# Embedding session types in Haskell

**Citation for published version:**
Lindley, S & Morris, JG 2016, Embedding session types in Haskell. in *Haskell 2016 Proceedings of the 9th International Symposium on Haskell.* ACM SIGPLAN Notices, no. 12, vol. 51, ACM, pp. 133-145, 2016 Proceedings of the 9th International Symposium on Haskell, Nara, Japan, 18/09/16.
https://doi.org/10.1145/2976002.2976018

**Digital Object Identifier (DOI):**
10.1145/2976002.2976018



**Published In:**
Haskell 2016 Proceedings of the 9th International Symposium on Haskell

# Embedding Session Types in Haskell

Sam Lindley    J. Garrett Morris

The University of Edinburgh, UK

{Sam.Lindley,Garrett.Morris}@ed.ac.uk

## Abstract

We present a novel embedding of session-typed concurrency in Haskell. We extend an existing HOAS embedding of linear $\lambda$-calculus with a set of core session-typed primitives, using indexed type families to express the constraints of the session typing discipline. We give two interpretations of our embedding, one in terms of GHC's built-in concurrency and another in terms of purely functional continuations. Our safety guarantees, including deadlock freedom, are assured statically and introduce no additional runtime overhead.

***Categories and Subject Descriptors*** D.1.1 [*Programming Techniques*]: Applicative (functional) programming; D.1.3 [*Programming Techniques*]: Concurrent programming

***Keywords*** linear types, session types, embedded languages

## 1. Introduction

Many communication protocols specify not just the types or formats of data or commands in the protocol, but also place restrictions on the order in which data is to be communicated. For example, the simple mail transfer protocol (SMTP) not only includes commands to specify the sender, recipients, and contents of an email message, but also requires that the sender command precede the recipient commands, which must in turn precede the commands giving the message body. Session types [6, 7, 20] capture such protocols in the types of communication channels. Session types have two distinguishing features. First, the endpoints of a channel must be given dual types: if one process expects to send a value along some channel, the process on the other end of the channel must expect to receive it. Second, session types must evolve over the course of a computation to prevent processes from repeating or skipping steps of the protocol.

Much of the existing work presents session types in the context of core concurrency-focused calculi (frequently based on either $\pi$-calculus or linear $\lambda$-calculus). Such calculi provide a holistic view of session types, integrating aspects of their syntax, the distinguishing aspects of the types themselves (such as duality), and their concurrent interpretations. However, typically they do not address how session types can be integrated into existing languages or the relationship between the concurrency expressed using session typing and that provided by existing concurrent primitives. We have developed a core session-typed functional calculus called GV [11, 12]. GV has strong connections to classical linear logic; consequently, its type system guarantees deadlock freedom in addition to typical safety properties. Our development of GV is also intended to be modular. We build on a standard linear $\lambda$-calculus, and attempt to minimize the number of concurrent features, preferring to express concurrent features in terms of $\lambda$-calculus constructs when possible. GV's metatheory is developed modularly as well; for example, this allows us to show that the addition of several non-logical features does not compromize GV's deadlock freedom, even though the extended calculus no longer enjoys a tight correspondence with classical linear logic.

This paper presents a parameterized tagless embedding [1, 3] of GV in Haskell and two implementations of that embedding. (We will use the term *parameterized tagless* or just *tagless* in preference to *finally tagless* or *tagless final*.) We begin by presenting the embedding of GV, building on Polakow's [17] embedding of linear $\lambda$-calculus in Haskell. In doing so, we demonstrate the generality of Polakow's embedding: first, we are able to extend his core calculus with GV's concurrent primitives, and second, we are able to build a monadic interpretation of his embedding to support computations with side effects. Then, we present two implementations of our embedding, one based on the concurrent primitives in Haskell's IO monad and another that expresses concurrency using continuations. The former shows that this approach has practical applicability. We are able to wrap existing concurrent primitives with new type information, providing additional static safety guarantees without introducing runtime cost. The latter validates that our primitives also have a purely functional interpretation, following the formal semantics of GV. It also provides general insight into parameterized tagless embeddings and translations between them; in particular, while we are able to implement GV in terms of a more explicit language, Polarized GV, such an implementation requires limitations on the modularity of our source language.

The paper proceeds as follows. We review session types and the role of linearity in session typing (§2), and Polakow's embedding of linear $\lambda$-calculus in Haskell (§3). In the course of the latter, we introduce our monadic interpretation. We introduce the core GV calculus and give its semantics (§4). We present two implementations of GV. The first uses the IO monad, and demonstrates that GV's static guarantees need introduce no runtime overhead. We also show extensions of this embedding that increase its expressivity (at the cost of some of its static guarantees), demonstrating GV's modular nature. The second realizes the CPS semantics of GV in the continuation monad. The CPS semantics is non-parametric in that the translation of some term forms depends on the type at which they are used. To restore parametricity, we introduce a polarized version of the calculus (§6). We then show that we can implement the original language in terms of its polarized variant (§7). These implementations show that GV concurrency can be used in a purely functional setting (or other setting in which using IO would be undesirable, such as STM), and shows that our embeddings are are suitable for metaprogramming. We conclude by discussing future work and the difficulties we discovered in the course of our implementation (§8).

This document is literate Haskell. An extended (albeit illiterate) version of the code in this paper is available at the following URL:

`http://github.com/jgbm/gvinhs/`

## 2. Session Types and Linearity

Session types, originally proposed by Honda [6], are an approach to statically verifying communicating concurrent programs. Session types specify both the format (i.e., data type) and ordering of messages along channels. As a simple example, we consider the client-side protocol for a concurrent calculator. The session type for a single binary operation on integers might be as follows:

$$Int\ \langle ! \rangle\ (Int\ \langle ! \rangle\ (Int\ \langle ? \rangle\ \mathsf{End}_?))$$

The type $T\ \langle ! \rangle\ S$ means to send a $T$ and then continue as $S$, the type $T\ \langle ? \rangle\ S$ means to receive a $T$ and then continue as $S$, and the type $\mathsf{End}_?$ means to wait for the channel to close. The whole type means send two integers, receive an integer in return, and then wait for communication to end. We assume that $\langle ! \rangle$ and $\langle ? \rangle$ group to the right and omit parentheses accordingly. Session types also include constructs corresponding to selecting and offering a choice. For example, our calculator might offer a choice between one binary and one unary operation. The client-side view of its protocol would then be captured by the following session type:

$$(Int\ \langle ! \rangle\ Int\ \langle ! \rangle\ Int\ \langle ? \rangle\ \mathsf{End}_?)\ \langle + \!\! + \rangle\ (Int\ \langle ! \rangle\ Int\ \langle ? \rangle\ \mathsf{End}_?)$$

The type $S_1\ \langle + \!\! + \rangle\ S_2$ means to select between $S_1$ and $S_2$.

One important feature of session types is duality: if the session type above represents the client's view of a communication, the server must have dual behavior. The session type of the corresponding server is as follows:

$$(Int\ \langle ? \rangle\ Int\ \langle ? \rangle\ Int\ \langle ! \rangle\ \mathsf{End}_!)\ \langle \&\& \rangle\ (Int\ \langle ? \rangle\ Int\ \langle ! \rangle\ \mathsf{End}_!)$$

The offer construct $S_1\ \langle \&\& \rangle\ S_2$ on the server is dual to the selection construct $S_1\ \langle + \!\! + \rangle\ S_2$ in the client: the server must be able to provide either behavior, while the client only has to select one of the offered behaviors. Unlike many presentations of session types, but inspired by their logical connections, our session types represent closing of channels explicitly. The type $\mathsf{End}_!$ means to close the channel, while $\mathsf{End}_?$ means to wait for the channel to close.

Functional session-typed calculi typically present communication primitives as transforming channels of one session type into channels of another session type. For example, the sending primitive might have a type like $T \to (T\ \langle ! \rangle\ S) \to S$, reflecting that it consumes a channel that expects an output to occur, and returns a new channel without that expectation (i.e., with the expectation satisfied). However, this in itself is not enough to assure that protocols are followed: a process could reuse the original channel (with type $T\ \langle ! \rangle\ S$) to send unexpected $T$ values, or could discard channels without performing the expected communications. To rule out these possibilities, session-typed calculi either rely on linear type systems [5, 22] or on some amount of dynamic checking [14, 19]. GV is a linear calculus: its type system excludes duplication or discarding of variables, and thus statically assures *session fidelity*, that is, that all communication along a channel satisfies the protocol specified by its session type.

## 3. Linear $\lambda$-Calculus, Monadically

GV is based on extending a standard linear $\lambda$-calculus with a small set of concurrent primitives. This simplifies the metatheory of GV, by relying on standard metatheoretic results for (linear) $\lambda$-calculi. It is also beneficial for embedding GV in Haskell. It allows us to build on an existing embedding of linear $\lambda$-calculus in Haskell, and thus to distinguish those aspects of the language unique to session typing from those aspects shared by other linear $\lambda$-calculi.

We build on Polakow's [17] embedding of linear $\lambda$-calculus in Haskell. This is a parameterized tagless embedding, using higher-order abstract syntax (HOAS) to account for the treatment of binders. We will give a brief overview of this embedding, and then show how it can be given a monadic interpretation. We refer readers to Polakow [17] for a full description of the embedding and the required type-level machinery.

Tagless embeddings use terms of the meta language to embed terms of an object language. Parameterizing over the concrete representation of an object term, for instance using type classes, allows the same term to be given multiple interpretations. A canonical example is a parameterized tagless embedding of simply-typed lambda calculus.

```
class Exp repr where
  lam :: (repr a → repr b) → repr (a → b)
  app :: repr (a → b) → repr a → repr b
```

A term of type $repr\ \ a$ represents the type-correct construction of a $\lambda$-term of type $a$; each type constructor $repr$ denotes a particular concrete interpretation of simply-typed $\lambda$-calculus. Because Haskell's type system includes that of simply-typed $\lambda$-calculus, there is a natural correspondence between the typing of terms of the meta language and the typing of terms of the object language. The same is not true for embedding linear $\lambda$-calculus. For reference, we give typing rules for variables, abstraction, and application in linear $\lambda$-calculus.

$$\frac{}{x : A \vdash x : A} \qquad \frac{\Delta, x : A \vdash M : B}{\Delta \vdash \lambda x.M : A \multimap B}$$

$$\frac{\Delta \vdash M : A \multimap B \quad \Delta' \vdash N : A}{\Delta \uplus \Delta' \vdash M \mathbin{\hat{}} N : B}$$

The variable rule insists that there can be no other variables in the environment, while the application rule divides its typing environment among its hypotheses. (We write $\Delta \uplus \Delta'$ to indicate that $\Delta$ and $\Delta'$ must have disjoint domains.) These do not correspond to the treatment of variables and functions in Haskell, and so we cannot immediately treat a Haskell term (of a type like $repr\ \ a \to repr\ \ b$) as a linear $\lambda$-calculus term of type $repr\ (a \multimap b)$.

To address this problem, Polakow uses representation types which make explicit the linear variable environment as well as the result type. Doing so allows him to capture the treatment of linear assumptions in the types of the term constructors, and thus to define a HOAS embedding of type-correct linear $\lambda$-calculus. He gives an alternative presentation of the typing rules for linear $\lambda$-calculus, using judgments of the form

$$\Gamma; \Delta \setminus \Delta' \vdash M : A$$

Intuitively, $\Delta$ contains the assumptions available before checking $M$, while $\Delta'$ contains the assumptions remaining after checking $M$; their difference, then, reflects the assumptions used by $M$. Once a variable has been consumed it is replaced by the special assumption $\square$, rather than being removed from the type environment; thus maintaining the invariant that the $\Delta$ and $\Delta'$ always have the same length. Finally, $\Gamma$ captures an unrestricted (i.e., non-linear) environment, allowing the use of both linear and non-linear types in linear $\lambda$-calculus terms. Figure 1 gives the linear $\lambda$-calculus typing rules in this form. The rules include linear and intuitionistic abstraction and application ($A \multimap B$ and $A \twoheadrightarrow B$), linear pairs ($A \otimes B$), linear sums ($A \oplus B$), and the ! modality, which can be used to

$$\frac{}{\Gamma;\Delta,x:A,\Delta'\setminus\Delta,\Box,\Delta'\vdash x:A}\qquad\frac{\Gamma;\Delta,x:A\setminus\Delta',\Box\vdash M:B}{\Gamma;\Delta\setminus\Delta'\vdash\lambda x.M:A\multimap B}\qquad\frac{\Gamma;\Delta\setminus\Delta'\vdash M:A\multimap B\quad\Gamma;\Delta'\setminus\Delta''\vdash N:A}{\Gamma;\Delta\setminus\Delta''\vdash M\,\hat{}\,N:B}$$

$$\frac{\Gamma;\Delta\setminus\Delta'\vdash M:A\quad\Gamma;\Delta'\setminus\Delta''\vdash N:B}{\Gamma;\Delta\setminus\Delta''\vdash(M,N):A\otimes B}\qquad\frac{\Gamma;\Delta\setminus\Delta'\vdash M:A\otimes B\quad\Gamma;\Delta',x:A,y:B\setminus\Delta'',\Box,\Box\vdash N:C}{\Gamma;\Delta\setminus\Delta''\vdash\textbf{let}\ (x,y)=M\ \textbf{in}\ N:C}$$

$$\frac{}{\Gamma;\Delta\setminus\Delta\vdash():\mathbf{1}}\qquad\frac{\Gamma;\Delta\setminus\Delta'\vdash M:\mathbf{1}\quad\Gamma;\Delta'\setminus\Delta''\vdash N:A}{\Gamma;\Delta\setminus\Delta''\vdash\textbf{let}\ ()=M\ \textbf{in}\ N:A}\qquad\frac{\Gamma;\Delta\setminus\Delta'\vdash M:A}{\Gamma;\Delta\setminus\Delta'\vdash\mathsf{inl}\,M:A\oplus B}\qquad\frac{\Gamma;\Delta\setminus\Delta'\vdash M:B}{\Gamma;\Delta\setminus\Delta'\vdash\mathsf{inr}\,M:A\oplus B}$$

$$\frac{\begin{array}{c}\Gamma;\Delta\setminus\Delta'\vdash M:A\oplus B\quad\Gamma;\Delta',x:A\setminus\Delta'',\Box\vdash N_1:C\\\Gamma;\Delta',y:B\setminus\Delta'',\Box\vdash N_2:C\end{array}}{\Gamma;\Delta\setminus\Delta''\vdash\textbf{case}\ M\ \textbf{of}\ \mathsf{inl}\,x\mapsto N_1\mid\mathsf{inr}\,y\mapsto N_2:C}\qquad\frac{\Gamma;\cdot\setminus\cdot\vdash M:A}{\Gamma;\Delta\setminus\Delta\vdash{!M}:{!A}}\qquad\frac{\begin{array}{c}\Gamma;\Delta\setminus\Delta'\vdash M:{!A}\\\Gamma,x:A;\Delta'\setminus\Delta''\vdash N:B\end{array}}{\Gamma;\Delta\setminus\Delta''\vdash\textbf{let}\ {!x}=M\ \textbf{in}\ N:B}$$

$$\frac{}{\Gamma,x:A;\Delta\setminus\Delta\vdash x:A}\qquad\frac{\Gamma,x:A;\Delta\setminus\Delta'\vdash M:B}{\Gamma;\Delta\setminus\Delta'\vdash\lambda x.M:A\twoheadrightarrow B}\qquad\frac{\Gamma;\Delta\setminus\Delta'\vdash M:A\twoheadrightarrow B\quad\Gamma;\cdot\setminus\cdot\vdash N:A}{\Gamma;\Delta\setminus\Delta'\vdash M\,N:B}$$

**Figure 1:** Linear $\lambda$-calculus typing rules.

move between the linear and unrestricted contexts. We have omitted several constructs included in Polakow's embedding, namely the additive sum $A\ \&\ B$ and its unit $\top$. The treatment of $\top$ adds significant complication to the overall type system (and thus to the embedding), as it can consume arbitrary linear assumptions. As we have no use for these constructs in our embedding of GV, we chose a simpler type system.

We now review Polakow's HOAS embedding of this type system in Haskell. We begin by defining the linear types:

```
newtype a ⊸ b  = Lolli {unLolli :: a → b}
data a ⊗ b     = Tensor a b
data One       = One
data a ⊕ b     = Inl a | Inr b
newtype a ↠ b  = Arrow {unArrow :: a → b}
newtype Bang a = Bang {unBang :: a}
infixr 5 ⊸, ↠
```

Note that $\twoheadrightarrow$ is the intuitionistic function space: $a \twoheadrightarrow b$ is isomorphic to Bang $a \multimap b$.

Next, we present the encodings of terms, as the methods of a class LLC of interpretations of linear $\lambda$-calculus. The characterization of terms includes not just their types, as in standard tagless embeddings, but also captures the linear environment. Polakow represents the linear environment by (type-level) lists of type Maybe Nat where Nat is a standard Peano encoding of the natural numbers.

```
data Nat = Z | S Nat
```

Each entry in the list represents the presence of a particular variable in the environment, with $\Box$ denoted by Nothing. As the types of terms are already captured in the encoding, the encoding of the environment can omit them. The representation is also parameterized by a counter $v$ used to generate fresh naturals.

```
class LLC (repr :: Nat → [Maybe Nat] → [Maybe Nat]
                  → ⋆ → ⋆) where
  llam :: (LVar repr v a →
           repr (S v) (Just v : i) (Nothing : o) b)
        → repr v i o (a ⊸ b)
  (^) :: repr v i h (a ⊸ b) → repr v h o a → repr v i o b
```

Linear application is a simple example of the encoding. The $(\hat{})$ method takes two terms, one of type $a \multimap b$ and one of type $a$, threading the initial environment through the types of the terms. The result term of of type $b$. The fresh index $v$ is unused as application does not introduce binders. Linear abstraction demonstrates the treatment of binders. The argument is a function from a linear variable (of type LVar $repr\ v\ a$) to a term of type $b$, which is required to have used the new variable. Note that binders in the subterm will be numbered from S $v$. We will return to the definition of the variable type LVar shortly. Other linear term forms are defined similarly.

```
(⊗) :: repr v i h a → repr v h o b → repr v i o (a ⊗ b)
letStar :: repr v i h (a ⊗ b)
        → (LVar repr v a → LVar repr (S v) b →
             repr (S (S v))
                  (Just v : Just (S v) : h)
                  (Nothing : Nothing : o)
                  c)
        → repr v i o c
one :: repr v i i One
letOne :: repr v i h One → repr v h o a → repr v i o a
inl :: repr v i o a → repr v i o (a ⊕ b)
inr :: repr v i o b → repr v i o (a ⊕ b)
letPlus :: repr v i h (a ⊕ b)
        → ( LVar repr v a →
              repr (S v) (Just v : h) (Nothing : o) c)
        → ( LVar repr v b →
              repr (S v) (Just v : h) (Nothing : o) c)
        → repr v i o c
```

The treatment of unrestricted terms is similar. The type UVar $repr\ a$ represents an unrestricted variable of type $a$. In the rules for (\$\$) and bang, we require that the subterm use no linear assumptions.

```
ilam :: (UVar repr a → repr v i o b)
     → repr v i o (a ↠ b)
($$) :: repr v i o (a ↠ b) → repr v o o a
     → repr v i o b
bang :: repr v i i a → repr v i i (Bang a)
letBang :: repr v i h (Bang a)
        → (UVar repr a → repr v h o b)
        → repr v i o b
```

We return to the encoding of variables. A linear variable LVar $repr\ v\ a$ for representation $repr$ with index $v$ and type $a$ is a term of type $a$ that replaces Just $v$ with Nothing in its environment:

```
type LVar repr (v :: Nat) a =
  ∀(w :: Nat) (i :: [Maybe Nat]) (o :: [Maybe Nat]).
    Consume v i o ⇒ repr w i o a
```

The type class Consume implements the details of the environment transformation; we omit it here, for space reasons, but its definition can be found in Polakow [17]. An unrestricted variable UVar $repr\ a$ has no effect on the linear environment:

```
type UVar repr a = ∀(v :: Nat) (i :: [Maybe Nat]). repr v i i a
```

Polakow gives a representation of linear $\lambda$-calculus terms of type $a$ as Haskell terms of the same type; this shows that the HOAS encoding need introduce no run-time overhead. However, it is limited to expressing pure computations (as the representation is in terms of pure Haskell terms). We seek a notion of side effects expressive enough to capture GV concurrency, but without requiring changes to the signature of LLC. Our solution is to define a monadic representation for linear $\lambda$-calculus terms:

```
newtype RM (m :: ⋆ → ⋆)
           (v :: Nat)
           (i :: [Maybe Nat])
           (o :: [Maybe Nat])
           (a :: ⋆)
  = RM {unRM :: m (Mon a m)}

eval :: RM m v '[] '[] a → m (Mon a m)
eval = unRM
```

The representation type RM is parameterized by a monad $m$, so RM $m\ v\ i\ o\ a$ represents the linear $\lambda$-calculus terms of type $a$. However, unlike Polakow's representation, we cannot define RM in an entirely uniform way: the representation of an $a \multimap b$ function cannot simply be $m\ (a \to b)$, but must instead be $m\ (a \to m\ b)$. We account for this by introducing a type family Mon, which maps from the linear type constructors (such as $\multimap$ and $*$) to their monadic interpretations (again parameterized by the particular monad $m$).

```
type family Mon (t :: ⋆) :: (⋆ → ⋆) → ⋆
```

We can then introduce monadic versions of each of the linear type constructors.

```
newtype MFun (a :: (⋆ → ⋆) → ⋆) (b :: (⋆ → ⋆) → ⋆)
             (m :: ⋆ → ⋆) =
  MFun {unMFun :: a m → m (b m)}
type instance Mon (a ⊸ b) = MFun (Mon a) (Mon b)
type instance Mon (a ↠ b) = MFun (Mon a) (Mon b)

newtype MProd a b (m :: ⋆ → ⋆) =
  MProd {unMProd :: (a m, b m)}
type instance Mon (a ⊗ b) = MProd (Mon a) (Mon b)

data MOne (m :: ⋆ → ⋆) = MOne
type instance Mon One = MOne

newtype MSum a b (m :: ⋆ → ⋆) =
  MSum {unMSum :: Either (a m) (b m)}
type instance Mon (a ⊕ b) = MSum (Mon a) (Mon b)

type instance Mon (Bang a) = Mon a
```

Finally, we can give the LLC instance for RM $m$; the methods are straightforward liftings of the corresponding methods in the non-monadic case.

```
instance Monad m ⇒ LLC (RM m) where
  llam f =
    RM $ return $ MFun $ λx → unRM $ f $ RM (return x)
  f ⌢ x = RM $ do f′ ← unRM f
                  x′ ← unRM x
                  unMFun f′ x′
  x ⊗ y = RM $ do x′ ← unRM x
                  y′ ← unRM y
                  return (MProd (x′, y′))
  letStar xy f = RM $ unRM xy ≫= unRM ∘ f′
    where f′ (MProd (x, y)) = f (RM $ return x)
                                (RM $ return y)
  one = RM $ return MOne
  letOne x y = RM (unRM x ≫= const (unRM y))
```

The remainder of the cases are similarly routine; the details can be found in the extended version online.

Our construction of a monadic interpretation of linear $\lambda$-calculus is (unsurprisingly) similar to the construction Carette et al. [3] for a CPS interpretations of their tagless embedding of $\lambda$-calculus. The primary difference is in the details of our treatment of functions. Our introduction of the Mon type family follows from their observation that the treatment of functions and other values of base type cannot be uniform. However, unlike Carette et al., we do not limit the domain of types that can appear in our representations; in particular, we will later want to extend the grammar of linear types with session types. Instead, we explicitly wrap values of base type, and extend the linear calculus to permit application of base functions to base values.

```
newtype Base a = Base {unBase :: a}
class LLC (repr :: Nat → [Maybe Nat] → [Maybe Nat]
                   → ⋆ → ⋆) where
  ...
  constant :: a → repr v i i (Base a)
  ($$$) :: repr v i h (Base (a → b))
        → repr v h o (Base a)
        → repr v i o (Base b)
```

The interpretation of these methods in the monadic representation is straightforward.

```
newtype MBase a m = MBase {unMBase :: a}
type instance Mon (Base a) = MBase a
instance Monad m ⇒ LLC (RM m) where
  ...
  constant x = RM (return (MBase x))
  RM m $$$ RM n = RM $
    do MBase f ← m
       MBase x ← n
       return $ MBase (f x)
```

# 4. The GV Calculus

GV [11, 12] draws on a line of research on session types in functional languages. Vasconcelos et al. [22] and Gay and Vasconcelos [5] initially explored the integration of session types and functional programming. Building on work by Caires and Pfenning [2], Wadler [23] presented a correspondence between classical linear logic (CLL) and a session-typed process calculus; he also demonstrated a type-preserving translation from a simple functional language (inspired by the work of Gay and Vasconcelos) and his process calculus. Drawing on its correspondence to CLL, Wadler's calculi guarantee deadlock freedom as well as session fidelity. GV is based on Wadler's functional calculus; in our work, we have focused on distinguishing its functional and concurrent features, have given it a direct semantics (with semantics-preserving translations to and from Wadler's CLL-based process calculus), and have shown extensions of GV that increase its expressivity without (necessarily) giving up its metatheoretic properties. This section will introduce GV's types, show how they extend linear $\lambda$-calculus, and present our tagless embedding of GV.

GV session types $S$ are given by the following grammar, in which $T$ can range over arbitrary (Haskell) types:

$$\begin{aligned} S ::= {} & T\ \langle ? \rangle\ S \ \mid\ S_1\ \langle \&\& \rangle\ S_2 \ \mid\ \mathsf{End}_? \\ \mid {} & T\ \langle ! \rangle\ S \ \mid\ S_1\ \langle + \rangle\ S_2 \ \mid\ \mathsf{End}_! \end{aligned}$$

A session type on a channel captures the expected communication along that channel. Types $T\ \langle ?\rangle\ S$ and $T\ \langle !\rangle\ S$ denote receiving and sending values of type $T$, with the remaining communication captured by $S$. Types $S_1\ \langle \&\& \rangle\ S_2$ and $S_1\ \langle +\!\!+ \rangle\ S_2$ reflect offering and making a choice between expectations $S$ and $S'$. Finally, $\mathsf{End}_?$ and $\mathsf{End}_!$ reflect closing a channel (where the $\mathsf{End}_?$ endpoint will wait for the $\mathsf{End}_!$ endpoint to close). We introduce Haskell types corresponding to each session type constructor; as we intend them to be used as indices in the representation of GV, we do not introduce data constructors for these types.

```
data t ⟨?⟩ s; data s1 ⟨&&⟩ s2; data End?
data t ⟨!⟩ s; data s1 ⟨++⟩ s2; data End!
```

We have intentionally chosen not to define session types by data type promotion, so that the grammar of session types admits further extensions. We will take advantage of this openness later when we introduce a notion of polarized session types (§6); our previous work [11, 12], discusses extending GV session types with polymorphism, unrestricted channels, and recursion. A central feature of session types is duality: if the process on one end of a channel expects to send a value of type $T$, the process on the other end should expect to receive a value of type $T$. We write $\overline{S}$ to denote the dual of session type $S$, defined as follows:

$$
\begin{array}{cc}
\overline{T\ \langle ?\rangle\ S} = T\ \langle !\rangle\ \overline{S} & \overline{T\ \langle !\rangle\ S} = T\ \langle ?\rangle\ \overline{S} \\
\overline{S_1\ \langle \&\& \rangle\ S_2} = \overline{S_1}\ \langle +\!\!+ \rangle\ \overline{S_2} & \overline{S_1\ \langle +\!\!+ \rangle\ S_2} = \overline{S_1}\ \langle \&\& \rangle\ \overline{S_2} \\
\overline{\mathsf{End}_?} = \mathsf{End}_! & \overline{\mathsf{End}_!} = \mathsf{End}_?
\end{array}
$$

We realize duality directly in Haskell, using an indexed type family.

```
type family Dual s                :: ⋆
type instance Dual (t ⟨?⟩ s)      = t ⟨!⟩ Dual s
type instance Dual (t ⟨!⟩ s)      = t ⟨?⟩ Dual s
type instance Dual (s1 ⟨&&⟩ s2)   = Dual s1 ⟨++⟩ Dual s2
type instance Dual (s1 ⟨++⟩ s2)   = Dual s1 ⟨&&⟩ Dual s2
type instance Dual End?           = End!
type instance Dual End!           = End?
```

We also introduce a type class characterizing session types, capturing that duality for session types must be involutive.

```
class (Dual (Dual s)∼s) ⇒ Session s
instance Session s ⇒                        Session (t ⟨?⟩ s)
instance Session s ⇒                        Session (t ⟨!⟩ s)
instance                                    Session End?
instance                                    Session End!
instance (Session s1, Session s2) ⇒         Session (s1 ⟨&&⟩ s2)
instance (Session s1, Session s2) ⇒         Session (s1 ⟨++⟩ s2)

type DualSession (s :: ⋆) = (Session s, Session (Dual s))
```

Our session types differ from Honda's original specification [6] only in the treatment of closed channels: he provides a single, self-dual session type End which imposes no expectations on processes, where we require an explicit channel-closing synchronization. This change stems from Wadler's identification of session types with the propositions of CLL: linear logic has no self-dual proposition to stand in for End.

The concurrent portion of GV is defined by a collection of polymorphic constants, corresponding to the introduction and elimination of session types:

$$
\begin{array}{r}
\mathsf{fork} :: (S \multimap \mathsf{End}_!) \multimap \overline{S} \\
\mathsf{send} :: T \multimap (T\ \langle !\rangle\ S) \multimap S \\
\mathsf{recv} :: (T\ \langle ?\rangle\ S) \multimap T \otimes S \\
\mathsf{wait} :: \mathsf{End}_? \multimap \mathbf{1} \\
\mathsf{chooseLeft} :: (S_1\ \langle +\!\!+ \rangle\ S_2) \multimap S_1 \\
\mathsf{chooseRight} :: (S_1\ \langle +\!\!+ \rangle\ S_2) \multimap S_2 \\
\mathsf{offer} :: (S_1\ \langle \&\& \rangle\ S_2) \multimap (S_1 \multimap T) \multimap (S_2 \multimap T) \multimap T
\end{array}
$$

The only introduction form for session types is fork; the remaining operations eliminate session types by performing communication. The operation fork $f$ creates a fresh channel, forks a new thread which invokes $f$ with one endpoint of the channel, and returns the other endpoint of the channel. The operation send $v\ x$ sends value $v$ through endpoint $x$, returning the updated endpoint. Correspondingly, the operation recv $x$ receives a value through endpoint $x$, returning a pair of the value and the updated endpoint. The operation wait $x$ synchronizes on the endpoint $x$, waiting for the corresponding forked thread to terminate and hence close the other end of the channel. The operations chooseLeft $x$ and chooseRight $x$ each make a choice on endpoint $x$. Correspondingly, the operation offer $x\ l\ r$ receives a choice along endpoint $x$ and proceeds accordingly with either $l$ or $r$ applied to the updated endpoint.

We define a class of GV representations, extending our class of linear $\lambda$-calculus representations.

```
class GV (ch :: ⋆ → ⋆)
         (repr :: Nat → [Maybe Nat] → [Maybe Nat]
                  → ⋆ → ⋆) | repr → ch where
  send :: DualSession s
       ⇒ repr v i h t
       → repr v h o (ch (t ⟨!⟩ s))
       → repr v i o (ch s)
  recv :: DualSession s
       ⇒ repr v i o (ch (t ⟨?⟩ s))
       → repr v i o (t ⊗ ch s)
```

An instance of GV fixes both a representation type $repr$ and a channel type constructor $ch$, parameterized by session types. We require that the representation type determine the channel type. The majority of the method signatures are straightforward translations of their type signatures above. The DualSession constraint for the continuation type $s$ is sufficient to assure that the initial types $(t\,\langle !\rangle\, s$ and $t\ \langle ?\rangle\ s)$ are session types as well.

```
  fork :: DualSession s
       ⇒ repr v i o (ch s ⊸ ch End!)
       → repr v i o (ch (Dual s))
  wait :: repr v i o (ch End?)
       → repr v i o One
```

The fork primitive both constructs a new channel (before calling its argument function) and closes the $\mathsf{End}_!$-typed channel its argument returns. The process holding the other endpoint must wait for the channel to close. The signatures and interpretation of the choice constants is unsurprising.

```
  chooseLeft  :: (DualSession s1, DualSession s2)
              ⇒ repr v i o (ch (s1 ⟨++⟩ s2))
              → repr v i o (ch s1)
  chooseRight :: (DualSession s1, DualSession s2)
              ⇒ repr v i o (ch (s1 ⟨++⟩ s2))
              → repr v i o (ch s2)
  offer       :: (DualSession s1, DualSession s2)
              ⇒ repr v i h (ch (s1 ⟨&&⟩ s2))
              → repr v h o (ch s1 ⊸ t)
              → repr v h o (ch s2 ⊸ t)
              → repr v i o t
```

We present a short example of a GV program embedded in Haskell, implementing a simple concurrent calculator. First, we define a process that receives two integers along a channel $c$, and returns their product along the same channel:

```
multiplier = defnGV $ llam $ λc → recv c ‘bind‘ (llp $ λx c →
                                  recv c ‘bind‘ (llp $ λy c →
                                  send (times ⌢ x ⌢ y) c))
```

The times function lifts Haskell multiplication to apply to linear terms; it has the following type signature.

```
times :: (Num b, LLC repr) ⇒
  repr v o o (Bang (Base b) ⊸ Bang (Base b) ⊸ Bang (Base b))
```

The Base constructors lift Haskell types to linear types (as discussed in the previous section), while the Bang constructors are necessary because we have no guarantee that the Haskell function $(\star)$ uses its arguments linearly. The implementation of times is entirely unsurprising. The bind, llp, and llz functions allow us to write GV code in a logical order and simplify the plumbing of channels.

```
bind e f = f ^ e
ret e    = e
llp f    = llam (λp → letStar p f)
llz f    = llam (λz → letOne z f)
```

The defnGV function assures that the term gives rise to no unsatisfiable constraints (which would indicate type errors), equivalently to the defn function in Polakow's embedding.

```
type DefnGV ch a = ∀repr i v.
  (LLC repr, GV ch repr) ⇒ repr v i i a
defnGV :: DefnGV ch a → DefnGV ch a
defnGV x = x
```

We can use multiplier in the context of a larger process, which offers both multiplication and negation behaviors:

```
negater = defnGV $ llam $ λc →
  recv c `bind` (llp $ λx c →
  send (times ^ (bang (constant (−1))) ^ x) c)
calculator = defnGV $ llam $ λc → offer c multiplier negater
```

Finally, we can use the calculator to perform a simple arithmetic operation.

```
answer =
  defnGV $ fork calculator `bind` (llam $ λc →
           chooseLeft c   `bind` (llam $ λc →
           send (bang (constant 6)) c
                          `bind` (llam $ λc →
           send (bang (constant 7)) c
                          `bind` (llam $ λc →
           recv c         `bind` (llp $ λz c →
           wait c         `bind` (llz $
           ret z
  ))))))
```

One concern with embeddings like ours is the legibility of error messages. One of the strengths of Polakow's technique is that it yields relatively readable error messages resulting from misuse of linear assumptions. The situation is even better for violations of session types. For example, the following term fails to provide the multiplier:

```
wrongAnswer =
  defnGV $ fork calculator `bind` (llam $ λc →
           chooseLeft c   `bind` (llam $ λc →
           send (bang (constant 6)) c
                          `bind` (llam $ λc →
           recv c         `bind` (llp $ λz c →
           wait c         `bind` (llz $
           ret z
  )))))
```

The resulting error message correctly identifies that the type of calculator, which requires two arguments, does not align with its use in wrongAnswer, which only supplies one:

```
gvhs.lhs:921:17:
    Couldn't match type ‘a <?> EndIn’
    with ‘Bang (Base Integer)
          <!> (Bang (Base Integer)
          <?> EndIn)’
  ...
```

***Remark*** Prior accounts of GV typically include an additional constant link for linking two dual endpoints together.

$$\mathsf{link} :: (S \otimes \overline{S}) \multimap \mathsf{End}_!$$

Communication is forwarded between the two end points. We chose not to include link here as we rarely need it in practice. It is in fact admissible, and one way to define it is to add a link method to the Session type class.

### 4.1 A CPS semantics for GV

Before giving concrete implementations, we present a formal semantics of GV through a CPS translation, following our previous work [12]. This serves two purposes. First, it captures our intuitive understanding of GV. Second, it motivates our CPS-based implementation of GV, and our introduction of polarized session types.

Our CPS translation $\mathcal{K}[\![-]\!]$ is a call-by-value CPS translation into simply-typed lambda calculus. The translation on functional types and terms is standard. The important part is the translation on function types:

$$\mathcal{K}[\![A \multimap B]\!] = \mathcal{K}[\![A]\!] \to (\mathcal{K}[\![B]\!] \to R) \to R$$

where $R$ is a fixed return type. The translation on all of the other type constructors is homomorphic. The key idea is that rather than returning a value, each function is augmented with a continuation argument which is supplied with the return value of the function.

As observed by Kobayashi et al. [10] and Dardha et al. [4], choice in session types can be encoded in terms of the input and output session types and (linear) sums. We will take advantage of this operation to simplify our CPS translation. We begin by defining a translation $\mathcal{Q}[\![-]\!]$ that implements the choice primitives. On types, it is defined as the homomorphic extension of the equations:

$$\begin{aligned} \mathcal{Q}[\![S_1 \langle +\!\!+ \rangle S_2]\!] &= (\overline{\mathcal{Q}[\![S_1]\!]} \oplus \overline{\mathcal{Q}[\![S_2]\!]})\ \langle ! \rangle\ \mathsf{End}_! \\ \mathcal{Q}[\![S_1 \langle \&\& \rangle S_2]\!] &= (\mathcal{Q}[\![S_1]\!] \oplus \mathcal{Q}[\![S_2]\!])\ \langle ? \rangle\ \mathsf{End}_? \end{aligned}$$

These translations preserve the expected duality requirement: $\overline{\mathcal{Q}[\![S_1 \langle +\!\!+ \rangle S_2]\!]} = \mathcal{Q}[\![S_1 \langle \&\& \rangle S_2]\!]$. The translation of terms is directed by the type translation:

$$\begin{aligned} \mathcal{Q}[\![\mathsf{chooseLeft}\ M]\!] &= \mathsf{fork}\ (\lambda x.\mathsf{send}\ (\mathsf{inl}\ x)\ \mathcal{Q}[\![M]\!]) \\ \mathcal{Q}[\![\mathsf{chooseRight}\ M]\!] &= \mathsf{fork}\ (\lambda x.\mathsf{send}\ (\mathsf{inr}\ x)\ \mathcal{Q}[\![M]\!]) \\ \mathcal{Q}[\![\mathsf{offer}\ M\ N_1\ N_2]\!] &= \mathbf{let}\ (x, c) = \mathsf{recv}\ \mathcal{Q}[\![M]\!]\ \mathbf{in} \\ &\quad\ \mathbf{let}\ () = \mathsf{wait}\ c\ \mathbf{in} \\ &\quad\ \mathbf{case}\ x\ \mathbf{of}\ \mathsf{inl}\ x \mapsto \mathcal{Q}[\![N]\!]_1\ x \\ &\qquad\qquad\quad\ \mid \mathsf{inr}\ x \mapsto \mathcal{Q}[\![N]\!]_2\ x \end{aligned}$$

We now define the CPS translation for session types on the image of $\mathcal{Q}[\![-]\!]$. The intuition for translating session types is as follows: communication between two endpoints of a channel is modelled as function application in which the function represents the input endpoint and the argument represents the output endpoint. The translation on input and output types is as follows.

$$\begin{aligned} \mathcal{K}[\![\mathsf{End}_!]\!] &= R \\ \mathcal{K}[\![\mathsf{End}_?]\!] &= R \to R \\ \mathcal{K}[\![A\ \langle ! \rangle\ S]\!] &= \mathcal{K}[\![A]\!] \to \mathcal{K}[\![\overline{S}]\!] \to R \\ \mathcal{K}[\![A\ \langle ? \rangle\ S]\!] &= (\mathcal{K}[\![A]\!] \to \mathcal{K}[\![S]\!] \to R) \to R \end{aligned}$$

The central property that captures the notion of communication as function application is that if $S$ is an output type then $\mathcal{K}[\![\overline{S}]\!] = \mathcal{K}[\![S]\!] \to R$ (equivalently, if $S$ is an input type then $\mathcal{K}[\![S]\!] = \mathcal{K}[\![\overline{S}]\!] \to R$).

Given the translation on types, there is little choice in the translation on terms. A subtlety is that for send and fork the translation depends on the particular session type at which they are instantiated. We write $\mathsf{send}_!$ for send if the continuation is an output type and $\mathsf{send}_?$ if it is an input type. Similarly, we write $\mathsf{fork}_!$ for fork if the body of its argument takes an output type and $\mathsf{fork}_?$ if it takes an input type.

$$\mathcal{K}[\![\mathsf{send}_!]\!] x\ c\ k = (c\ x)\ k$$
$$\mathcal{K}[\![\mathsf{send}_?]\!] x\ c\ k = k\ (c\ x)$$
$$\mathcal{K}[\![\mathsf{receive}]\!] c\ k = c\ (\lambda x\ d.k\ (x, d))$$
$$\mathcal{K}[\![\mathsf{fork}_!]\!] f\ k = k\ (\lambda x.f\ x\ id)$$
$$\mathcal{K}[\![\mathsf{fork}_?]\!] f\ k = (\lambda x.f\ x\ id)\ k$$
$$\mathcal{K}[\![\mathsf{wait}]\!] c\ k = c\ (k\ ())$$

The reason for the non-uniformity in the translation is that duality is symmetric whereas function application is asymmetric. Notice that despite the non-uniformity the only difference between the two translations of send and fork is the order in which the outer application occurs (we deliberately introduce a $\beta$ expansion in translation of $\mathsf{fork}_?$ in order to emphasize this point). One way of avoiding the non-uniformity is to switch to a polarized variant of GV. We implement polarization in Haskell (§6) and give a translation from GV to polarized GV (§7).

In prior work [11], we give a direct concurrent semantics for GV, and show that it corresponds to cut elimination in Wadler's process calculus CP [23]. The CPS translation agrees with the direct semantics, but in order to simulate all possible reduction paths of GV in the direct semantics, it is necessary to reduce under $\lambda$-abstractions. Interpreting the translation under call-by-name reduction rules, as in Haskell, amounts to choosing a canonical reduction strategy (in which reduction is always driven by the continuation of a $\mathsf{fork}_!$ or the body of a $\mathsf{fork}_?$). Note that GV is confluent, so this restriction does not affect the results of GV programs.

## 5. A Primitive Interpretation

One immediate approach to interpreting GV is to use the concurrency primitives provided in the IO monad, which include primitives for thread creation and synchronization. The obstacle to doing so is the typing of the synchronization primitives. For example, the `synchronous-channels` package [21] provides a type Chan $a$ of synchronous channels between threads; but, all values communicated on the channel must be of type $a$. This is exactly the restriction that session types are designed to lift: a session typed channel may by used to communicate values of arbitrary types safely. For our implementation, we will rely on the boxing of Haskell values giving them a uniform runtime representation, regardless of type.

First we define a dummy channel representation STC $s$ and set its monadic translation to be a synchronous channel.

```
data STC (s :: ⋆)
type instance Mon (STC s) = IOChan s
newtype IOChan s (m :: ⋆ → ⋆) = IOChan (Chan Int)
```

The use of Int in the definition of IOChan is essentially arbitrary: any (boxed) Haskell type would do as well.

The instance of GV for the IO monad is shown in Figure 2. GV's primitives wrap the underlying Haskell primitives; we use unsafeCoerce to make the types appear uniform. The final wait synchronization is accomplished by transmitting a unit value, while choice is implemented by transmitting booleans. Safety of unsafeCoerce is guaranteed by type safety of GV, which we have proved independently [11].

Our implementation of channels is quite similar to that of Pucella and Tov [18]. In particular, they also rely on untyped channels (defined using unsafeCoerce), and prove safety by appeal to the safety of a core session-typed calculus $\lambda^{F\|F}$. Nevertheless, GV is quite different from their embedding. A key difference is the treatment of delegation, or transmitting channels along channels. Here is a (slightly contrived) example of delegation.

```
instance GV STC (RM IO) where
  send (RM mv) (RM mc) = RM $
    do v ← mv
       IOChan c ← mc
       writeChan c (unsafeCoerce v)
       return (IOChan c)
  recv (RM mc) = RM $
    do IOChan c ← mc
       v ← readChan c
       return (MProd (unsafeCoerce v, IOChan c))
  wait (RM mc) = RM $
    do IOChan c ← mc
       v ← readChan c
       case unsafeCoerce v of () → return MOne
  fork (RM mf) = RM $
    do MFun f ← mf
       c ← newChan
       forkIO (do (IOChan c) ← f (IOChan c)
                  writeChan c (unsafeCoerce ()))
       return (IOChan c)
  chooseLeft (RM mc) = RM $
    do IOChan c ← mc
       writeChan c (unsafeCoerce False)
       return (IOChan c)
  chooseRight (RM mc) = RM $
    do IOChan c ← mc
       writeChan c (unsafeCoerce True)
       return (IOChan c)
  offer (RM mc) (RM mleft) (RM mright) = RM $
    do IOChan c ← mc
       MFun left ← mleft
       MFun right ← mright
       v ← readChan c
       if unsafeCoerce v then right (IOChan c)
                         else left (IOChan c)
```

**Figure 2:** IO Implementation of GV

```
sender n =
  defnGV $ llam $
  λc → recv c ‘bind‘ (llp $ λd c →
    send (bang (constant n)) d
              ‘bind‘ (llam $ λd →
    send d c
  ))
answer′ =
  defnGV $ fork (sender 6) ‘bind‘ (llam $ λd →
           fork multiplier ‘bind‘ (llam $ λc →
           send c d        ‘bind‘ (llam $ λd →
           recv d          ‘bind‘ (llp $ λc d →
           send (bang (constant 7)) c
                           ‘bind‘ (llam $ λc →
           recv c          ‘bind‘ (llp $ λx c →
           wait c          ‘bind‘ (llz $
           wait d          ‘bind‘ (llz $
           ret x
  ))))))))
```

Evaluating answer′ yields 42, but relies on a subprocess to provide the multiplicand to the calculator. Note that sending and receiving channels $c$ and $d$ is handled identically to sending and receiving values; in contrast, in Pucella and Tov's system, capabilities to use channels must be sent independently of the channels themselves, and using special primitive operators. We believe that our approach is more compositional; for example, arbitrary values containing multiple channels can be sent without sending the corresponding capabilities separately.

### 5.1 Access Points

GV has a close connection to classical linear logic: in our previous work [11], we showed semantics-preserving translations between GV and Wadler's calculus CP, whose typing and evaluation rules are precisely the proof formation and normalization rules of CLL. This means that GV has strong metatheoretic properties, such as deadlock freedom, but correspondingly limits its expressiveness. Previous work on session-typed functional languages [5, 22] uses a more expressive session initiation mechanism, called access points [20], that avoids these limitations, at the cost of allowing deadlock. We can easily extend our embedding of GV with access points.

```
class GVX (ap :: ⋆ → ⋆) (ch :: ⋆ → ⋆)
          (repr :: Nat → [Maybe Nat] → [Maybe Nat]
              → ⋆ → ⋆)
          | repr → ch ap where
  spawn   :: repr v i o (One ⊸ One)
          → repr v i o One
  close   :: repr v i o (ch End!)
          → repr v i o One
  new     :: DualSession s
          ⇒ repr v i o (ap s ↠ t)
          → repr v i o t
  accept  :: DualSession s
          ⇒ repr v i o (ap s)
          → repr v i o (ch s)
  request :: DualSession s
          ⇒ repr v i o (ap s)
          → repr v i o (ch (Dual s))
```

In addition to the $repr$ and $ch$ types, which serve the same roles they did for the GV class, the GVX class includes a new type constructor for access points, $ap$. Access points are introduced by new; note that in the argument to new, the new access point does not have to be used linearly. Processes initiate communication by calling accept or request on a given access point. Channels are constructed for pairs of accepting and requesting processes, with no guarantee as to which accepters will be paired with which requesters. With this model of communication, we can present a simplified model of process creation, spawn, and allow channels of type $EndOut$ to be closed explicitly with close. It is easy to implement our previous model in terms of this model; fork is defined by

```
fork′ f =
  new (ilam $ λap →
    spawn (llam $ λz → f (accept ap) ‘bind‘ (llam $ λc →
                       close c       ‘bind‘ (llz $
                       ret z)))      ‘bind‘ (llz $
    request ap))
```

We can also see that this model of communication is more expressive than that of pure GV; for example, here is a simple deadlocked term:

```
stuck = new (ilam $ λap → close (accept ap))
```

There can clearly never be a requester for $ap$, so this code must be stuck. Despite the loss of deadlock freedom, and the non-logical character of this extension, we do not lose session fidelity. This illustrates the modularity of GV. It is straightforward to define an instance of GVX in terms of existing Haskell concurrency constructs in a similar manner to the instance of GV in Figure 2. Due to lack of space we omit the code.

## 6. A Polarizing Development

The previous sections develop an implementation of GV based on GHC's concurrency primitives. However, these primitives are more expressive than GV's concurrency. In particular, as we have shown previously [11], GV is terminating and confluent. We now take advantage of that observation to give another, purely functional, implementation of GV.

Our starting point is the CPS interpretation of GV given earlier (§4.1). However, that definition is type directed: negative (or input-like) session types are translated differently from positive (or output-like) session types. To reflect this distinction, we begin by considering a polarized variant of session types, making explicit the distinction between input and output types and requiring coercions (or *shifts*) between them. We give a polarized version of GV and an implementation using continuations (via the Cont monad). In the next section, we show how to interpret our tagless embedding of GV as the tagless embedding of polarized GV in Haskell. Composing the continuation with this interpretation we obtain an implementation of GV in terms of continuations.

We define polarized session types as follows.

$$
\begin{aligned}
S_? &::= \mathsf{Shift}_?\, S_! \mid T\,\langle ?\rangle\, S_? \mid S_?\,\langle \&\& \rangle\, S'_? \mid \mathsf{End}_? \\
S_! &::= \mathsf{Shift}_!\, S_? \mid T\,\langle !\rangle\, S_! \mid S_!\,\langle + \! + \rangle\, S'_! \mid \mathsf{End}_!
\end{aligned}
$$

The existing types for input, output, choice, and closed channels are classified as expected. We add two session types, $\mathsf{Shift}_? S_!$ and $\mathsf{Shift}_!\, S_?$, to explicitly shift output to input session types and vice versa. These constructors have the expected duality relationship:

$$
\overline{\mathsf{Shift}_?\, S_!} = \mathsf{Shift}_!\, \overline{S_!} \qquad \overline{\mathsf{Shift}_!\, S_?} = \mathsf{Shift}_? \overline{S_?}
$$

We can add these type to our embedding following the pattern of the other session type constructors:

```
data Shift? s
data Shift! s
type instance Dual (Shift! s) = Shift? (Dual s)
type instance Dual (Shift? s) = Shift! (Dual s)
```

We must also introduce new constants to our polarized GV language that inhabit the shift types, typed as follows.

$$
\frac{\Gamma \vdash M : \mathsf{Shift}_?\, S_!}{\Gamma \vdash \mathsf{osh}\, M : S_!} \qquad \frac{\Gamma \vdash M : \mathsf{Shift}_!\, S_?}{\Gamma \vdash \mathsf{ish}\, M : S_?}
$$

As with our other communication primitives, these serve as eliminators; fork remains the only term to introduce session types. The naming of these constants follows from their role as eliminators of the corresponding session types; for example, osh eliminates a shift to input, yielding a channel of output type. We now present the embedding of polarized GV.

```
class PGV
  (os :: ⋆ → ⋆) (is :: ⋆ → ⋆)
  (repr :: Nat → [Maybe Nat] → [Maybe Nat] → ⋆ → ⋆)
  | repr → os is where
  sendp  :: repr v i h t
         → repr v h o (os (t ⟨!⟩ s))
         → repr v i o (os s)
  recvp  :: repr v i o (is (t ⟨?⟩ s))
         → repr v i o (t ⊗ is s)
  waitp  :: repr v i o (is End?)
         → repr v i o One
  forkp  :: Dual (Dual s)∼s
         ⇒ repr v i o (os s ⊸ os End!)
         → repr v i o (is (Dual s))
  osh    :: repr v i o (is (Shift? s))
         → repr v i o (os s)
  ish    :: repr v i o (os (Shift! s))
         → repr v i o (is s)
```

```
  chooseLeftp  :: repr v i o (os (s1 ⟨++⟩ s2))
               → repr v i o (os s1)
  chooseRightp :: repr v i o (os (s1 ⟨++⟩ s2))
               → repr v i o (os s2)
  offerp       :: (Dual (Dual s1)∼s1, Dual (Dual s2)∼s2)
               ⇒ repr v i h (is (s1 ⟨&&⟩ s2))
               → repr v h o (is s1 ⊸ t)
               → repr v h o (is s2 ⊸ t)
               → repr v i o t

type DefnPGV os is a = ∀repr i v.
  (LLC repr, PGV os is repr) ⇒ repr v i i a
defnPGV :: DefnPGV os is a → DefnPGV os is a
defnPGV x = x
```

The key difference from GV is that PGV is parameterized by two channel constructors, one ($os$) for channels of output session type and the other ($is$) for channels of input session type. The types of the familiar primitives reflect this distinction: sendp acts on and returns output channels, for instance, while recvp acts on and returns input channels.

Programs in polarized GV closely resemble those in GV, but with the addition of explicit shift operations each time a channel switches from being used for input to being used for output or vice versa. For instance, here is a simplified adaptation of the calculator example (§4) in which only multiplication is supported.

```
multiplierp =
  defnPGV $ llam $
  λc → ish c    ‘bind‘ (llam $ λc →
        recvp c ‘bind‘ (llp $ λx c →
        recvp c ‘bind‘ (llp $ λy c →
        osh c    ‘bind‘ (llam $ λc →
        sendp (times ^ x ^ y) c
  ))))
answerp =
  defnPGV $
    forkp multiplierp              ‘bind‘ (llam $ λc →
    osh c                          ‘bind‘ (llam $ λc →
    sendp (bang (constant 6)) c    ‘bind‘ (llam $ λc →
    sendp (bang (constant 7)) c    ‘bind‘ (llam $ λc →
    ish c                          ‘bind‘ (llam $ λc →
    recvp c                        ‘bind‘ (llp $ λz c →
    waitp c                        ‘bind‘ (llz $
    ret z
  )))))))
```

In this case, the explicit shifts may seem to only add administrative overhead. However, Pfenning and Griffith [16] and Paykin and Zdancewic [15] observe that polarized calculi provide precise control over execution strategy that is left either undetermined, in purely concurrent presentations, or is fixed a priori, as in our CPS translation (§4.1).

We now give a CPS implementation of polarized GV, derived from the CPS semantics of (unpolarized) GV (§4.1). Our implementation relies on two features of the CPS interpretation. First, while the CPS interpretations of output session types vary, the CPS interpretations of input session types are uniform in terms of the interpretation of the output types. Second, because of polarization, we now know whether the continuation of a channel has input or output type statically, even if we do not know its exact session type.

We begin by introducing type families CPSO and CPSI for the CPS translations of input and output session types, respectively. We define types COutput $t\ s\ r$ and CEndOut $r$, the CPS translations of $t\ \langle!\rangle\ s$ and $\mathsf{End}_!$ respectively. Note that those translations refer to the result type $r$ explicitly, and so it appears as a parameter of their translations. We also define a type for the translation of all of the input session types, CInput $s\ r$, defined in terms of the output translation CPSO.

```
type family CPSO (s :: ⋆) :: ⋆ → ⋆
type family CPSI (s :: ⋆) :: ⋆ → ⋆

newtype COutput t s r        =
  COutput {unCOutput :: t (Cont r) → s r → r}
newtype CEndOut r            = CEndOut {unCEndOut :: r}
type instance CPSO (t ⟨!⟩ s) = COutput (Mon t)
                                       (CPSI (Dual s))
type instance CPSO End!      = CEndOut

newtype CInput s r  = CIn {unCIn :: CPSO (Dual s) r → r}
type instance CPSI s = CInput s
```

We define wrapper types for input and output session types, as targets of the Mon type family.

```
type family Ret (m :: ⋆ → ⋆) where
  Ret (Cont r) = r

data OutC (s :: ⋆) (m :: ⋆ → ⋆) where
  OutC :: Dual (Dual s)∼s ⇒ CPSO s (Ret m) → OutC s m
data InC (s :: ⋆) (m :: ⋆ → ⋆) where
  InC :: Dual (Dual s)∼s ⇒ CPSI s (Ret m) → InC s m

data ICH (s :: ⋆)
data OCH (s :: ⋆)

type instance Mon (ICH s) = InC s
type instance Mon (OCH s) = OutC s

type instance CPSO (s1 ⟨++⟩ s2) =
  CPSO ((ICH (Dual s1) ⊕ ICH (Dual s2)) ⟨!⟩ End!)
type instance CPSO (Shift! s) =
  CPSO (OCH (Dual s) ⟨!⟩ End!)
```

The dummy types ICH and OCH represent input and output channels, and are implemented by InC and OutC. We introduce type family Ret to give us access to the result type of the continuation monad. We have not provided implementations of the choice or shift types. To do so, we rely on an extension of the $\mathcal{Q}[\![-]\!]$ translation (§4.1), as follows:

$$\mathcal{Q}[\![\mathsf{Shift}_!\ S_?]\!] = S_?\ \langle!\rangle\ \mathsf{End}_! \qquad \mathcal{Q}[\![\mathsf{Shift}_?\ S_!]\!] = S_!\ \langle?\rangle\ \mathsf{End}_?$$

We give instances of CPSO for $\langle{+\!\!+}\rangle$ and $\mathsf{Shift}_!$ in terms of the interpretation of $\langle!\rangle$ and $\mathsf{End}_!$; the translations of $\langle\&\&\rangle$ and $\mathsf{Shift}_?$ are obtained generically as for the other input session types.

We can now implement the polarized communication primitives. We begin with a helper routine comm that implements communication; that we can do so parametrically in $s$ is the core implementation benefit of the polarized presentation.

```
comm :: (CInput s r → r) → (CPSO (Dual s) r → r) → r
comm c d = c (CIn d)
```

We also define another simple helper routine rid for unwrapping boxed return values.

```
rid :: OutC End! (Cont r) → r
rid (OutC (CEndOut x)) = x
```

The CPS interpretation of polarized GV is given in Figure 3. We can implement sendp, recvp, waitp and forkp following the CPS interpretation of GV (§4.1); our implementation differs from the formal presentation only in the introduction and elimination of wrapper types. The implementations of the shift primitives ish and osh echo the implementations of recvp and sendp. The implementation of choice is somewhat more complicated. Following the $\mathcal{Q}[\![-]\!]$ translation, we expect the implementation of chooseLeftp $m$ to be (the expansion of) the term:

```
osh $ forkp $ llam (λx → sendp (inl (ish x)) m)
```

The shifts are necessary because the result of chooseLeftp should be an output session, but the result of forkp is always an input session. The difficulty we encounter in implementing this is that CPSO is not injective, and thus the type of an application of comm

```
instance PGV OCH ICH (RM (Cont r)) where
  sendp (RM m) (RM n) = RM $ cont $ λk → runCont m $ λx → runCont n $ λ(OutC (COutput f)) → comm (f x) (k ∘ OutC)
  recvp (RM m) = RM $ cont $ λk → runCont m $ λ(InC (CIn f)) → f (COutput (λx y → k (MProd (x, InC y))))
  waitp (RM m) = RM $ cont $ λk → runCont m $ λ(InC (CIn f)) → f (CEndOut (k MOne))
  forkp (RM m) = RM $ cont $ λk → runCont m $ λ(MFun f) → comm (k ∘ InC) (λx → runCont (f (OutC x)) rid)
  osh (RM m) = RM $ cont $ λk → runCont m $ λ(InC (CIn f)) → f (COutput (λx (CIn g) → g (CEndOut (k x))))
  ish  (RM m) = RM $ cont $ λk → runCont m $ λ(OutC (COutput f)) → comm (k ∘ InC) (λz → comm (f (OutC z)) (rid ∘ OutC))

  chooseLeftp (RM m) =
    RM $ cont $ λ(k :: OutC s₁ (Cont r) → r) → runCont m $ λ(OutC (COutput f)) →
      (comm :: (CInput (Shift? s₁) r → r) → (CPSO (Shift! (Dual s₁)) r → r) → r)
        (λ(CIn y) → y (COutput (λx (CIn g) → g (CEndOut (k x)))))
        (λ(COutput g) → comm (λx′ → f (MSum (Left (InC x′))) (CIn (λ(CEndOut x) → x)))
                             (λz → comm (g (OutC z)) (λx → rid (OutC x))))
  chooseRightp (RM m) =
    RM $ cont $ λ(k :: OutC s₂ (Cont r) → r) → runCont m $ λ(OutC (COutput f)) →
      (comm :: (CInput (Shift? s₂) r → r) → (CPSO (Shift! (Dual s₂)) r → r) → r)
        (λ(CIn y) → y (COutput (λx (CIn g) → g (CEndOut (k x)))))
        (λ(COutput g) → comm (λx′ → f (MSum (Right (InC x′))) (CIn (λ(CEndOut x) → x)))
                             (λz → comm (g (OutC z)) (λx → rid (OutC x))))

  offerp (RM m) (RM n₁) (RM n₂) =
    RM $ cont $ λk → runCont m $ λ(InC (CIn f)) →
      f (COutput (λx y → case x of MSum (Left x₁)   → runCont n₁ (λ(MFun f1) → runCont (f1 x₁) k)
                                   MSum (Right x₂) → runCont n₂ (λ(MFun f2) → runCont (f2 x₂) k)))
```

**Figure 3:** CPS Interpretation of Polarized GV

may not be uniquely determined by its arguments. Nevertheless, other than specifying the type of comm, the remainder of the implementation follows the expansion of the term above.

## 7. A Polarizing Interpretation

In this section we define a representation that allows us to interpret (unpolarized) GV as polarized GV in GHC. Doing so gives a concrete implementation of our formal semantics of GV, in a pure setting, and demonstrates that our GV type class admits multiple implementations. We are able to draw more general lessons about the tradeoffs necessary for translating between tagless embeddings.

Our key observation is that for any unpolarized session type, we can compute a minimal set of shifts to produce a corresponding polarized session type, and can introduce corresponding shifts to interpret communication on the unpolarized channel as communication on the polarized channel. We begin by introducing a data type to represent polarity explicitly.

```
data Polarity = O | I
```

We can now define translations from unpolarized session types to input and output polarized session types. We begin with a type family Pol that classifies session types according to their polarity.

```
type family Pol s :: Polarity
type instance Pol (t ⟨!⟩ s)       = O
type instance Pol End!            = O
type instance Pol (t ⟨?⟩ s)       = I
type instance Pol End?            = I
type instance Pol (s₁ ⟨++⟩ s₂)    = O
type instance Pol (s₁ ⟨&&⟩ s₂)    = I
```

We now define the translations, taking advantage of Pol to avoid repetition.

```
type family SToO (s :: ⋆) :: ⋆
type instance SToO s = SToOShift (Pol s) s

type family SToI (s :: ⋆) :: ⋆
type instance SToI s = SToIShift (Pol s) s
```

```
type family SToOShift (p :: Polarity) (s :: ⋆) :: ⋆
type instance SToOShift O s = OSToO s
type instance SToOShift I s  = Shift! (ISToI s)

type family OSToO (s :: ⋆) :: ⋆
type instance OSToO (t ⟨!⟩ s)      = t ⟨!⟩ SToO s
type instance OSToO End!           = End!
type instance OSToO (s₁ ⟨++⟩ s₂)   = SToO s₁ ⟨++⟩ SToO s₂

type family SToIShift (p :: Polarity) (s :: ⋆) :: ⋆
type instance SToIShift I s  = ISToI s
type instance SToIShift O s = Shift? (OSToO s)

type family ISToI (s :: ⋆) :: ⋆
type instance ISToI (t ⟨?⟩ s)      = t ⟨?⟩ SToI s
type instance ISToI End?           = End?
type instance ISToI (s₁ ⟨&&⟩ s₂)   = SToI s₁ ⟨&&⟩ SToI s₂
```

These type families simply insert shifts where appropriate.

Next, we define a similar translation on terms. We do so by introducing a new representation type, RP, which will define a GV representation in terms of the PGV class.

```
newtype RP (os :: ⋆ → ⋆) (is :: ⋆ → ⋆)
  (repr :: Nat → [Maybe Nat] → [Maybe Nat] → ⋆ → ⋆)
  (v :: Nat) (i :: [Maybe Nat]) (o :: [Maybe Nat]) a =
    RP {unRP :: (LLC repr, PGV os is repr, Conv repr) ⇒
            repr v i o a}

evalPolCont :: RP OCH ICH (RM (Cont r)) v '[] '[] a →
               RM (Cont r) v '[] '[] a
evalPolCont = unRP
```

It is parameterized by type constructors for output and input session types, the underlying representation type, and the usual parameters of an LLC representation. The LLC instance for RP is straightforwardly defined in terms of that for RM. We also define a dummy type for representing unpolarized channels in terms of a pair of polarized channel representations:

```
data STP (os :: ⋆ → ⋆) (is :: ⋆ → ⋆) (s :: ⋆)
```

```
class Conv (repr :: Nat → [Maybe Nat] → [Maybe Nat] → ⋆ → ⋆) where
  stoo :: Pol s∼O ⇒ repr v i o (STP os is s)  → repr v i o (os (SToO s))
  stoi  :: Pol s∼I  ⇒ repr v i o (STP os is s)  → repr v i o (is (SToI s))
  otos :: Pol s∼O ⇒ repr v i o (os (SToO s)) → repr v i o (STP os is s)
  itos  :: Pol s∼I  ⇒ repr v i o (is (SToI s))   → repr v i o (STP os is s)
```

**Figure 4:** Interface for Converting between Polarized and Unpolarized Representations

We introduce an monadic interpretation of STP channels, relying on the Pol class to choose the underlying channel representation.

```
type instance Mon (STP os is s) = Mon′ (Pol s) (STP os is s)

type family Mon′ (p :: Polarity) (a :: ⋆) :: (⋆ → ⋆) → ⋆
type instance Mon′ O (STP os is s) = Mon (os (SToO s))
type instance Mon′ I  (STP os is s) = Mon (is (SToI s))
```

The Conv type class Figure 4 is used to mediate between polarized and unpolarized representations of channels, relying on type families SToO and SToI for translating between unpolarized and polarized session types. For the RM type, this translation is straightforward, as the channel representations are all dummy types.

```
instance Conv (RM m) where
  stoo = RM ∘ unRM
  stoi  = RM ∘ unRM
  otos = RM ∘ unRM
  itos  = RM ∘ unRM
```

To obtain the constraints we need for polarized GV we will need to generate equations that state that dualization commutes with the transformations on types. We reify these equations using a GADT:

```
data DualTrans (s :: ⋆) where
  DualTrans :: (Dual (SToI s)∼SToO (Dual s),
                Dual (SToO s)∼SToI (Dual s),
                Dual (SToI (Dual s))∼SToO s,
                Dual (SToO (Dual s))∼SToI s) ⇒ DualTrans s
```

Alas, the proof of these equations in general is by induction over the structure of session types. One way of capturing such an inductive proof is to build the constraints into the Session type class. This has the advantage that session types can remain open, but it has the disadvantage that it requires us to change Session to refer to additional type families that have nothing to do with unpolarized GV. Instead, we will augment the Session class to compute a closed singleton type representation of session types, which we can subsequently use to define proofs by induction. This has the disadvantage of being closed, but the advantage of not needing to hard-wire information which is not relevant to unpolarized GV.

We define a singleton representation ST $s$ for session types $s$.

```
data ST (s :: ⋆) where
  SOutput  :: Session s ⇒ Proxy t → ST s → ST (t ⟨!⟩ s)
  SEndOut :: ST End!
  SInput    :: Session s ⇒ Proxy t → ST s → ST (t ⟨?⟩ s)
  SEndIn    :: ST End?
  SChoose  :: (Session s1, Session s2) ⇒
                ST s1 → ST s2 → ST (s1 ⟨++⟩ s2)
  SOffer    :: (Session s1, Session s2) ⇒
                ST s1 → ST s2 → ST (s1 ⟨&&⟩ s2)
```

We define a singleton type that reifies the polarity of a session type in terms of the Pol type family.

```
data SPolarity s where
  SO :: Pol s∼O ⇒ SPolarity s
  SI  :: Pol s∼I  ⇒ SPolarity s
```

We can now augment the Session class to compute singleton session type and polarity witnesses.

```
class (Dual (Dual s)∼s, Flip (Pol s)∼Pol (Dual s)) ⇒
  Session (s :: ⋆) where
    polarity :: SPolarity s
    sing :: ST s
instance Session s ⇒ Session (t ⟨!⟩ s) where
  polarity = SO
  sing = SOutput Proxy sing
instance Session End! where
  polarity = SO
  sing = SEndOut
instance Session s ⇒ Session (t ⟨?⟩ s) where
  polarity = SI
  sing = SInput Proxy sing
instance Session End? where
  polarity = SI
  sing = SEndIn
instance (Session s1, Session s2) ⇒ Session (s1 ⟨++⟩ s2) where
  polarity = SO
  sing = SChoose sing sing
instance (Session s1, Session s2) ⇒ Session (s1 ⟨&&⟩ s2) where
  polarity = SI
  sing = SOffer sing sing
```

The second class constraint expresses the relationship between polarity and duality, and relies on a type family to flip polarities.

```
type family Flip (p :: Polarity) :: Polarity where
  Flip O = I
  Flip I  = O
```

Now we can build a proof of the commutation equations for any session type. The witnesses are unsurprisingly trivial.

```
dualTrans :: ST s → DualTrans s
dualTrans (SOutput _ s) = case dualTrans s of
                            DualTrans → DualTrans
dualTrans SEndOut = DualTrans
dualTrans (SInput _ s) = case dualTrans s of
                          DualTrans → DualTrans
dualTrans SEndIn = DualTrans
dualTrans (SChoose s1 s2) =
  case (dualTrans s1, dualTrans s2) of
    (DualTrans, DualTrans) → DualTrans
dualTrans (SOffer s1 s2) =
  case (dualTrans s1, dualTrans s2) of
    (DualTrans, DualTrans) → DualTrans
```

As a convenience, we define functions for converting from polarized to unpolarized session types of a specified polarity. This allows us to invert a translation in the other direction which may have flipped the polarity by inserting a shift.

```
otosShift :: (PGV os is repr, Conv repr) ⇒ SPolarity s →
  repr v i o (os (SToO s)) → repr v i o (STP os is s)
otosShift SO = otos
otosShift SI = itos ∘ ish

itosShift :: (PGV os is repr, Conv repr) ⇒ SPolarity s →
  repr v i o (is (SToI s)) → repr v i o (STP os is s)
itosShift SO = otos ∘ osh
itosShift SI = itos
```

```
instance (LLC repr, PGV os is repr, Conv repr) ⇒ GV (STP os is) (RP os is repr) where
  send (RP m) (RP n) = RP (otosShift polarity (sendp m (stoo n)))
  recv (RP m) = RP (letStar (recvp (stoi m)) (λx y → x ⊗ itosShift polarity y))
  wait (RP m) = RP (waitp (stoi m))
  fork (RP (m :: (PGV os is repr, Conv repr) ⇒ repr v i o (STP os is s ⊸ STP os is End!))) =
    case (dualTrans (sing :: ST s), dualTrans (sing :: ST (Dual s))) of
      (DualTrans, DualTrans) → RP (itosShift polarity (forkp m′))
                                where m′ = compose ⌢ llam stoo ⌢ (compose ⌢ m ⌢ (llam (λx → otosShift polarity x)))
  chooseLeft (RP m) = RP (otosShift polarity (chooseLeftp (stoo m)))
  chooseRight (RP m) = RP (otosShift polarity (chooseRightp (stoo m)))
  offer (RP (m :: (PGV os is repr, Conv repr) ⇒ repr v i h (STP os is (s1 ⟨&&⟩ s2)))) (RP n1) (RP n2) =
    case (dualTrans (sing :: ST s1), dualTrans (sing :: ST s2)) of
      (DualTrans, DualTrans) → RP (offerp (stoi m) n1 n2)
                                where n1 = compose ⌢ n1 ⌢ llam (λx1 → itosShift polarity x1)
                                      n2 = compose ⌢ n2 ⌢ llam (λx2 → itosShift polarity x2)
```

**Figure 5:** Unpolarized GV as Polarized GV

We now have all of the ingredients in place to define the full interpretation of GV as polarized GV, which is given in Figure 5. Each case amounts to calling the underlying polarized operator, incorporating shifts as necessary. We make use of a compose operator for linear lambdas in order to perform coercions in the object language.

Perhaps the most important general lesson we have learnt in this section is that if we wish to translate between two typed embedded languages then we are faced with a choice: we can either prime the source language with some knowledge about the type system of the target language (in our case the equations captured by DualTrans), or we can insist that the types of the source language inhabit a closed universe (in our case captured by the singleton type ST). Both choices hurt modularity. An interesting research question is whether it is possible to augment GHC with a richer constraint language in order to support open translations from a source embedded language with an open universe of types into another embedded language.

## 8. Discussion

We have presented a tagless embedding of GV, a session-typed functional calculus, in Haskell. We have presented two interpretations of our embedding, a concurrent one in terms of the primitives of the IO monad and a purely functional one in terms of continuation-passing style. We have also presented extensions to the core calculus: namely access points and polarization.

There have been several recent embedding of session types in mainstream programming languages: including those of Pucella and Tov [18], Imai et al. [8], and Orchard and Yoshida [13] for Haskell; Scalas and Yoshida's `lsessions` library for Scala [19], Jespersen et al.'s library for Rust [9], and Padovani's `FuSe` library for OCaml [14]. We will briefly compare their approaches to ours.

Pucella and Tov [18] target Haskell and use similar mechanisms to ours to account for duality. Their implementation also relies on (potentially unsafe) use of channels in the IO monad. However, where we rely on an embedding of linear $\lambda$-calculus to capture the linearity of channels, they track channel capabilities using a parameterized monad. On the one hand, this means that their approach requires less wrapping when interacting with other Haskell code; for example, they do not require a wrapper like our Base class, or introduction and elimination of the Bang modality. On the other hand, this makes manipulation of channels themselves more complicated in their approach; for example, they cannot simply send or receive channels, but require additional primitives (and some impressive type-level machinery) to transfer channel capabilities independently of the channels themselves. Imai et al [8] describe an alternative approach to representing channel types in a parameterized monads, identifying channels using de Bruijn indexing. Their approach avoids some of the difficulties in that of Pucella and Tov; for example, they are able to send and receive channels directly, rather than separating channels and their capabilities. However, their approach still relies on distinct primitives (and indeed distinct session types) for transmitting channels instead of other forms of data. Finally, Orchard and Yoshida [13] present an embedding of session types in Haskell as an instance of a general approach for encoding effects using parameterized monads. Their approach to channels differs from both those of Pucella and Tov and Imai et al., using names rather than indexing. While more convenient to use, this defeats type inference for many processes.

Scalas and Yoshida [19] provide a library implementing session types in Scala. They rely on a CPS-like interpretation of session-types in terms of one-shot (or linear) channels, which they can implement using Scala's `Future` type. Consequently, their channels do not rely on underlying unsafe operations, but still benefit from using primitive concurrency mechanisms. However, they do not attempt to express linearity in the Scala type system, instead relying on the run-time behavior of the `Promise` and `Future` types to prevent reuse of channels. As a result, erroneous programs may not be detected until run-time, where our approach would reject them statically.

Jespersen et al. [9] give an implementation of session types in Rust making use of Rust's affine types. A value of affine type can be used no more than once, but it may not be used at all. Thus, the Rust encoding guarantees that if a protocol proceeds then it will comply with its session type, but does not prevent a program from simply discarding a channel half way through a protocol.

Padovani [14] implements session types in OCaml. As in Pucella and Tov's implementation, he uses an underlying implementation of simply-typed channels and potentially unsafe conversions; as in Scalas and Yoshida's approach, he defers linearity checking to runtime. This means that his approach is more smoothly integrated with other OCaml code, but that it may not detect until execution errors our approach would have rejected at compilation.


## Acknowledgments

Thanks to Jeff Polakow for providing his Haskell embedding of linear $\lambda$-calculus and for discussions of this work. Thanks for Dominic Orchard for helpful feedback. This work was funded by EPSRC grant number EP/K034413/1.

## References


[1] R. Atkey, S. Lindley, and J. Yallop. Unembedding domain-specific languages. In S. Weirich, editor, *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell, Haskell 2009, Edinburgh, Scotland, UK, 3 September 2009*, pages 37–48. ACM, 2009.

[2] L. Caires and F. Pfenning. Session types as intuitionistic linear propositions. In P. Gastin and F. Laroussinie, editors, *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings*, volume 6269 of *Lecture Notes in Computer Science*, pages 222–236. Springer, 2010.

[3] J. Carette, O. Kiselyov, and C. Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, 19(5):509–543, 2009.

[4] O. Dardha, E. Giachino, and D. Sangiorgi. Session types revisited. In D. D. Schreye, G. Janssens, and A. King, editors, *Principles and Practice of Declarative Programming, PPDP'12, Leuven, Belgium - September 19 - 21, 2012*, pages 139–150. ACM, 2012.

[5] S. J. Gay and V. T. Vasconcelos. Linear type theory for asynchronous session types. *J. Funct. Program.*, 20(1):19–50, 2010.

[6] K. Honda. Types for dyadic interaction. In E. Best, editor, *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993.

[7] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In C. Hankin, editor, *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998.

[8] K. Imai, S. Yuen, and K. Agusa. Session type inference in haskell. In K. Honda and A. Mycroft, editors, *Proceedings Third Workshop on Programming Language Approaches to Concurrency and communication-cEntric Software, PLACES 2010, Paphos, Cyprus, 21st March 2010.*, volume 69 of *EPTCS*, pages 74–91, 2010.

[9] T. B. L. Jespersen, P. Munksgaard, and K. F. Larsen. Session types for rust. In P. Bahr and S. Erdweg, editors, *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming, WGP@ICFP 2015, Vancouver, BC, Canada, August 30, 2015*, pages 13–22. ACM, 2015.

[10] N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the pi-calculus. *ACM Trans. Program. Lang. Syst.*, 21(5):914–947, 1999.

[11] S. Lindley and J. G. Morris. A semantics for propositions as sessions. In J. Vitek, editor, *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 560–584. Springer, 2015.

[12] S. Lindley and J. G. Morris. Talking bananas: Structural recursion for session types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 19-21, 2016*. ACM, 2016.

[13] D. A. Orchard and N. Yoshida. Effects as sessions, sessions as effects. In R. Bodík and R. Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 568–581. ACM, 2016.

[14] L. Padovani. Fuse - a simple library implementation of binary sessions. `http://www.di.unito.it/~padovani/Software/FuSe/FuSe.html`, 2016.

[15] J. Paykin and S. Zdancewic. Linear $\lambda\mu$ is CP (more or less). In S. Lindley, C. McBride, P. W. Trinder, and D. Sannella, editors, *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *Lecture Notes in Computer Science*, pages 273–291. Springer, 2016.

[16] F. Pfenning and D. Griffith. Polarized substructural session types. In A. M. Pitts, editor, *Foundations of Software Science and Computation Structures - 18th International Conference, FoSSaCS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9034 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2015.

[17] J. Polakow. Embedding a full linear lambda calculus in Haskell. In B. Lippmeier, editor, *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, pages 177–188. ACM, 2015.

[18] R. Pucella and J. A. Tov. Haskell session types with (almost) no class. In A. Gill, editor, *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*, pages 25–36. ACM, 2008.

[19] A. Scalas and N. Yoshida. Lightweight session programming in scala. In S. Krishnamurthi and B. S. Lerner, editors, *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, volume 56 of *LIPIcs*, pages 21:1–21:28. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[20] K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In C. Halatsis, D. G. Maritsas, G. Philokyprou, and S. Theodoridis, editors, *PARLE '94: Parallel Architectures and Languages Europe, 6th International PARLE Conference, Athens, Greece, July 4-8, 1994, Proceedings*, volume 817 of *Lecture Notes in Computer Science*, pages 398–413. Springer, 1994.

[21] J. Tov. The `synchronous-channels` package. `https://hackage.haskell.org/package/synchronous-channels`, 2015.

[22] V. T. Vasconcelos, S. J. Gay, and A. Ravara. Type checking a multithreaded functional language with session types. *Theor. Comput. Sci.*, 368(1-2):64–87, 2006.

[23] P. Wadler. Propositions as sessions. *J. Funct. Program.*, 24(2-3):384–418, 2014.