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 λ-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.

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, 17] 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 π-calculus or linear λ-calculus). This gives 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, it makes less clear 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 [9, 10]. 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 λ-calculus, and attempt to minimize the number of concurrent features, preferring to express concurrent features in terms of λ-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 compromise GV’s deadlock freedom, even though the extended calculus no longer enjoys a tight correspondence with classical linear logic.

In this paper, we present a parameterized tagless embedding [1, 3] of GV in Haskell. (We will use the term parameterized tagless or just tagless in preference to finally tagless or tagless finally.) GV’s modularity makes it ideally suited to this purpose. First, we can build on a standard linear calculus; we use Polakow’s [14] embedding of linear λ-calculus in Haskell. To support linear calculi with side effects, such as GV’s concurrency, we build a monadic interpretation of Polakow’s embedding. We characterize the primitives GV adds to the core λ-calculus, and give two interpretations of those primitives, one in terms of the concurrency operations in the IO monad and another that expresses concurrency using continuations (and so uses the continuation monad). The former demonstrates that GV’s primitives can be interpreted directly, without introducing additional encoding overhead, while the latter validates that they also have a purely functional interpretation. Neither Polakow’s embedding of linear λ-calculus nor our embedding of GV require the programmer to supply types explicitly, either directly or with proxy arguments, showing that Haskell’s type system and type inference are expressive enough for session types.

The paper proceeds as follows. We review session types and the role of linearity in session typing (§2), and Polakow’s embedding of linear λ-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 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:

\[ \text{Int} \ (\text{!}) (\text{Int} \ (\text{!}) (\text{Int} \ ?) \text{End}) \]

The type \( T \ (\text{!}) S \) means to send a \( T \) and then continue as \( S \), the type \( T \ ?) S \) means to receive a \( T \) and then continue as \( S \), and the type \( \text{End}_1 \) 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 \( !) \) and \( ? \) 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:

\[ (\text{Int} \ (\text{!}) (\text{Int} \ (\text{!}) (\text{Int} \ ?) \text{End})) \]

The type \( S_1 \ (\#) S_2 \) 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:

\[ (\text{Int} \ ?) (\text{Int} \ (\text{!}) \text{End}) \]

The offer construct \( S_1 \ (\&k) S_2 \) on the server is dual to the selection construct \( S_1 \ (\#) S_2 \) 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 \( \text{End}_1 \) means to close the channel, while \( \text{End}_2 \) 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 \( T \to (T \ (\text{!}) S) \to S \), 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 guarantee that protocols are followed: a process could reuse the original channel (with type \( T \ (\text{!}) S \)) 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, 19] or on some amount of dynamic checking [11, 16].

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 meta-theory of GV, by relying on standard meta-theoretic 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 [14] 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 [14] 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. Parameterising 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.

\[
\text{class} \ \text{Exp} \ \text{repr} \ \text{where}
\]

\[
\lambda :: (\text{repr} \ a \to \text{repr} \ b) \to \text{repr} \ (a \to b)
\]

A term of type \( \text{repr} \ a \) represents the type-correct construction of a \( \lambda \)-term of type \( a \); each type constructor \( \text{repr} \) 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.

\[
\frac{x : A \vdash x : A}{\Delta, x : A \vdash M : B}
\]

\[
\frac{\Delta \vdash \lambda x.M : A \to B}{\Delta \vdash M : A \to B \quad \Delta' \vdash N : A}
\]

\[
\frac{}{\Delta \vdash \Delta' \vdash M \otimes N : B}
\]

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 \( \Delta \otimes \Delta' \) 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 \( \text{repr} \ a \to \text{repr} \ b \)) as a linear \( \lambda \)-calculus term of type \( \text{repr} \ (a \to b) \).

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

\[
\Gamma; \Delta \vdash M : A
\]

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 \( M \). To track the use of specific assumptions, as required by the rule for abstraction, assumptions are replaced by the special assumption \( □ \) instead of being removed from the environment when they are used.

Finally, \( \Gamma \) captures a 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 (\( A \to B \) and \( A \to B \)), linear pairs (\( A \otimes B \), linear sums (\( A \oplus B \)), and the ! modality, which can be used to 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 \( 1 \). The treatment of \( □ \) 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:
Figure 1: Linear λ-calculus typing rules.

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 }
infix 5 ⊔ →

Note that → is the intuitionistic function space: a → b is isomorphic to Bang a → b.

Next, we present the encodings of terms, as the methods of a class LLC of interpretations of linear λ-calculus. The characterization of terms includes not just their types, as in standard tagless embeddings, but also 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 ⊔ 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] → a → b) where
	llam :: (LVar repr v a → repr (S v) (Just v ∷ i) (Nothing : o) a) → repr v ∷ i o (a → b)
	(λ) :: 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 (λ) method takes two terms, one of type a → 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)

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
bang :: repr v ∷ i a → repr v ∷ i (Bang a)
letBang :: repr v ∷ 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]).
Consumes v ∷ i o ⇒ repr v ∷ i o a

The type class Consumes implements the details of the environment transformation; we omit it here, for space reasons, but its definition can be found in Polakow [14]. 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 a

Polakow gives a representation of linear λ-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 λ-calculus terms:

```haskell
newtype RM (m :: * → *)
  (vid :: Nat)
  (ki :: [Maybe Nat])
  (ka :: [Maybe Nat])
  (a :: * )
  = RM { unRM :: m (Mon a m) }
```

```haskell
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 λ-calculus terms of type a. However, unlike Polakow’s representation, we cannot define RM in an entirely uniform way: the representation of an a -> b function cannot simply be m (a -> b), but must instead be m (a -> m b). We account for this by introducing a type family Mon, which maps from the linear type constructors (such as -> and *) to their monadic interpretations (again parameterized by the particular monad m).

```haskell
type family Mon (t :: *) :: (→ *) → *
```

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

```haskell
newtype MFun (a :: (→ *) → *)
  (b :: (→ *) → *)
  (m :: * → *)
  = MFun { mFun :: a m → m (b m) }
```

```haskell
type instance Mon (a -> b) = MFun (Mon a) (Mon b)
```

```haskell
newtype MProd a b (m :: * → *)
  = MProd { mProd :: a m, b m, m } = MProd (Mon a) (Mon b)
```

```haskell
type instance Mon (a ⊗ b) = MProd (Mon a) (Mon b)
data MOne (m :: * → *) = MOne
```

```haskell
type instance Mon One = MOne
```

```haskell
newtype MSum a b (m :: * → *)
  = MSum { mSum :: Either (a m) (b m) }
```

```haskell
type instance Mon (a ⊕ b) = MSum (Mon a) (Mon b)
```

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

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

```haskell
instance Monad m ⇒ LLC (RM m) where
  return f =
    RM $ return $ MFun $ λ x → unRM $ f $ RM $ return x
  f ∘ x = RM $ do f' ← unRM x
                 x' ← unMFun f' x'
                 unMFun f' x' x' ⊗ y = RM $ do x' ← unRM x
                                       y' ← unRM y
                                       return (MProd (x', y'))

  letStar x y f = RM $ unRM x y $ return f
  where f (MProd (x', y)) = f (RM $ return x)
       (RM $ return y)
  one = RM $ return MOne
  letOne x y = RM (unRM x $ return (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 λ-calculus is (unsurprisingly) similar to the construction Carette et al. [3] for a CPS interpretations of their tagless embedding of λ-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.

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

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

```haskell
newtype MBone a m = MBone { unMBase :: a }
type instance Mon (Base a) = MBone a
```

```haskell
instance Monad m ⇒ LLC (RM m) where
...  constant x = RM (return (MBone x))
  RM m v i o a = RM $ do MBone f ← m
                      MBone x ← n
                      return $ MBone (f x)
```

4. The GV Calculus

GV [9, 10] draws on a line of research on session types in functional languages. Vasconcelos et al. [19] and Gay and Vasconcelos [5] initially explored the integration of session types and functional programming. Building on work by Caires and Pfenning [2], Wadler [20] 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 λ-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:

```plaintext
S ::= T (?) S | S1 (&&) S2 | End1 |
     | T (?) S | S1 (+) S2 | End1
```

A session type on a channel captures the expected communication along that channel. Types T (?) S and T (!) S denote receiving and sending values of type T, with the remaining communication captured by S. Types S1 (&&) S2 and S1 (+) S2 reflect offering and making a choice between expectations S and S’. Finally, End1 and End reflect closing a channel (where the End1 endpoint will wait for the End2 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.
We have intentionally chosen not to define session types by datatype 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 [9, 10], 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 \( S \) to denote the dual of session type \( S \), defined as follows:

\[
\begin{align*}
T (\langle ? \rangle S) &= T (\langle ! \rangle S) \\
S \langle k; c \rangle S_2 &= S_1 \langle + \rangle S_2 \\
S \langle + \rangle S_2 &= S_1 \langle k; c \rangle S_2 \\
End &= End \\
End &= End
\end{align*}
\]

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

```haskell
data t (?):s; data s1 (k&c) s2: data End:
data t (!):s; data s1 (+):s2: data End:
```

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

```haskell
type family Dual s
  type instance Dual t (?):s = t (!):s Dual s
  type instance Dual t (!):s = t (?):s Dual s

  type instance Dual (s1 (k&c) s2) = Dual s1 (+):s2 Dual s2
  type instance Dual (s1 (+):s2) = Dual s1 (k&c) s2 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.

```haskell
class (Dual (Dual s):s) => Session s
instance Session s => Session t (?):s => Session t (!):s
instance Session s => Session t (!):s => Session t (?):s
instance Session End => Session End
instance Session End => Session End
instance (Session s1, Session s2) => Session (s1 (k&c) s2)
instance (Session s1, Session s2) => Session (s1 (+):s2)

type DualSession (s :: x) = (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 \( \text{End} \) which imposes no expectations on processes, where we require an explicit channel-closing synchronization. This change stems from Waldner’s identification of session types with the propositions of CLL: linear logic has no self-dual proposition to stand in for \( \text{End} \).

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

```
fork :: (S -> End) -> S
send :: T (\langle ? \rangle S) -> S
recv :: T (\langle ! \rangle S) -> T @ S
wait :: End -> 1
chooseLeft :: (S1 (+):s2) -> S1
chooseRight :: (S1 (+):s2) -> S2
offer :: (S1 (k&c) s2) -> (S1 -> T) -> (S2 -> T) -> T
```

The only introduction form for session types is \( \text{fork} \); the remaining constructs eliminate session types (by performing the expected communication). We define a class of GV representations, extending our class of linear \( \lambda \)-calculus representations.

```
class GV (ch :: x -> x)
  (repr :: Nat -> Maybe Nat) -> [Maybe Nat] -> \text{where}
  send :: DualSession s
  => repr v i o 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 \( \text{repr} \) and a channel type constructor \( \text{ch} \), 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 \( \text{DualSession} \) constraint for the continuation type \( s \) is sufficient to assure that the initial types \( t (?):s \) and \( t (!):s \) 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 \( \text{End} \)-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 (k&c) s2))
chooseRight :: (DualSession s1, DualSession s2)
  => repr v i o (ch (s1 (+):s2))
  => repr v i o (ch (s1 (k&c) s2))
offer :: (DualSession s1, DualSession s2)
  => repr v i h o (ch (s1 (k&c) s2))
  => repr v h o (ch (s1 (+):s2))
```

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 $ \lambda c -> recv c 'bind' (llp $ \lambda x c ->
  recv c 'bind' (llp $ \lambda y 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 i 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 \( * \) 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 (\lambda p -> letStar p f)
llz f = llam (\lambda x -> letOne z f)
```

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

```
type DefnGV ch a = \forall repr i v.
  (LLC repr, GV ch repr) => repr v i o a
```

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 →
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 linear assumptions. The situation is even better for violations of session types. For example, the following term fails to provide the linear assumptions. 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:

gvha.1hs:921:17:
Couldn’t match type ‘a <? EndIn’
with ‘Bang (Base Integer)
<|> (Bang (Base Integer)
<<? EndIn)’


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 [10]. 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 \( K[-] \) is a call-by-value CPS translation into simply-typed lambda calculus. The translation on functional types and terms is standard. For instance

\[
K[\text{T} \to U] = \lambda x. (K[x] \to K[y] \to R) \to R
\]

where \( R \) is a fixed return type.

As observed by Kobayashi et al. [8] 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:

\[
\mathcal{Q} [S_1 \oplus S_2] = \mathcal{Q}[S_1] \oplus \mathcal{Q}[S_2]; \text{End},
\mathcal{Q} [S_1 \otimes S_2] = \mathcal{Q}[S_1] \otimes \mathcal{Q}[S_2]; \text{End}\}
\]

These translations preserve the expected duality requirement:

\[
\mathcal{Q}[S_1 \oplus S_2] = \mathcal{Q}[S_1 \otimes S_2]
\]

The translation of terms is directed by the type translation:

\[
\mathcal{Q}[\text{chooseLeft } M] = \text{fork } (\lambda x. \text{send } (\text{inl } x) \mathcal{Q}[M])
\]
\[
\mathcal{Q}[\text{chooseRight } M] = \text{fork } (\lambda x. \text{send } (\text{inr } x) \mathcal{Q}[M])
\]
\[
\mathcal{Q} [\text{offer } M, N_1, N_2] = \lambda (x, c) \to \text{recv } \mathcal{Q}[M] \text{ in}
\]

\[
\text{let } () = \text{wait } c \text{ in}
\]

\[
\text{case } x \text{ of } \text{inl } x \mapsto \mathcal{Q}[N_1]_1 x
\]

\[
\text{inr } x \mapsto \mathcal{Q}[N_2]_2 x
\]

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.

\[
K[\text{End}] = R
\]
\[
K[\text{End}_?] = R \to R
\]
\[
K[\text{T} (?) S] = (K[T] \to K[S]) \to R
\]
\[
K[\text{T} (!) S] = (K[T] \to K[S]) \to R
\]

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

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 \( \text{send}_d \) for send if the continuation is an output type and \( \text{send}_i \) for it is an input type. Similarly, we write \( \text{fork}_d \) for fork if the body of its argument takes an output type and \( \text{fork}_i \) if it takes an input type.

\[
K[\text{send}_d] x \to c k = (e x) k
\]
\[
K[\text{send}_i] x \to c k = (e x) k
\]
\[
K[\text{receive}_d] c k = c = (\lambda x. d k (x, d))
\]
\[
K[\text{fork}_d] f k = k = (\lambda x. f x id)
\]
\[
K[\text{fork}_i] f k = k = (\lambda x. f x id) k
\]
\[
K[\text{wait}] k c = c = (c ()
\]

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 send terms of \( \text{send}_d \) and \( \text{send}_i \) is that in which the order in which the outer application occurs (we deliberately introduce a \( \beta \) expansion in translation of \( \text{fork}_d \) 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 [9], we give a direct concurrent semantics for GV, and show that it corresponds to cut elimination in Wadler’s process calculus CP [20]. 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 \( \text{fork}_d \) or the body of a \( \text{fork}_d \)). 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 [18] provides a type \( \text{Chan } a \) 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 \( \text{STC} s \) and set its monadic translation to be a synchronous channel.

```haskell
data DualSession = One | ap s

newtype DualSession s = IOChan s

class GV s where
    send (RM nm) (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 MFIO (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

In the definition of \( \text{IOChan} \) is essentially arbitrary: any (boxed) Haskell type would do as well.

The instance of \( \text{GV} \) for the IO monad is shown in Figure 2. GV’s primitives wrap the underlying Haskell primitives; we use \( \text{unsafeCoerce} \) 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 \( \text{unsafeCoerce} \) is guaranteed by type safety of GV, which we have proved independently [9].

Our implementation of channels is quite similar to that of Pucella and Tov [15]. In particular, they also rely on untyped channels (defined using \( \text{unsafeCoerce} \)), and prove safety by appeal to the safety of a core session-typed calculus \( \lambda S \). Nevertheless, GV is quite different from their implementation. A key difference is the treatment of delegation, or transmitting channels along channels. Here is a (slightly contrived) example of delegation.

```haskell
sender n =
    defnGV $ llam $
    (\lambda c \rightarrow recv c \cdot bind \cdot (\llp \rightarrow \lambda d c \rightarrow
         send (bang (constant n)) d
         \cdot bind \cdot (\llam \rightarrow \lambda d \rightarrow
         send d c
        ))
    answer' =
    defnGV $ fork (sender 6) \cdot bind \cdot (\llam \rightarrow \lambda c \rightarrow
        fork multiplier \cdot bind \cdot (\llam \rightarrow \lambda c \rightarrow
        send c d \cdot bind \cdot (\llam \rightarrow \lambda d \rightarrow
        recv d \cdot bind \cdot (\llp \rightarrow \lambda c d \rightarrow
        send (bang (constant 7)) c
        \cdot bind \cdot (\llam \rightarrow \lambda c \rightarrow
        recv c \cdot bind \cdot (\llp \rightarrow \lambda x c \rightarrow
        wait c \cdot bind \cdot (\llz \rightarrow
        wait d \cdot bind \cdot (\llz \rightarrow
        ret x
      )))))))
```

Evaluating \( \text{answer'} \) 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 [9], 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, 19] uses more expressive session initiation mechanism, called access points [17], that avoids these limitations, at the cost of allowing deadlock. We can easily extend our embedding of GV with access points.

```haskell
class GVX (ap :: * → *) (ch :: * → *)
    (repr :: Nat → [Maybe Nat] → [Maybe Nat]
      → * → *)
    (send :: ch ap)
    (receive :: ch ap)
    (put :: ch ap)

instance GVX (ap :: * → *) (ch :: * → *)
    where
        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 \( \text{repr} \) and \( \text{ch} \) types, which serve the same roles they did for the GV class, the GVX class includes a new type constructor for access points, \( \text{ap} \). Access points are introduced by \( \text{new} \); note that in the argument to \( \text{new} \), the new access point
We can add these types to our embedding following the pattern of

definitions by

\text{accept}

by calling

does not have to be used linearly. Processes initiate communication
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, \text{spawn}, and allow channels
of type \text{EndOut} to be closed explicitly with close. It is easy
to implement our previous model in terms of this model; \text{fork}
is defined by

\text{fork}' =

\text{new (\lambda a \rightarrow \text{fork}' (\lambda x \rightarrow f (\text{accept } a \cdot \text{bind} (\lambda z \rightarrow \text{close } c \cdot \text{bind} (\lambda z \rightarrow \text{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:

\text{stuck = new (\lambda a \rightarrow \text{close } c \cdot \text{accept } ap)}

There can clearly never be a requester for \text{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
illuminates 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 [9], 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 ear-
lier (§4.1). However, that definition is type directed: negative (or
input-like) session types get different translations 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 coer-
cions (or \text{shifts}) between them. We give a polarized version of GV
and an implementation using continuations (via the \text{Cont} monad).

In the next section, we show how to interpret our tagless embed-
ing of GV as the tagless embedding of polarized GV in Haskell.

We define polarized session types as follows.

\begin{align*}
S_1 & ::= \text{Shift}_1 \cdot S_1 \mid T (\tau') \cdot S_1 \mid S_1 \cdot (\text{\&} \cdot c) \cdot S_1' \mid \text{End}_1 \\
S_1 & ::= \text{Shift}_1 \cdot S_1 \\ T & ::= \text{End}_1
\end{align*}

The existing types for input, output, choice, and closed channels
are classified as expected. We add two session types, \text{Shift}_1 \cdot S_1
and \text{Shift}_1 \cdot S_1, to explicitly shift output to input session types and vice
versa. These constructors have the expected duality relationship:

\text{Shift}_1 \cdot S_1 \equiv \text{Shift}_2 \cdot S_1 \\
\text{Shift}_2 \cdot S_1 \equiv \text{Shift}_1 \cdot S_2

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

\begin{align*}
\text{data} & \text{Shift}_1 \cdot s \\
\text{data} & \text{Shift}_1 \cdot s \\
\text{type} \text{instance} & \text{Dual} (\text{Shift}_1 \cdot s) \equiv \text{Shift}_1 (\text{Dual } s) \\
\text{type} \text{instance} & \text{Dual} (\text{Shift}_1 \cdot s) \equiv \text{Shift}_1 (\text{Dual } s)
\end{align*}

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

\begin{align*}
\Gamma & \vdash M : \text{Shift}_1 \cdot S_1 \\
\Gamma & \vdash \text{os} : \text{Shift}_1 \cdot S_1 \\
\Gamma & \vdash \text{ish} : M : \text{Shift}_1 \cdot S_1
\end{align*}

As with our other communication primitives, these serve as elimi-
nators; \text{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, \text{os} eliminates a
shift to input, yielding a channel of output type. We now present
the embedding of polarized GV.

\begin{align*}
\text{class} & \text{PGV} \\
(\text{os} :: \ast \rightarrow \ast) & (\text{is} :: \ast \rightarrow \ast) \\
[\text{repr} :: \text{Nat} \rightarrow \text{Maybe} \text{Nat} & \rightarrow \text{Maybe} \text{Nat} \rightarrow \ast \rightarrow \ast) \\
| & \text{repr} \rightarrow \text{os} \text{ is where} \\
\text{sendp} & :: \text{repr} \rightarrow \text{ish} \rightarrow \text{t} \\
& \rightarrow \text{ish} \rightarrow \text{os} \\
\text{recup} & :: \text{repr} \rightarrow \text{os} \rightarrow \text{t} \\
& \rightarrow \text{os} \rightarrow \text{t} \\
\text{waitp} & :: \text{repr} \rightarrow \text{os} \rightarrow \text{t} \\
& \rightarrow \text{os} \rightarrow \text{t} \\
\text{forkp} & :: \text{Dual} \rightarrow \text{os} \rightarrow \text{t} \\
& \rightarrow \text{os} \rightarrow \text{t} \\
\text{osh} & :: \text{repr} \rightarrow \text{os} \rightarrow \text{t} \\
& \rightarrow \text{os} \rightarrow \text{t} \\
\text{ish} & :: \text{repr} \rightarrow \text{os} \rightarrow \text{t} \\
& \rightarrow \text{os} \rightarrow \text{t} \\
\text{chooseLeftp} & :: \text{os} \rightarrow \text{os} \\
& \rightarrow \text{os} \\
\text{chooseRightp} & :: \text{os} \rightarrow \text{os} \\
& \rightarrow \text{os} \\
\text{offerp} & :: \text{Dual} \rightarrow \text{os} \\
& \rightarrow \text{os} \\
\text{type} & \text{DefnPGV} \text{ os is a} \equiv \text{repr} \rightarrow \text{ish} \\
\text{defnPGV} & :: \text{PGV} \text{ os is a} \rightarrow \text{DefnPGV} \text{ os is a} \\
\text{defnPGV} & :: x \equiv x
\end{align*}

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

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

\begin{align*}
\text{multiplier} & = \\
\text{defnPGV} & :: \text{PGV} \rightarrow \text{lam} \rightarrow \\
\lambda c \rightarrow \text{ish} c & \equiv \text{lam} \rightarrow \text{os} c \equiv \text{lam} \rightarrow \\
\text{recv} c & \equiv \text{lam} \rightarrow \text{os} c \equiv \text{lam} \rightarrow \\
\text{sendp} & \text{ times } (x \times y) c
\end{align*}

\begin{align*}
\text{answerp} & = \\
\text{defnPGV} & :: \text{PGV} \rightarrow \text{lam} \rightarrow \\
\lambda c \rightarrow \text{ish} c & \equiv \text{lam} \rightarrow \text{os} c \equiv \text{lam} \rightarrow \\
\text{recv} c & \equiv \text{lam} \rightarrow \text{os} c \equiv \text{lam} \rightarrow \\
\text{sendp} & \text{ bang } (\text{constant } 6) c \equiv \text{lam} \rightarrow
\end{align*}
We capture these by giving instances of CPSO for (++) and Shift: in terms of the interpretation of (!) and End; the translations of (\&\&c) and 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.

\[\text{comm} :: (\text{Clinput}\ s \ r \to r) \to (\text{CPSO\ (Dual\ s)\ r \to r})\]

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

\[\text{rid} :: \text{OutC\ End}\ ; (\text{Cont}\ r) \to r\]

\[\text{rid\ (OutC\ (CEndOut\ x)) = x}\]

The CPS interpretation of polarized GV is given in Figure 3. We can implement sendp, recv\(\_\)p, 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 \(\text{ish}\) and \(\text{osh}\) echo the implementations of recv\(\_\)p and send\(\_\)p. The implementation of choice is somewhat more complicated. Following the \([\_\_]\) translation, we expect the implementation of chooseLeftp m to be (the expansion of) the term:

\[\text{osh}\ \$\ \text{forkp}\ \$\ \text{llam}\ (\lambda x\ .\ \text{sendp}\ (\text{inl}\ (\text{ish}\ x)))\ m}\]

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 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. In doing so we learn more general lessons about what is needed in order to translate one tagless embedding into another.

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.

\[(\text{data\ Pol\ s\ ::\ Polarity\}} = \text{O} \mid \text{I})\]

We can now define two translation from unpolarized session types to input and output polarized session types. To avoid repetition, we begin with a type family \(\text{Pol}\) that classifies session types according to their polarity.

\[(\text{type family\ Pol\ s\ ::\ Polarity\}} = \text{O} \mid \text{I})\]

\[(\text{type instance\ Pol\ (t\ (!)\ s)\ =\ \text{O})\]

\[(\text{type instance\ Pol\ (\text{End})\ =\ \text{O})\]

\[(\text{type instance\ Pol\ (t\ (??)\ s)\ =\ \text{I})\]

\[(\text{type instance\ Pol\ (\text{End})\ =\ \text{I})\]

\[(\text{type instance\ Pol\ (s\ (\&\&c)\ s)\ =\ \text{I})\]

We can now define the translations, relying Pol to avoid duplicating cases.

\[(\text{type family\ StToO\ s\ ::\ Polarity\}} = \text{O} \mid \text{I})\]

\[(\text{type instance\ StToO\ s\ =\ StToO\ Shift\ (Pol\ s)\ s})\]

\[(\text{type family\ StToI\ s\ ::\ Polarity\}} = \text{O} \mid \text{I})\]

\[(\text{type instance\ StToI\ s\ =\ StToI\ Shift\ (Pol\ s)\ s})\]
instance PGV OCH ICH (RM (Cont r)) where
sendp (RM m) = RM $ cont § λx → runCont m $ λx → runCont m $ λ(OutC (COutput f)) → comm (f x) (k o OutC)
recv (RM m) = RM $ cont § λk → runCont m $ λ(InC (CIn f)) → f (COutput (λy → k (MProd x, Inc y)))
wait (RM m) = RM $ cont § λk → runCont m $ λ(InC (CIn f)) → f (CEndOut (k MOne))
forCP (RM m) = RM $ cont § λk → runCont m $ λ(MFun f) → comm (k o 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 MOne))))
ish (RM m) = RM $ cont § λk → runCont m $ λ(OutC (COutput f)) → comm (k o InC) (λx → comm (f (OutC x)) (rid o OutC))
chooseLeftp (RM m) =
RM $ cont § λ (k :: OutC s1 (Cont r) → r) → runCont m $ λ(OutC (COutput f)) →
(com => (CInput (Shifts s1) r → r) → (CPSO (Shifts (Dual s1)) r → r) →
(λ(Cl n y) → y) (COutput (λx (CIn g) → g (CEndOut (k x))))
(λ(COutput g) → comm (λ' (MFun (Left (Inc x')))) (CIn (λ(CEndOut x) → x)))
λ (λx → comm (g (OutC z)) (λx → rid (OutC z)))
chooseRightp (RM m) =
RM $ cont § λ (k :: OutC s2 (Cont r) → r) → runCont m $ λ(OutC (COutput f)) →
(com => (CInput (Shifts s2) r → r) → (CPSO (Shifts (Dual s2)) r → r) →
(λ(Cl n y) → y) (COutput (λx (CIn g) → g (CEndOut (k x))))
(λ(COutput g) → comm (λ' (MFun (Right (Inc x')))) (CIn (λ(CEndOut x) → x)))
λ (λx → comm (g (OutC z)) (λx → rid (OutC z)))
offerp (RM m) (RM n1) (RM n2) =
RM $ cont § λk → runCont m $ λ(InC (CIn f)) →
f (COutput (λx y → case 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

type family SToOShift (p :: Polarity) (s :: *) :: *
type instance SToOShift O s = OSToO s
type instance SToOShift I s = Shifts (ISToO s)
type family OSToO (s :: *) :: *
type instance OSToO (t []) s = t [] SToO s

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] → *)

RP { unRP { (LLC repr, PGV os is repr, Conv repr) => repr v hi a o })

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

It is parameterised 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 :: *)
We define a singleton representation \( ST \) for session types as follows.

```haskell
data ST (s :: *) where
  SOutput :: Session s ⇒ Proxy t → ST s → ST (t (!) s)
  SInput :: ST End₁
  SChoose :: (Session s₁, Session s₂) ⇒ ST s₁ → ST s₂ → ST (s₁ (+) s₂)
  SOffer :: (Session s₁, Session s₂) ⇒ ST s₁ → ST s₂ → ST (s₁ (&κ) s₂)
```

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

```haskell
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.

```haskell
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 End₁ where
  polarity = SI
  sing = SEndOut
```

The second superclass expresses the relationship between polarity and duality, and relies on a type family to negate polarities.

```haskell
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.

```haskell
dualTrans :: ST s → DualTrans s
dualTrans (SOutput _ s) = case dualTrans s of
  DualTrans → DualTrans
```

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 Pucella and Tov’s embedding of session types in Haskell [15], Scala and Yoshida’s 1sessions library for Scala [16], and Padovani’s FüSe library for OCaml [11]. We will briefly compare their approaches to ours.

Pucella and Tov [15] also target Haskell, and use similar mechanisms to ours to introduce linearity and 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.

Scalas and Yoshida [16] 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.

Padovani [11] implements session types in OCaml. As in Puella 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.

References


