Embedding session types in Haskell

Citation for published version:
https://doi.org/10.1145/2976002.2976018

Digital Object Identifier (DOI):
10.1145/2976002.2976018

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

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

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.
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 compromise 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 fu-
be undesirable, such as
for a single binary operation on integers might be as follows:
client-side protocol for a concurrent calculator. The session type
messages along channels. As a simple example, we consider the
types specify both the format (i.e., data type) and ordering of
Session types, originally proposed by Honda [6], are an approach to
server must have dual behavior. The session type of the correspond-
type above represents the client's view of a communication, the
means send two integers, receive an integer in return, and then wait
The type
The offer construct
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:

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

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

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

The type \( S_1 \left( \text{++} \right) S_2 \) 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 correspond-
server is as follows:

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

The offer construct \( S_1 \left( \text{++} \right) S_2 \) on the server is dual to the selection
construct \( S_1 \left( \text{?} \right) S_2 \) 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
behaviors. Unlike many presentations of session types, but inspired
by their logical connections, our session types represent closing
behaviors. Unlike many presentations of session types, but inspired
by their logical connections, our session types represent closing
behaviors. Unlike many presentations of session types, but inspired
by their logical connections, our session types represent closing
behaviors.

Functional session-typed calculi typically present communica-
tion primitives as transforming channels of one session type into
channels of another session type. For example, the sending primi-
tive might have a type like \( T \rightarrow (T \left( \text{!} \right) S) \rightarrow S \), 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 proto-
cols are followed: a process could reuse the original channel (with
\( T \left( \text{!} \right) S \)) to send unexpected \( T \) values, or could discard chan-
nels 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) \( \lambda \)-caluli.
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 \( \lambda \)-caluli.

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 rep-
resentation of an object term, for instance using type classes, al-
ows the same term to be given multiple interpretations. A canoni-
cal example is a parameterized tagless embedding of simply-typed
lambda calculus.

\begin{align*}
\text{class Exp repr where} \\
\text{lam} :: (\text{repr} a \rightarrow \text{repr} b) \rightarrow \text{repr} (a \rightarrow b) \\
\text{app} :: \text{repr} (a \rightarrow b) \rightarrow \text{repr} a \rightarrow \text{repr} b
\end{align*}

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 par-
ticular 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.

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

The variable rule insists that there can be no other variables in the
environment, while the application rule divides its typing environ-
ment among its hypotheses. (We write \( \Delta \circ \Delta' \) 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 \( \text{repr} a \rightarrow \text{repr} b \))
as a linear \( \lambda \)-calculus term of type \( \text{repr} (a \rightarrow b) \).

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

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

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 \( \Box \), rather than being removed from the type environ-
ment; 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 abstrac-
tion and application (\( A \rightarrow B \) and \( A \rightarrow B \)), linear pairs (\( A \times B \)),
linear sums (\( A + B \)), 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 \( \top \). 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:

<code>
newtype a \rightarrow b = Lolli \{ unLolli :: a \rightarrow b \}
data a \otimes b = Tensor a b
data One = One
data a \otimes b = Inl a | Inr b
newtype a \rightarrow b = Arrow \{ unArrow :: a \rightarrow b \}
newtype Bang a = Bang \{ unBang :: a \}
infixr 5 \rightarrow:
</code>

Note that \( \rightarrow \) is the intuitionistic function space: \( a \rightarrow b \) is isomorphic to \( \text{Bang } a \rightarrow b \).

Next, we present the encodings of terms, as the methods of a class \( \text{LLC} \) 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 \( \text{Maybe } \text{Nat} \) where \( \text{Nat} \) is a standard Peano encoding of the natural numbers.

<code>data Nat = Z | S Nat</code>

Each entry in the list represents the presence of a particular variable in the environment, with \( \Box \) denoted by \( \text{Nothing} \). 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.

<code>class LLC (repr :: Nat \rightarrow [Maybe Nat] \rightarrow [Maybe Nat] \rightarrow *) where
  \text{lam} :: (LVar \text{ repr } v a \rightarrow repr (S v) (\text{Just } v : i) (\text{Nothing} : o) b)
  \rightarrow repr v i o (a \rightarrow b)
  \text{UVar} :: repr v i h (a \rightarrow b)
  \rightarrow repr v i o b
\end{code>

Linear application is a simple example of the encoding. The \( (\hat{\cdot}) \) method takes two terms, one of type \( a \rightarrow b \) and one of type \( a \), threading the initial environment through the types of the terms. The result term 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 \( \text{LVar } \text{ repr } v a \)) 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 \( \text{LVar} \) shortly. Other linear term forms are defined similarly.

<code>(\hat{\hat{\cdot}}) :: \text{repr } v i h a \rightarrow \text{repr } v h o b \rightarrow \text{repr } v i o (a \otimes b)
\text{letStar} :: \text{repr } v i h (a \otimes b)
  \rightarrow (\text{LVar } \text{ repr } v a \rightarrow \text{LVar } \text{ repr } (S v) b \rightarrow
  \text{repr } (S v))) (Just v) (Just (S v) : h)
  (\text{Nothing} : \text{Nothing} : o) c)
  \rightarrow \text{repr } v i o c
\text{one} :: \text{repr } v i i One\n\text{letOne} :: \text{repr } v i h One \rightarrow \text{repr } v h o a \rightarrow \text{repr } v i o a
\text{inl} :: \text{repr } v i o a \rightarrow \text{repr } v i o (a \oplus b)
\text{inr} :: \text{repr } v i o b \rightarrow \text{repr } v i o (a \oplus b)
\text{letPlus} :: \text{repr } v i h (a \oplus b)
  \rightarrow (\text{LVar } \text{ repr } v a \rightarrow
  \text{repr } (S v) (\text{Just } v : h) (\text{Nothing} : o) c)
  \rightarrow (\text{LVar } \text{ repr } v b \rightarrow
  \text{repr } (S v) (\text{Just } v : h) (\text{Nothing} : o) c)
  \rightarrow \text{repr } v i o c
\end{code>

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

<code>\text{ilam} :: (UVar \text{ repr } a \rightarrow \text{repr } v i o b)
  \rightarrow \text{repr } v i o (a \rightarrow b)
\text{UVar} :: \text{repr } v i o (a \rightarrow b) \rightarrow \text{repr } v i o a a
  \rightarrow \text{repr } v i o b
\text{bang} :: \text{repr } v i i a \rightarrow \text{repr } v i i (\text{Bang } a)
\text{letBang} :: \text{repr } v i h (\text{Bang } a)
  \rightarrow (\text{UVar } \text{ repr } a \rightarrow \text{repr } v h o b)
  \rightarrow \text{repr } v i o b
\end{code>

We return to the encoding of variables. A linear variable \( \text{LVar } \text{ repr } v a \) for representation \( \text{repr} \) with index \( v \) and type \( a \) is a term of type \( a \) that replaces \( \text{Just } v \) with \( \text{Nothing} \) in its environment:
A representation of an RM type, which maps from the linear type constructors (such as $\otimes$) to their monadic interpretations (again parameterized by the particular monad $m$), is straightforward.

```mراجع
newtype MFun (a :: (λ x -> *)) (b :: (λ x -> *)) = MFun \{ unMFun :: m a \rightarrow m b \}

instance Mon (a -> b) = Mon (a) \rightarrow Mon (b)

instance Mon (a \otimes b) = Mon (a) \otimes Mon (b)

data MOne (m :: * \rightarrow *) = MOne

data MonOne Mon One = MOne

newtype MSum a b (m :: * \rightarrow *) = MSum \{ unMSum :: Either (a m) (b m) \}

instance Mon (a \oplus b) = Mon (a) \oplus Mon (b)

data MBase (m :: * \rightarrow *) = MBase

instance Mon (a \oplus b) = Mon (a \oplus Mon (b))

type instance Mon (a \oplus b) = Mon (a) \oplus Mon (b)

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

```mراجع

```

instance Monad m ⇒ LLC (RM m) where

instance Monad m ⇒ LLC (RM m) where

instance Monad m ⇒ LLC (RM m) where

instance Monad m ⇒ LLC (RM m) where

instance Monad m ⇒ LLC (RM m) where

```mراجع

```

```

```

```

```

```mراجع

```

```

```

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 calculus can model 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 do away with the need for run-time environment recording (necessary) giving up its metatheoretic properties. This section will introduce GV's type system, show how they extend linear λ-calculus, and present our tagless embedding of GV.
A session type on a channel captures the expected communication along that channel. Types \( T(\top) \), \( S \) and \( T(\bot) \) denote receiving and sending values of type \( T \), with the remaining communication captured by \( S \). Types \( S_1 \) (\&\& \( S_2 \)) and \( S_1 \) (\(+\) \( S_2 \)) reflect offering and making a choice between expectations \( S \) and \( S' \). Finally, \( \text{End} \) and \( \text{End}^\top \) reflect closing a channel (where the \( \text{End}^\top \) endpoint will wait for the \( \text{End} \) 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.

\[
\begin{align*}
data
t\cdot(\top)\cdot&\cdot data\ s_1\ (\&\&\ s_2)\cdot data\ \text{End}^\top\cdot 
data
t\cdot(\bot)\cdot&\cdot data\ s_1\ (\+\)\ s_2\cdot data\ \text{End}^\top\cdot
\end{align*}
\]

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 \cite{GV, FT}, 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:

\[
\begin{align*}
T(\top)\cdot S\cdot &\cdot S(\top)\cdot T \cdot 
S_1\ (\&\&\ S_2)\cdot &\cdot S(\top)\cdot S_1\ (\&\&\ S_2) 
\text{End}^\top\cdot &\cdot \text{End}^\top
\end{align*}
\]

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

\[
\begin{align*}
type
dual\ s\ &\cdot\cdot\ (\&\&\ s)\cdot\cdot\ s\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\·
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 liz 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 = llvm (λp → letStar p f)
liz f = llvm (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 = ∀v repr i v.
((LLC repr, GV ch repr) ⇒ repr v i a)
defnGV :: DefnGV ch a → DefnGV ch a
defnGV z = z

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

negater = defnGV $ llvm $ λc →
recv c 'bind' (llp $ λx c →
send (times − (bang (constant (−1)))) ∘ x) c)
calculator = defnGV $ llvm $ λc → offer c multiplier negater

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

answer =
defnGV $ fork calculator 'bind' (llvm $ λc →
chooseLeft c 'bind' (llvm $ λc →
send (bang (constant 6)) c
'bind' (llvm $ λc →
send (bang (constant 7)) c
'bind' (llvm $ λc →
recv c 'bind' (llp $ λx c →
wait c 'bind' (liz $
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' (llvm $ λc →
chooseLeft c 'bind' (llvm $ λc →
send (bang (constant 6)) c
'bind' (llvm $ λc →
recv c 'bind' (llp $ λx c →
wait c 'bind' (liz $
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)’
...
an input type.

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

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 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 in the direct semantics, it is necessary to reduce under \( \lambda \). Nevertheless, 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 \( \text{Chan} a \) 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 be 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 :: s)
  type instance Mon (STC s) = IOChan s
newtype IOChan s = IOChan (Chan Int)
```

The use of \( \text{Int} \) 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 \( \lambda^{\Pi}_{\Pi} F \). 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.

```haskell
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 (MPred 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 $ Illam $ 
    \c \rightarrow \text{recv } c \text{ 'bind' (llp } \lambda \! d \ c \rightarrow\text{ send (bang (constant n)) } d \text{ 'bind' (llam } \lambda \! d \rightarrow\text{ send } d \ c \text{ )})

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

Evaluating answer’ yields 42, but relies on a subprocess to provide the multiplicant 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] → * → *)
    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 fork; fork is defined by

```
fork' f =
  new (lam $ λap →
     spawn (lam $ λz → f (accept ap) ‘bind’ (lam $ λc →
        close c ‘bind’ (lam $ ret z))) ‘bind’ (lam $
          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 (lam $ 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.

```
S₀ :: Shift S₀ | T (?) S₀ | S₀ (κκ) S₀’ | End₀
S₁ :: Shift S₁ | T () S₁ | S₁ (κκ) S₁’ | End₀
```

The existing types for input, output, choice, and closed channels are classified as expected. We add two session types, `Shift S₀` and `Shift S₁`, to explicitly shift output to input session types and vice versa. These constructors have the expected duality relationship:

```
Shift S₀ = Shift S₁
Shift S₁ = Shift S₀
```

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

```
data Shift₀ s
data Shift₁ s

type instance Dual (Shift₀ s) = Shift₁ (Dual s)
type instance Dual (Shift₁ s) = Shift₀ (Dual s)
```

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

```
Γ ⊢ M : Shift₀ S₀
Γ ⊢ M : Shift₁ S₁
```

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] → * → *)
    where
        sendp :: repr v h t
        → repr v i o (os (t (λ! s)) s)
        → repr v i o (os s)

        recup :: repr v i o (is (t (λ? s)) s)
        → repr v i o (t i os)

        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 (os s)

        osh :: repr v i o (os (Shift t))
        → repr v i o (os s)

        ish :: repr v i o (os (Shift s))
        → repr v i o (os s)
```
The key difference from GV versa. For instance, here is a simplified adaptation of the calculator with the addition of explicit shift operations each time a channel returns input channels.

The CPS interpretation of polarized GV is given in Figure 3. In this case, the explicit shifts may seem to only add administrative overhead. However, Pfennig 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 ! s ! r and End r, 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, CIn s r, defined in terms of the output translation CPSO.

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 recv p 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 multiplicative is supported.

multiplierp =
  defnPGV $ llam $ λc → ish c 'bind' (llam $ λc →
  recv p 'bind' (llp $ λx c →
  recv p '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)) 'bind' (llam $ λc →
  sendp (bang (constant 7)) 'bind' (llam $ λc →
  ish c 'bind' (llam $ λc →
  recv p 'bind' (llp $ λy c →
  waitp c 'bind' (llz $
  ret x
  ))))))

In this case, the explicit shifts may seem to only add administrative overhead. However, Pfennig 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).

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 Q[−] translation (§4.1), as follows:

Q[Shift; S] = S (!) End

We give instances of CPSO for (+) and Shift; in terms of the interpretation of (!) and End; the translations of (λ&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.

comm :: (CIn 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, 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 ish and osh echo the implementations of recv p and sendp. The implementation of choice is somewhat more complicated. Following the 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
We now define translations from unpolarized session types to
polarized types. We do so by introducing a new representation type
family \textit{SToShift} parameterized by types \textit{Pol} of
session types and \textit{Pol} of their polarity.

\textbf{Instance Polarity Polarity (Pol \ (Cont \ r)) where}

<table>
<thead>
<tr>
<th>Type Family</th>
<th>Type</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{Pol} \ (Cont \ r)</td>
<td>\textit{Pol}</td>
<td>\textit{Pol}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type Family</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{Pol} \ (Cont \ r)</td>
<td>\textit{Pol}</td>
</tr>
</tbody>
</table>

\textbf{Instance SToShift (p :: Polarity) (s :: *) :: *}

<table>
<thead>
<tr>
<th>Type Family</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{SToShift} (Pol) s = \textit{SToShift} (Pol) s</td>
<td></td>
</tr>
</tbody>
</table>

We now define translations that take advantage of \textit{Pol} to avoid
duplication.

<table>
<thead>
<tr>
<th>Type Family</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{SToO} (s :: *) :: *</td>
<td>\textit{SToO}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type Family</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{SToI} (s :: *) :: *</td>
<td>\textit{SToI}</td>
</tr>
</tbody>
</table>

We now define translations that take advantage of \textit{Pol} to avoid
duplication.
To obtain the constraints we need for polarized GV we will need to introduce a monadic interpretation of the STP channels, relying on the Pol class to choose the underlying channel representation.

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

As a convenience, we define functions for converting from polarized GV to unpolarized GV.

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

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.
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 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 IQ 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 FuSe 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 IQ monad. However, while we refer on an embedding of linear λ-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 λ-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.

---

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

---

```haskell
instance (LLC repr, PGV as is repr, Conv repr) ⇒ GV (STP as is) (RP as is repr) where
  send (RP m) (RP n) = RP (otosShift polarity (sendp m (stoo n)))
  recv (RP m) = RP (letStar (recv (stoo m)) (λx y → x ⊗ itosShift polarity y))
  wait (RP m) = RP (waitp (stoo m))
  fork (RP (m :: (PGV as is repr, Conv repr)) ⇒ repr v i o (STP as is s → STP as is End))) =
    case (DualTrans (sing :: ST s1), DualTrans (sing :: ST (Dual s2))) of
      (DualTrans, DualTrans) ⇒ RP (itosShift polarity (forkp m'))
        where m' = compose ~ llam stool ~ (compose ~ m ~ (llam (λx → itosShift polarity x)))
  chooseLeft (RP m) = RP (otosShift polarity (chooseLeftp (stoo m)))
  chooseRight (RP m) = RP (otosShift polarity (chooseRightp (stoo m)))
  offer (RP (m :: (PGV as is repr, Conv repr)) ⇒ repr v i h (STP as is s3 (κκκ κκκ κκκ κκκ κκκ)) (RP n1) (RP n2)) =
    case (DualTrans (sing :: ST s1), DualTrans (sing :: ST s2)) of
      (DualTrans, DualTrans) ⇒ RP (offerp (stoi m1 n1 n2))
        where n1 = compose ~ n1 ~ llam (λx1 → itosShift polarity x1)
          n2 = compose ~ n2 ~ llam (λx2 → itosShift polarity x2)
```

---

We refer on an embedding of linear λ-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 λ-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


