussed finally in Section 10.4.

Throughout, "Haskell" refers to the variant of Haskell provided by GHC (the Glasgow Haskell Compiler) which provides various type system extensions, the use of which is indicated and explained as required.

10.2 Pre-Session Types in Haskell

Haskell provides a library for message-passing concurrency with channels similar in design to the concurrency primitives of CML [14]. The core primitives have types:

```
newChan   :: IO (Chan a)        writeChan :: Chan a -> a -> IO ()
readChan  :: Chan a -> IO a     forkIO    :: IO () -> IO ThreadId
```

These functions operate within the `IO` monad for encapsulating side-effectful computations; creating channels (`newChan`), sending and receiving values on these channels (`writeChan` and `readChan`), and forking processes (`forkIO`) are all effectful. Channels have a single type and are bi-directional. The following program implements Example 1:

```
server c d = do
  x <- readChan c
  case x of
    Nothing -> return ()
    Just x' -> do
      (Just y') <- readChan c
      writeChan d (x' == y')
      server c d
```

```
client c d = do
  writeChan c (Just 42)
  writeChan c (Just 53)
  r <- readChan d
  putStrLn $ "Result: " ++ show r
  writeChan c Nothing
```

```
main = do {c <- newChan; d <- newChan; forkIO (client c d); server c d}
```

The choice between the two services is provided via a `Maybe` type, where `server :: Chan (Maybe Int) -> Chan Bool -> IO ()`. Two channels are used so that values of different type can be communicated. The channel types ensure *data safety*: communicated values are of the expected type. However, this typing cannot ensure communication safety. For example, the following two alternate clients are well-typed but are communication unsafe:

```
client' c d = do
    writeChan c (Just 42)
    writeChan c (Just 53)
    writeChan c (Just 53)
    r <- readChan d
    putStrLn $ "Result: " ++ show r
    writeChan c Nothing
```

```
client'' c d = do
    writeChan c (Just 42)
    readChan c
    r <- readChan d
    putStrLn $ "Result: " ++ show r
    writeChan c Nothing
```

On the left, an additional message is sent which is left unreceived in the server's channel buffer. On the right, a spurious `readChan` occurs after the first `writeChan` leading to a deadlock for the server and client.

```
send c x = do
    c' <- newChan
    writeChan c (Send x c')
    return c'

recv c = do
    (Recv x c') <- readChan c
    return (x, c')
```

```
fork f = do
    c <- newChan
    c' <- newChan
    forkIO (link (c, c'))
    forkIO (f c)
    return c'

close c = return ()
```

Figure 10.1 Implementations of the communication-typed combinators where `link :: Links => (Chan s, Chan (Dual s)) -> IO ()`.

A significant proportion of communication safety (mainly the order of interactions) can be enforced with just algebraic data types, polymorphism, and a type-based encoding of duality.

10.2.1 Tracking Send and Receive Actions

Taking inspiration from Gay and Vasconcelos [3], we define the following alternate combinators (with implementations shown in Figure 10.1) and data types:

```
send  :: Chan (Send a t) -> a -> IO (Chan t)
recv  :: Chan (Recv a t) -> IO (a, Chan t)
close :: Chan End -> IO ()

data Send a t = Send a (Chan t)
data Recv a t = Recv a (Chan t)
data End
```

The `send` combinator takes as parameters a channel which can transfer values of type `Send a t` and a value `x` of type `a` returning a new channel which can transfer the values of type `t`. This is implemented via the constructor `Send`, pairing the value `x` with a new channel `c'`, sending those on the channel `c`, and returning the new continuation channel `c'`.

The `recv` combinator is somewhat dual to this. It takes a channel `c` on which is received a pair of a value `x` of type `a` and channel `c'` which can transfer values of type `t`. The pair `(x, c')` is then returned. The `close` combinator discards its channel which has only the capability of transferring `End` values, which are uninhabited (empty data types).

The following implements a non-recursive version of the integer equality server with delegation from Example 2 (for brevity `C` = `Chan`):

```
server :: C (Recv Int (Recv Int (Recv (C (Send Bool End)) End))) -> IO ()
server c = do
    (x, c) <- recv c
    (y, c) <- recv c
    (d, c) <- recv c
    d <- send d (x == y)
    close c
    close d
```

The type of the channel `c` gives a representation of the session type $?\mathbb{Z}.?\mathbb{Z}.?\langle!\mathbb{B}\rangle.$**end** from Example 2. At each step of the program, the channel returned by a send or receive is bound to a variable shadowing the channel variable used *e.g.* `(x,c) <- recv c`. This programming idiom provides linear channel use.

10.2.2 Partial Safety via a Type-Level Function for Duality

One way to capture duality is via a *type family*. Type families are primitive recursive type functions, with strong syntactic restrictions to enforce termination. We define the (closed) type family `Dual`:

```
type family Dual s where
            Dual (Send a t) = Recv a (Dual t)
            Dual (Recv a t) = Send a (Dual t)
            Dual End        = End
```

Duality is used to type the `fork` operation, which spawns a process with a fresh channel, returning a channel of the dual type:

```
fork :: Link s => (Chan s -> IO ()) -> IO (Chan (Dual s))
```

Figure 10.1 shows the implementation which uses a method `link` of the type class `Link` to connect sent messages to received messages and vice versa. A client interacting with `server` above can then be given as:

```
client c = do
    c <- send c 42
    c <- send c 53
    d <- fork (\d' -> do { c <- send c d'; close c })
    (r, d) <- recv d
    putStrLn ("Result: " ++ show r)
    close d

example = do { c' <- fork client; server c' }
```

Thus, the client sends two integers on `c` then creates a new channel `d'`, which is sent via `c` before `c` is closed. On the returned channel `d` (with dual session type to `d'`), we receive the result, which is output before closing `d`. Thus, `Chan` essentially provides the end-points of a bi-directional channel. The type of `client` can be given as:[1]

```
client :: (Dual s ~ Recv Bool End, Link s) =>
          Chan (Send Int (Send Int (Send (Chan s) End))) -> IO ()
```

Swapping a `send` for a `recv`, or vice versa, means the program will no longer type check. Likewise, sending or receiving a value of the wrong type or at the wrong point in the interaction is also a type error.

10.2.3 Limitations

The approach described so far captures sequences of actions, but cannot enforce exact linear usage of channels; nothing is enforcing the idiom of shadowing each channel variable once it is used. For example, the first few lines of the above example client could be modified to:

```
client c = do
    c <- send c (42 :: Int)
    _ <- send c 53
    c <- send c 53
    ...
```

By discarding the linear variable-shadowing discipline, an extra integer is sent on `c` in the third line. This is not prevented by the types. While the typing captures the order of interactions, it allows every action to be repeated, and entire session interactions to be repeated. Thus, the session type theory captured above is a kind of Kleene-star-expanded version where sequences of actions in a session type $A_1 . \ldots . A_n$.**end** are effectively expanded to allow arbitrary repetition of individual actions and entire interaction sequences: $(A_1^* . \ldots . A_n^*)^*$.**end**.

We thus need some additional mechanism for enforcing proper linear use of channels, rather than relying on the discipline or morality of a programmer writing against a communication specification. We have also not yet considered branching behaviour or recursion, which are highlighted in the approaches from the literature.

[1] A more general type can be inferred, since both `Int` types can be replaced with arbitrary types of the `Num` class and `Bool` with an arbitrary type of the `Show` class.

10.3 Approaches in the Literature

There are various different approaches in the literature providing session-typed concurrent, communicating programs in Haskell with linearity:

- Neubauer and Thiemann [9] give an encoding of first-order single-channel session types with recursion;
- Using *parameterised monads*, Pucella and Tov [13] provide multiple channels, recursion, and some building blocks for delegation, but require manual manipulation of a session type context;
  (`http://hackage.haskell.org/package/simple-sessions`)
- Sackman and Eisenbach [15] provide an alternate approach where session types are constructed via a value-level witness;
  (`http://hackage.haskell.org/package/sessions`)
- Imai et al. [5] extend Pucella-Tov with delegation and a more user-friendly approach to handling multiple channels;
  (`http://hackage.haskell.org/package/full-sessions`)
- Orchard and Yoshida [11] use an embedding of effect systems into Haskell via graded monads based on a formal encoding of session-typed π-calculus into PCF with an effect system;
  (`https://github.com/dorchard/sessions-in-haskell`)
- Lindley and Morris [8] provide a *finally tagless* embedding of the GV session-typed functional calculus into Haskell, building on a linear λ-calculus embedding due to Polakow [12].
  (`https://github.com/jgbm/GVinHs`)

The following table summarises the various implementations' support for desirable session-type implementation features: recursion, delegation, multiple channels (for which we summarise how session contexts are modelled and its members are accessed), idiomatic Haskell code, and whether manual user-given specification of session types is feasible.

	NT04	PT08	SE08	IYA10	OY16	LM16
Recursion	✓	✓ deBruijn	✓labels	✓	Affine	
Delegation			✓	✓	✓	✓
Multi-channel		✓	✓	✓	✓	✓
– Contexts		stack	map	list	map	list
– Access		positional	labels	deBruijn	names	member
Idiomatic	✓	✓		✓✓	✓✓	✓
Manual spec	✓	✓	✓value		✓	✓

We characterise idiomatic Haskell as code which does not require interposing combinators to replace standard syntactic elements of functional languages, *e.g.*, λ-abstraction, application, *let*-binding, recursive bindings, and variables. In the above, for example, PT08 has one tick and IYA10 has two since PT08 must use specialised combinators for handling multiple channel variables whilst IYA10 does not require such combinators, instead using standard Haskell variables.

10.3.1 Note on Recursion and Duality

Early formulations of session types *e.g.* [18], defined duality of recursive types as $\overline{\mu\alpha.S} = \mu\alpha.\overline{S}$. Whilst this duality is suitable for tail-recursive session types, it is inadequate when recursive variables appear in a communicated type [2]. For example, the type $\mu\alpha.!\langle\alpha\rangle$ should have the unfolded dual type of $?\langle\mu\alpha.!\langle\alpha\rangle\rangle$ but under the earlier approach is erroneously $?\langle\mu\alpha.?\langle\alpha\rangle\rangle$. In Section 10.1, duality was defined using dualisable recursion variables, akin to Lindley and Morris [7], which solves this problem. However, all session-type implementations which support delegation and recursion (PT08, IYA10, OY16) implement the erroneous duality. This is an area for implementations to improve upon.

10.3.2 Single Channel; Neubauer and Thiemann [9]

Neubauer and Thiemann provided the first published implementation of session types in Haskell. Their implementation is based on a translation from a simple session-typed calculus that is restricted to one end of a single communication channel. The session type theory is first order (*i.e.*, no channel delegation), but includes alternation and recursive sessions using a representation based on the following data types:

```
data NULL         = NULL         -- the closed session
data EPS          = EPS          -- the empty session
data SEND_MSG m r = SEND_MSG m r -- send message m, then session r
data RECV_MSG m r = RECV_MSG m r -- receive message m, then session r
data ALT l r      = ALT l r      -- alternative session: either l or r
data REC f        = REC (f (REC f)) -- fixed-point of a parametric type
```

Session types are specified by defining a value using the above data constructors which provides a homomorphic type-level representation of the session

type. For example, the following value and its type describes a sequence of receiving two integers and sending a bool:

```
simple = RECV_MSG intW (RECV_MSG intW (SEND_MSG boolW EPS))
```

where `intW = 0`, `boolW = False` witness the integer and boolean types and `simple :: RECV_MSG Int (RECV_MSG Int (SEND_MSG Bool EPS))`.

Duality is provided by parameterising such specification values by placeholders for the 'send' and 'receive' actions which can then be applied to `SEND_MSG` and `RECV_MSG` in one order or the other to provide the dual specification. For example, the above specification becomes:

```
simple (send :: (forall x y . x -> y -> s x y))
       (recv :: (forall x y . x -> y -> r x y)) =
    recv intW (recv intW (send boolW EPS))
```

This function specialisations to the dual behaviour of the server via `(simple RECV_MSG SEND_MSG)` and the client `(simple SEND_MSG RECV_MSG)`.

A recursive session type $(\mu\beta.\gamma)$ is represented as a fixed-point, via `REC`, of a parametric data type representing γ. For Example 1, the body of the server's recursive type $\&\{\mathsf{eq} : ?\mathbb{Z}.?\mathbb{Z}.!\mathbb{B}.\alpha, \mathsf{nil} : \mathbf{end}\}$ can be represented by the following data type, which also uses `ALT`:

```
data Exm s r a =
   MkExm (ALT (r Label (r Int (r Int (s Bool a)))) (r Label EPS))
```

where `data Label = Eq | Nil`. The full specification is constructed as:

```
exampleSpec (send :: (forall x y . x -> y -> s x y))
            (recv :: (forall x y . x -> y -> r x y)) = a0
  where a0 = REC (MkExm (ALT
               (recv Eq (recv intW (recv intW (send boolW a0))))
               (recv Nil EPS)))
```

A computation at one end-point of a channel is represented by the `Session` data type which is indexed by the session type representation and internally wraps the `IO` monad. The main communication primitives produce values of `Session`:

```
class SEND st message nextst | st message -> nextst where
    send    :: message -> Session nextst () -> Session st ()
class RECEIVE st cont | st -> cont where
    receive :: cont -> Session st ()
close   :: Session NULL () -> Session EPS ()
```

The `SEND` class provides sending values of type `message` given a continuation session with specification `nextst`, returning a computation with specification `st`. The *functional dependency* `st message -> nextst` enforces that the instantiation of `st` and `message` uniquely determines `nextst`. An instance `SEND (SEND_MSG m b) m b` specialises `send` to:

```
send :: m -> Session b () -> Session (SEND_MSG m b) ()
```

The `RECEIVE` class abstracts receiving, taking a general continuation and returning a computation with communication specified by `st`. For `RECV_MSG` and `ALT`, the `receive` method is specialised at the types:

```
receive :: (m -> Session x ()) -> Session (RECV_MSG m x) ()
receive :: (RECV s m, RECV s' m') => ALT m m' -> Session (ALT s s') ()
```

with `RECV` shorthand for `RECEIVE`. The Example 1 server can be defined:

```
exampleServer socket = do
    (h, _ ,_) <- accept socket
    let session    = receive (ALT (\Eq -> recvNum1) (\Nil -> finish))
        recvNum1   = receive (\x -> recvNum2 x)
        recvNum2 x = receive (\y -> sendEq x y)
        sendEq x y = send (x == y) session
        finish     = close (io $ putStrLn "Fin.")
    str <- hGetContents h
    run session (exampleSpec SEND_MSG RECV_MSG) str h
```

The communication pattern of `session` (line 3), encoded by its type, must match that of the specification `exampleSpec SEND_MSG RECV_MSG` as enforced by the `run` deconstructor which expects a computation of type `Session st a` and a corresponding specification value of type `st`. Any deviation from the specification is a static type error. Since computations are wrapped in the indexed `Session` type, they can only be executed via `run` and thus are always subject to this linearity check. This contrasts with the simple approach in Section 10.2 where actions on channels produce computations in the (unindexed) `IO` monad, which allowed arbitrary repetition of actions within the specified behaviour.

10.3.3 Multi-Channel Linearity; Pucella and Tov [13]

Pucella and Tov improve on the previous approach, providing multi-channel session types with recursion and some higher-order support, though not full delegation. Similarly to Neubauer-Thiemann, the basic structure of session

types is represented by several data types: binary type constructors `:!:` and `:?:` for send and receive and `Eps` for a closed session. Offering and selecting of choices are represented by binary type constructors `:&:` and `:+:`, which differs to Neubauer-Thiemann who coalesce these dual perspectives into `ALT`. Duality is defined as a relation via a type class with a functional dependency enforcing bijectivity:

```
class Dual r s | r -> s, s -> r
instance Dual r s => Dual (a :!: r) (a :?: s)
instance Dual r s => Dual (a :?: r) (a :!: s)
instance Dual Eps Eps
instance (Dual r1 s1, Dual r2 s2) => Dual (r1 :+: r2) (s1 :&: s2)
instance (Dual r1 s1, Dual r2 s2) => Dual (r1 :&: r2) (s1 :+: s2)
instance Dual r s => Dual (Rec r) (Rec s)
instance Dual (Var v) (Var v)
```

Recursive session types use a De Bruijn encoding where `Rec r` introduces a new recursive binder over `r` and `Var n` is the De Bruijn index of the n^{th} binder where n has a unary encoding (*e.g.*, `Z`, `S Z`, *etc.*).

Communication is provided by channels `Channel c` (which we abbreviate to `Chan c`) where the type variable `c` represents the name of the channel. The session type of a channel `c` is then a *capability* provided by the data type `Cap c e s` which associates session type `s` to channel `c` with an environment `e` of recursive variables paired with session types.

A *parameterised monad* [1] is used to capture the session types of the free channels in a computation. Parameterised monads generalise monads to type constructors indexed by a pair of types akin to pre- and post-conditions. Its operations are represented via the class:

```
class ParameterisedMonad (m :: k -> k -> * -> *) where
  (>>=) :: m p q a -> (a -> m q r b) -> m p r b
  return :: a -> m p p a
```

The "bind" operation `>>=` for sequential composition has type indices representing sequential composition of Hoare triples: a computation with post-condition `q` can be composed with a computation with pre-condition `q`. Relatedly, a pure value of type `a` can be lifted into a trivial computation which preserves any pre-condition `p` in its post-condition.

One of the original examples of parameterised monads is for encoding first-order single-channel session-typed computations [1]. This is expanded upon by Pucella and Tov to multi-channels. They provide a parameterised monad `Session`, indexed by *stacks* of session type capabilities associated

to channels. Pre-conditions are the channel capabilities at the start of a computation, and post-conditions are the remaining channel capabilities after computation.

Stacks are constructed out of tuples where `()` is the empty stack. For example, `(Chan c e s, (Chan c' e' s', ()))` is a stack of two capabilities for channels `c` and `c'`. The core communication primitives then manipulate the capability at the top of the stack:

```
send :: Chan c -> a -> Session (Cap c e (a :!: s), x) (Cap c e s, x) ()
recv :: Chan c ->      Session (Cap c e (a :?: s), x) (Cap c e s, x) a
```

For example, sending a value of type `a` on channel `c` requires the capability `a :!: s` at the top of the stack for `c` in the pre-condition, which becomes `s` in the post condition. Branching follows a similar scheme.

Recursive behaviour is mediated by combinators which provide the unrolling of a recursive session type (`enter`) and referencing a bound De-Bruijn-indexed recursive variable via `zero` and `suc`:

```
enter :: Chan c -> Session (Cap c e (Rec s), x) (Cap c (s, e) s, x) ()
zero :: Chan c -> Session (Cap c (s,e) (Var Z), x) (Cap c (s,e) s, x) ()
suc  :: Session (Cap t (r, e) (Var (S v)), x) (Cap t e (Var v), x) ()
```

Thus, entering a recursive sessions type adds the body of the type onto the top of De-Bruijn environment stack; `zero` peeks the session type from the top of the stack and `suc` pops and decrements the variable. The original paper has a slightly different but equivalent formulation for `suc`– the above is provided by the online implementation.

Example 1 can then be implemented as follows:

```
server c = do
    enter c
    loop
 where loop = offer c
         (do x <- recv c
             y <- recv c
             send c (x == y)
             zero c
             loop)
         (close c)
```

```
client c = do
    enter c
    sel1 c
    send c 42
    send c 53
    x <- recv c
    io $ putStrLn $ "Got: " ++ show x
    zero c
    sel2 c
    close c
```

The types of both can be inferred. For example, the type of `server` is:

```
server :: Eq a => Chan t -> Session
    (Cap t e (Rec ((a :?: (a :?: (Bool :!: Var Z))) :&: Eps)), x) x ()
```

Dual endpoints of a channel are created by functions `accept` and `request` capturing the notion of *shared channels* [18], called a *rendezvous* here:

```
accept :: Rendezvous r ->
   (forall t. Chan t -> Session (Cap t () r, x) y a) -> Session x y a
request :: Dual r r' => Rendezvous r ->
   (forall t. Chan t -> Session (Cap t () r', x) y a) -> Session x y a
```

Thus, for our example, the server and client processes can be composed by the following code which statically enforces duality through `request`:

```
example = runSession $ do rv <- io newRendezvous
                          forkSession (request rv client)
                          accept rv server
```

with `forkSession :: Session x () () -> Session x () ()` enforcing a closed final state for the forked subcomputation (line 2). Whilst the above code is fairly idiomatic Haskell (modulo the management of recursion variables), the example has only one channel. In the context of multiple channels, the capability of a channel may not be at the top of the session environment stack, thus context manipulating combinators must be used to rearrange the stack:

```
swap :: Session (r, (s, x)) (s, (r, x)) ()
dig  :: Session x x' a -> Session (s, x) (s, x') a
```

where `swap` is akin to exchange and `dig` moves down one place in the stack. Thus, multi-channel code requires the user to understand the type-level stack representation and to manipulate it explicitly. Multi-channel code is therefore non-idiomatic, in the sense that we can't just use Haskell variables on their own.

Example 2 cannot be captured as channels cannot be passed. Pucella and Tov provide a way to send and receive capabilities, however there is no primitive for sending channels along with an associated capability. Imai *et al.* describe a way to build this on top of Pucella and Tov's approach with an existentially quantified channel name, however this is still limited by the lack of a new channel constructor. Instead, channel delegation could be emulated with global shared channels for every delegation but this shifts away from the message-passing paradigm.

In their paper, Pucella and Tov use the `ixdo` notation which copies exactly the style of the `do` notation for monads, but which is desugared by a preprocessor into the operations of the parameterised monad. In modern GHC, this can be replaced with the `RebindableSyntax` extension which desugars the standard `do` notation using any functions in scope named `(>>=)` and

`return`, regardless of their type. The operations of a parameterised monad can therefore usurp the regular monad operations. Thus, the non-idiomatic pre-processed `ixdo` notation can be replaced with idiomatic `do` notation. The same applies to the work of Sackman and Eisenbach (Section 10.3.4) and Imai *et al.* (Section 10.3.5) who also use parameterised monads. Similarly, GHC's rebindable syntax is reused by Orchard and Yoshida with *graded monads* (Section 10.3.6).

10.3.4 An Alternate Approach; Sackman and Eisenbach [15]

In their unpublished manuscript, Sackman and Eisenbach provide an implementation also using a parameterised monad but with quite a different formulation to Pucella and Tov. The encoding of session environments is instead through type-level finite maps from channel names (as types) to session types. This requires significantly more type-level machinery (implemented mostly using classes with functional dependencies), resulting in much more complicated types than Pucella-Tov. However, they provide a parameterised monad `SessionType` for constructing session-type witnesses at the value level (similarly to Neubauer-Thiemann) which is much easier to read and write than the corresponding type-level representation. Session-based computations are then constructed through another parameterised monad called `SessionChain`.

Sackman-Eisenbach represent session types by type-level lists (via constructors `Cons` and `Nil`) of actions given by parametric data types `Send`, `Recv`, `Select`, `Offer`, `Jump`, and (non parametric) `End` similar to the other representations. For Example 2, the recursive session type of the server can be constructed via value-level terms as:

```
(serverSpec, a) = makeSessionType $ do
  a <- newLabel
  let eq = do {recv intW; recv intW; recvSession (send boolW); jump a}
  a .= offer (eq ~|~ end ~|~ BLNil)
  return a
```

This uses the `SessionType` parameterised monad indexed by `TypeState` types which have further indices managing labels and representing session types. The `makeSessionType` function returns a pair of a value capturing the specification `serverSpec` and the component of the type labelled by `a`. Labels are used to associate types to channels and for recursive types, where `newLabel` generates a fresh label bound to `a`. The third line associates to `a` the expected session behaviour: a choice is offered where `offer` takes a list of behaviours

constructed by `~|~` (cons) and `BLNil` (nil). As in Neubauer-Thiemann, `intW` and `boolW` are value witnesses of types. The recursive step is via `jump` on label `a`. The type of `send` illustrates the `SessionType` parameterised monad:

```
send :: (TyList f, TyList fs) => t -> SessionType
   (TypeState n d u (Cons (lab, f) fs))
   (TypeState n d u (Cons (lab, (Cons (Send (Normal, t)) f)) fs)) ()
```

The final parameter to `TypeState` provides a type-level list of labelled session types (themselves lists). In the post-condition, the session type `f` from the head of the list in the pre-condition has `Send` consed onto the front, parameterised by `(Normal, t)` indicating the value type `t`.

The session-type building primitives have computation building counterparts (whose names are prefixed with `s`, *e.g.* `ssend`) returning computations in the `SessionChain` parameterised monad. We elide the details, but show the implementation of the server from Example 2:

```
server = do
  cid <- fork serverChan dual (cons (serverSpec, notDual) nil) client
  c <- createSession serverSpec dual cid
  withChannel c (soffer ((do
       x <- srecv
       y <- srecv
       recvChannel c (\d ->
         withChannel d (do { ssend (x == y); sjump })))
   ~||~ (return ()) ~||~ OfferImplsNil))
```

The session type specification `serverSpec` is linked to computation to enforce linearity via `fork`. Above, `client` refers to the client code which is forked and given a channel whose behaviour is dual to that created locally by `createSession`, specified by `serverSpec`. The `sjump` primitive provides the recursive behaviour but has no target which is implicitly provided by the specification. The `withChannel` primitive "focuses" the computation on a particular channel such that the communication primitives are not parameterised by channels, similar to Neubauer-Thiemann. This has some advantage over Pucella-Tov, which required manual session-context manipulation, though channel variables still cannot be used directly here. Combined with the complicated type encoding, we therefore characterise this approach as the least idiomatic.

It should be noted that since the appearance of their manuscript, the type checking of functional dependencies in GHC has become more strict (particularly with the additional *Coverage Condition* [16, Def. 7]). At the time

of writing, the latest available online implementation of Sackman-Eisenbach fails to type check in multiple places due to the coverage conditions added later to GHC. It is not immediately clear how to remedy this due to their reliance on functional dependencies which do not obey the new coverage condition.

10.3.5 Multi-Channels with Inference; Imai et al. [5]

Imai, Yuen, and Agusa directly extend the Pucella-Tov approach, providing type inference, delegation, and solving the deficiencies with accessing multiple channels. They replace the positional, stack-based approach for multiple channels with a De Bruijn index encoding which is handled implicitly at the type level. For example, `send` has type

```
send :: (Pickup ss n (Send v a), Update ss n a ss', IsEnded ss F)
     => Channel t n -> v -> Session t ss ss' ()
```

Computations are modelled by the parameterised monad `Session` as before, but now pre- and post-condition indices `ss` and `ss'` are type-level lists of session types, rather than a labelled stack. Whilst these structures are isomorphic, the way session types are accessed within the list representation differs considerably.

A channel `Channel t n` has a type-level natural number `n` representing the position of the channel's session type in the list. The constraint `Pickup` above specifies that at the $\texttt{n}^{th}$ position in `ss` is the session type `Send v a`. The constraint `Update` then states that `ss'` is the list of session types produced by replacing the $\texttt{n}^{th}$ position in `ss` with the session type `a`. The rest of the communication primitives follow a similar scheme to the above, generalising Pucella-Tov primitives to work with the De Bruijn indices instead of just the capability at the top of the stack.

A fresh channel can be created by the following combinator:

```
new :: SList ss l => Session t ss (ss:>Bot) (Channel t l)
```

where `l` is the length of the list `ss` as defined by the constraint `SList`, and thus is a fresh variable for the computation.

Using this library leads to highly idiomatic Haskell code, with no additional combinators required for managing the context of session-typed channels. Both examples can be implemented, with code similar to that shown for Pucella-Tov in Section 10.3.3. The one downside of this approach however is that the types, whilst they can be inferred (which is one of the aims

of their work), are complex and difficult to read, let alone write. Relatedly, the type errors can be difficult to understand due to the additional type-level mechanisms for managing the contexts.

10.3.6 Session Types via Effect Types; Orchard and Yoshida [11]

Orchard and Yoshida studied the connection between effect systems and session types. One part of the work showed the encoding of a session-typed π-calculus into a parallel variant of PCF with a general, parameterised effect system. This formal encoding was then combined with an approach for embedding effect systems in Haskell [10] to provide a new implementation of session-typed channels in Haskell. The implementation supports multiple channels in an idiomatic style, delegation, and a restricted form of recursion (affine recursion only).

The embedding of general effect systems in Haskell types is provided by a *graded monad* structure, which generalises monads to type constructors indexed by a type-representation of effect information. This "effect type" has the additional structure of a monoid, encoded using type families. The graded monad structure in Haskell is defined:

```
class Effect (m :: ef -> * -> *) where
   type Unit m :: ef
   type Plus m (f :: ef) (g :: ef) :: ef
   return :: a -> m (Unit m) a
   (>>=) ::  m f a -> (a -> m g b) -> m (Plus m f g) b
```

Thus a value of type `m f a` denotes a computation with effects described by the type index `f` of kind `ef`. The `return` operation lifts a value to a trivially effectful computation, marked with the type `Unit m`. The "bind" operation (>>=) provides the sequential composition of effectful computations, with effect information composed by the type-level binary function `Plus m`. The session type embedding is provided by a graded monad structure for the data type `Process`:

```
data Process (s :: [Map Name Session]) a = Process (IO a)
```

Type indices `s` are finite maps of the form `'[c :-> s, d :-> t, ...]` mapping channel names `c`, `d` to session types `s`, `t`. The `Session` kind is given by a data type (representing a standard grammar of session types) promoted by the *data kinds* extension of GHC to the kind-level.

The `Plus` type operation of the `Process` graded monad takes the *union* of two finite maps and sequentially composes the session types of any channels

that appear in both of the finite maps. This relies on the closed type family feature of GHC to define type-level functions that can match on their types, *e.g.*, to compare types for equality.

The core send and received primitives then have the following types:

```
send  :: Chan c -> t -> Process '[c :-> t :! End] ()
recv  :: Chan c      -> Process '[c :-> t :? End] t
```

In each, the type-index on `Process` gives a singleton finite map from the channel name `c` to the session type. We elide the rest of the combinators. Duality is enforced when a pair of channel endpoints is created by `new`:

```
new :: (Duality env c) => ((Chan (Ch c), Chan (Op c)) -> Process env t)
    -> Process ((env :\ (Op c)) :\ (Ch c)) t
```

where `:\` removes a channel's session type from the environment.

A non-recursive implementation of Example 2 can be defined:

```
server (c :: (Chan (Op "c"))) =
 do l <- recv c
    case l of
     L -> subL $ do
          x <- recv c
          y <- recv c
          k <- chRecv c
          k (\d -> send d (x == y))
     R -> subR $ subEnd c (return ())

client (c :: (Chan (Ch "c"))) = do
 send c L
 subL' c $ do
  send c 42
  send c 53
  new (\(d :: (Chan (Ch "d")), d') ->
     do chSend c d
        x <- recv d'
        print $ "Got: " ++ show x)
```

which are composed by `new (\(c, c') -> client c ‘par‘ server c')`.

One advantage of this approach is that most types are easy to write by hand, with a succinct understandable presentation in terms of the finite maps from channel names to session types. Furthermore, the use of multiple channels is idiomatic, using Haskell's normal variables. The major disadvantage of this approach is that the user must give their own explicit type-level names to the channels, *e.g.*, type signatures like `Chan (Ch "c")` above. For simple examples this is not a burden, but manually managing uniqueness of variables does not scale well.

Furthermore, the approach is brittle due to complex type-level representation and manipulations of finite maps. For example, GHC has difficulty reasoning about the type-level *union* operation (used as `Plus`) when applied to types involving some polymorphism.

10.3.7 GV in Haskell; Lindley and Morris [8]

GV is a session-typed linear functional calculus, proposed by [17], based on the work of Gay and Vasconcelos [3], and adapted further by Lindley and Morris [6]. The GV presented by Lindley and Morris aims at re-use of standard components, defined as an extension of the linear λ-calculus with session-typed communication primitives. This provides a basis for their Haskell implementation by reusing an embedding of the linear λ-calculus into Haskell due to Polakow [12]. Polakow's embedding provides a "tagless final" encoding of the linear-λ calculus (LLC), meaning that terms of LLC are represented by functions of a type class, whose interpretation/implementation can be varied based on the underlying type. Furthermore, the embedding uses higher-order abstract syntax (HOAS) *i.e.*, binders in LLC are represented by Haskell binders.

To represent the linear types notion of context *consumption*, contexts are split in two with judgments of the form: $\Delta_I \setminus \Delta_O \vdash e : A$ with *input context* Δ_I and output context Δ_O which remains after computing e and thus after some parts of Δ_I have been consumed. Contexts come equipped with the notion of a "hole" (written $\Box$) denoting a variable that has been consumed. For example, a linear variable use is typed by $\Delta, x : A, \Delta' \setminus \Delta, \Box, \Delta' \vdash x : A$.

The embedding of this linear type system uses natural numbers to represent variables in judgements. Judgements are represented by types `repr :: Nat -> [Maybe Nat] -> [Maybe Nat] -> * -> *`. Thus, the LLC term representation is a type indexed by four pieces of information: a natural number denoting a fresh name for a new variable, the input context (a list of `Maybe Nat` where `Just n` is a variable and `Nothing` denotes $\Box$), the output context, and the term type.

The core of the embedding for the linear function space fragment, is then given by the `LLC` class, parameterised by a `repr` type:

```
class LLC (repr :: Nat -> [Maybe Nat] -> [Maybe Nat] -> * -> *) where
  llam :: (LVar repr v a -> repr (S v) (Just v ': i) (Box ': o) b)
       -> repr v i o (a -<> b)
  (^) :: reprv v i h (a -<> b) -> repr v h o a -> prepr v i o b
```

where `LVar` represents linear variables, defined as the type `forall v i o . (Consume x i o) => repr v i o a` describing that using a variable leads to its consumption for all input and output contexts `i` and `o`.

The session primitives of GV are added atop the LLC embedding via another tagless final encoding (we elide the primitives for branching):

```
class GV (ch :: * -> *) repr where
 send :: DualS s => repr v i h t -> repr v h o (ch (t <!> s))
                                 -> repr v i o (ch s)
 recv :: DualS s => repr v i o (ch (t <?> s)) -> repr v i o (t * ch s)
 wait :: repr v i o (ch EndIn)                -> repr v i o One
 fork :: DualS s => repr v i o (ch s -<> ch EndOut)
                 -> repr v i o (ch (Dual s))
```

The types involve duality as both a predicate (type constraint) `DualS` and as a type-level function `Dual`.

The approach does not provide recursive sessions so we implement a non-recursive version of Example 1 as:

```
server = llam $ \c ->
  recv c  `bind` (llp $ \x c ->
  recv c  `bind` (llp $ \y c ->
  send (const (==) $$$ x $$$ y) c))
example = fork server `bind` client
```

```
client = llam $ \c ->
  send (const 42) c `bind` (llam $ \c ->
  send (const 53) c `bind` (llam $ \c ->
  recv c    `bind` (llp $ \r c ->
  wait c    `bind` (llz $ ret r))))
```

This approach cleanly separates the notion of linearity from the channel capabilities of session types. The main downside is that application, λ-abstraction, and composition of terms must be mediated by the combinators of the LLC embedding. Therefore, the approach does not support idiomatic Haskell programming.

10.4 Future Direction and Open Problems

The table at the beginning of Section 10.3 (p. 226) indicates that there is no one implementation that provides all desirable features: a session-typed library for communication-safe concurrency with linearity, delegation, multiple-channels, recursion, idiomatic Haskell code, and the ability to easily give session type specifications by hand. Furthermore, none correctly implements duality with respect to recursion (Section 10.3.1).

So far there appears to be a trade-off between these different features. Pucella and Tov provide an idiomatic system with relatively simple types, but require the manual management of the capability stack. The work of Imai *et al.* provides a highly idiomatic system, but the types are hard to manipulate and understand. Orchard and Yoshida provide types that are easy to write, but at the cost of forcing the user to manually manage fresh channel names. Lindley and Morris handle variables idiomatically, but require additional

combinators for application, λ-abstraction and term composition. Sackman and Eisenbach provide session types which are easily specified by-hand with a value witness, but with non-idiomatic code and hard to manipulate types.

One possible solution is to adapt the approach of Orchard and Yoshida with a way to generate fresh channel names at the type-level automatically via a GHC *type checker plugin* (see, *e.g.*, [4]). Alternatively, existential names can be used for fresh names. However, the implementation of type-level finite maps relies on giving an arbitrary ordering to channel names (for the sake of normalisation) which is not possible for existential names. In which case, a type-checker plugin could provide built-in support for finite maps more naturally, rather than using the current (awkward) approach of Orchard and Yoshida.

We have examined the six major session type implementations for Haskell in this chapter. All of them provide static linear checks, leveraging Haskell's flexible type system, but all have some deficiencies; finding a perfectly balanced system remains an open problem.

Acknoweldgements We thank Garrett Morris and the anonymous reviewers for their helpful comments. This work was supported in part by EPSRC grants EP/K011715/1, EP/K034413/1, EP/L00058X/1, EP/M026124/1, and EU project FP7-612985 UpScale.

References

[1] Robert Atkey. Parameterised notions of computation. *Journal of functional programming*, 19(3–4):335–376, 2009.

[2] Giovanni Bernardi, Ornela Dardha, Simon J. Gay, and Dimitrios Kouzapas. On duality relations for session types. In *Trustworthy Global Computing 2014*, pages 51–66, 2014.

[3] Simon J. Gay and Vasco T. Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20(01): 19–50, 2010.

[4] Adam Gundry. A typechecker plugin for units of measure: domain-specific constraint solving in GHC Haskell. In *ACM SIGPLAN Notices*, volume 50, pages 11–22. ACM, 2015.

[5] Keigo Imai, Shoji Yuen, and Kiyoshi Agusa. Session Type Inference in Haskell. In *PLACES*, pages 74–91, 2010.

[6] Sam Lindley and J. Garrett Morris. A semantics for propositions as sessions. In *ESOPb*, pages 560–584. Springer, 2015.

[7] Sam Lindley and J. Garrett Morris. Talking Bananas: Structural Recursion for Session Types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 434–447. ACM, 2016.

[8] Sam Lindley and J Garrett Morris. Embedding session types in haskell. In *Proceedings of the 9th International Symposium on Haskell*, pages 133–145. ACM, 2016.

[9] Matthias Neubauer and Peter Thiemann. An Implementation of Session Types. In *PADL*, volume 3057 of *LNCS*, pages 56–70. Springer, 2004.

[10] Dominic Orchard and Tomas Petricek. Embedding effect systems in Haskell. *ACM SIGPLAN Notices*, 49(12):13–24, 2015.

[11] Dominic Orchard and Nobuko Yoshida. Effects as Sessions, Sessions as Effects. *ACM SIGPLAN Notices*, 51(1):568–581, 2016.

[12] Jeff Polakow. Embedding a Full Linear Lambda Calculus in Haskell. *ACM SIGPLAN Notices*, 50(12):177–188, 2016.

[13] Riccardo Pucella and Jesse A. Tov. Haskell Session Types with (Almost) no Class. In *Proc. of Haskell Symposium '08*, pages 25–36. ACM, 2008. ISBN 978-1-60558-064-7.

[14] John H. Reppy. CML: A Higher-Order Concurrent Language. In *PLDI*, pages 293–305, 1991.

[15] Matthew Sackman and Susan Eisenbach. Session Types in Haskell (Updating Message Passing for the 21st Century), 2008. Technical report, Imperial College London.

[16] Martin Sulzmann, Gregory J Duck, Simon Peyton-Jones, and Peter J Stuckey. Understanding functional dependencies via constraint handling rules. *Journal of Functional Programming*, 17(01):83–129, 2007.

[17] Philip Wadler. Propositions as sessions. *Journal of Functional Programming*, 24(2–3):384–418, 2014.

[18] Nobuko Yoshida and Vasco Thudichum Vasconcelos. Language Primi tives and Type Discipline for Structured Communication-Based Programming Revisited: Two Systems for Higher-Order Session Communication. *Electr. Notes Theor. Comput. Sci.*, 171(4):73–93, 2007.