Haskell Session Types with (Almost) No Class

Riccardo Pucella Jesse A. Tov
Northeastern University
{riccardo,tov}@ccs.neu.edu

Abstract

We describe an implementation of session types in Haskell. Session types statically enforce that client-server communication proceeds according to protocols. They have been added to several concurrent calculi, but few implementations of session types are available.

Our embedding takes advantage of Haskell where appropriate, but we rely on no exotic features. Thus our approach translates with minimal modification to other polymorphic, typed languages such as ML and Java. Our implementation works with existing Haskell concurrency mechanisms, handles multiple communication channels and recursive session types, and infers protocols automatically.

While our implementation uses unsafe operations in Haskell, it does not violate Haskell's safety guarantees. We formalize this claim in a concurrent calculus with *unsafe* communication primitives over which we layer our implementation of session types, and we prove that the session types layer is safe. In particular, it enforces that channel-based communication follows consistent protocols.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming—Haskell; D.3.3

[*Programming Languages*]: Language Constructs and Features—Concurrent programming structures

General Terms Languages

Keywords Session types, concurrency, Haskell, type classes, phantom types, functional programming, embedded type systems

1. Introduction

In typed languages with channel-based communication, such as CML (Reppy 1991) and Concurrent Haskell (Peyton Jones et al. 1996), channels are often homogeneous—parameterized by a single type—and provided with operations to send and receive values of that type over such a channel:

```
writeChan :: Chan a -> a -> IO ()
readChan  :: Chan a -> IO a
```

A natural extension is to parameterize a channel by a protocol regulating the sequence of values that can be sent or received over the channel. For example, a protocol `Int` ! `Int` ? `Bool` ! ε indicates that the associated communication channel can be used to send an integer, receive an integer, then send a Boolean before

Haskell'08, September 25, 2008, Victoria, BC, Canada.

finishing. One can use this channel to communicate with another thread whose corresponding channel has the *dual* protocol `Int` ? `Int` ! `Bool` ? ε that can receive an integer, send an integer, and then receive a Boolean before finishing.

Type systems to enforce that communication conforms to a particular protocol are known as *session-type systems.* A session type is the representation within a type system of the protocol associated with a channel. Session types were first introduced for the π calculus by Honda, Vasconcelos, Kubo, and others (Honda et al. 1998; Gay and Hole 1999, 2005). Recent work has focused on adapting session types to more conventional concurrent languages.

A major barrier to implementing session types in existing languages is aliasing. This is because session types are intrinsically stateful. Suppose a function f expects two arguments, each a channel with session type `Int` ! ε—that is, ready to send an integer, then finish—and suppose f sends an integer on each channel argument that it is given. We might assume a type system such that a channel c may be passed to f only when it can be used in exactly that way: to send an integer and then finish. To ensure that channels are used correctly according to their protocol, we must check wherever we apply f that the two channel arguments are distinct. Otherwise, calling $f\,c\,c$ will perform two sends on channel c, violating its protocol. A common way to deal with this aliasing problem is to use a substructural type system.

Implementations of session types. Several calculi for modeling session types in more conventional concurrent languages have been proposed. Vasconcelos, Gay, and Ravara (2006; 2007), for instance,

have developed calculi for direct-style functional concurrent languages with built-in session types. Others have developed session-type calculi to regulate method invocation in an object-oriented setting (Vallecillo et al. 2003; Dezani-Ciancaglini et al. 2005, 2006).

Armstrong (2002) describes UBF, a framework for manipulating XML data in Erlang. UBF controls the exchange of XML data over Erlang channels through dynamic checking of protocols rather than types.

DeLine and Fähndrich's (2001) Vault adds a form of session type to C, though their focus is more on resources than message-passing concurrency. Sing# (Fähndrich et al. 2006), the implementation language for Microsoft's experimental Singularity OS, adds full-featured session types to C#. Neither Vault nor Sing# is implemented as a library, however: each extends the type system of the base language.

Neubauer and Thiemann (2004) present a library implementation of session types in Haskell that provides session-type checking for client code that communicates via a single implicit channel with some server. Their main example is an SMTP client, where the session type enforces that the protocol be respected. They avoid aliasing by automatically threading the implicit channel through the computation. Neubauer and Thiemann model their implementation with a simple, typed calculus for session-type–regulated communication, and give a type-preserving embedding into Haskell. They use type classes with functional dependencies to model the progress of the state of the channel, and observe that this feature appears necessary.

Our contributions. The main message of this paper is that it is feasible to develop a usable and complete session-type system

on top of a real functional programming language. We argue, in particular, that Haskell's type system is sufficiently powerful to enable a reasonable encoding.

We describe a library, implemented in Haskell, that:

- works with existing Haskell concurrency mechanisms;
- handles multiple communication channels typed independently;
- handles a form of recursive session types; and
- infers session types automatically.

We argue correctness of our implementation in the single-channel case using a core calculus. Aliasing is avoided by threading session type information linearly through the system, by use of an indexed monad. We do not thread the channel itself, but rather a *capability* to use the channel. This permits the channel to be manipulated like any other value, thereby rendering channel aliasing harmless. Capabilities, unlike channels, are an artifact of the type system, and have no run-time existence. The implementation strategy is based on recent work on capability calculi for session types (Pucella and Heller 2008).

Our implementation highlights what seem to be basic prerequisites for a reasonable implementation of session types:

- a means to express the duality of session types (we use type classes); and
- a means to express linear threading of values with changing types (we use an indexed monad).

We show that duality is expressible in many languages, such as Standard ML or Java 1.5. Similarly, indexed monads can be im-

plemented in any higher-order language having some notion of parameterized type, again such as SML or Java. Thus, type classes and functional dependencies are convenient but not necessary for an implementation of session types.

Road map. In §2 we introduce an encoding of session types as Haskell types. In §3, we show how to enforce these session types in Haskell, initially limited to pairs of processes communicating over a single channel, and in §4, we expand this treatment to handle multiple channels at once. In §5, we discuss usability and how our implementation techniques may be applicable in other languages. We model our implementation with a core calculus and present several theorems in §6. Finally, we discuss future work and conclude in §7.

2. Session Types in Haskell

The central idea of session types (Gay and Hole 1999) is to parameterize a channel with some type that represents a protocol, which the type system then enforces. In Haskell, we may encode a protocol using ordinary datatypes:

```
data (:!:) a r
data (:?:) a r
data Eps
```

These datatypes require no constructors because they will have no run-time representation.

If `a` is any type, and `r` is a protocol, then we interpret `a :!: r` as the protocol, "first send an `a`, and then continue with `r`." Similarly, we interpret `a :?: r` as the protocol, "receive an `a`, and then

continue with `r`." The type `Eps` represents the empty protocol of a depleted channel that is not yet closed.

For example, the type `Int :!: Bool :?: Eps` represents the protocol, "send an `Int`, receive a `Bool`, and close the channel." [1]

If the process on one end of a channel speaks a particular protocol, its correspondant at the other end of the channel must be prepared to understand it. For example, if one process speaks `Int :!: Bool :?: Eps`, the other process must implement the dual protocol `Int :?: Bool :!: Eps`. We encode the duality relation using a type class with multiple parameters and functional dependencies (Peyton Jones et al. 1997; Jones 2000).

```
class Dual r s | r -> s, s -> r
```

The functional dependencies indicate that duality is bijective, which helps Haskell to infer protocols and enables a form of subtyping. Sending and receiving are dual: if `r` is dual to `s`, then `a :!: r` is dual to `a :?: s`. The empty session is dual to itself.

```
instance Dual r s => Dual (a :!: r) (a :?: s)
instance Dual r s => Dual (a :?: r) (a :!: s)
instance Dual Eps Eps
```

Our session types also represent alternation and recursion. If `r` and `s` are protocols, then `r :+: s` represents an active choice between following `r` or `s`. The type `r :&: s` represents an offer to follow either `r` or `s`, as chosen by the other process.

```
data (:+:) r s
data (:&:) r s
```

The two alternation operators are dual:

```
instance (Dual r1 s1, Dual r2 s2) =>
                  Dual (r1 :+: r2) (s1 :&: s2)
instance (Dual r1 s1, Dual r2 s2) =>
                  Dual (r1 :&: r2) (s1 :+: s2)
```

Recursion turns out to be slightly more difficult. It is tempting to use a fixed-point combinator, but this would require constructing a type of kind $\star \to \star$ for any desired loop body, which is not generally possible. We need some other way for a recursive type to refer to itself, so we represent this binding using de Bruijn indices.

```
data Rec r
data Var v

instance Dual r s => Dual (Rec r) (Rec s)
instance Dual (Var v) (Var v)
```

The type `Rec r` adds a binding for `r` inside `r`; that is, it implicitly defines a variable bound to the whole of `r` that can be used *within* `r`. We use `Var v` to refer to the variable bound by the `v`th `Rec`, counting outward, where `v` is a Peano numeral written with type constructors `Z` and `S` (*e.g.,* `Z` or `S (S Z)`). For example, the protocol

```
Request :!: Rec (Response :?: (Var Z :&: Eps))
```

says to send a request and then be prepared to receive one or more responses. By contrast, a process implementing the protocol

```
Request :!: Rec ((Response :?: Var Z) :&: Eps)
```

must send a request and be prepared to accept any number of responses.

3. Take 1: One Implicit Channel

Encoding protocols in Haskell is not enough. We cannot merely provide channels parameterized by session types and call it a day. For example, consider a hypothetical *send* operation:

[1] The type constructors `(:!:)` and `(:?:)` are declared right associative and with higher precedence than `(:+:)` and `(:&:)`.

```
send :: Channel (a :!: r) -> a -> IO (Channel r)
```

While this *send* returns the correct channel for the rest of the session, it fails to prevent reuse of the `a :!: r` channel, which would violate the protocol. One way to avoid this problem is to require that channels (or at least their sessions) be treated linearly. In this section, we show how this is done for processes having access to only one channel, which is left implicit in the environment; in the next section, we implement multiple concurrent channels.

We assume a substrate of synchronous channels in both typed and untyped varieties:

```
writeTChan       :: TChan a -> a -> IO ()
readTChan        :: TChan a -> IO a

unsafeWriteUChan :: UChan -> a -> IO ()
unsafeReadUChan  :: UChan -> IO a
```

These channels have dynamic semantics similar to Concurrent ML's (Reppy 1991) synchronous channels. While `TChan`s transmit only a single type, `UChan`s are indiscriminating about what they send and receive. In our implementation, they use *unsafeCoerce#*,

which can lead to undefined behavior if sent and received types differ. We must somehow impose our own type discipline.

We define an abstract type `Session s s' a`, which represents a computation that evolves a session from state `s` to state `s'` while producing a value of type `a`. `Session`'s constructor is not exported to client code, so that clients of the library cannot arbitrarily modify the session state. `Session` is implemented as the composition of the IO monad with a reader monad carrying a untyped channel.

```
newtype Session s s' a =
  Session { unSession :: UChan -> IO a }
```

The phantom parameters `s` and `s'` must track more information than just the current session. We define a type constructor `Cap` to hold not only the current session `r`, but another type `e`, which represents a session type environment:

```
data Cap e r
```

The type `Cap e r` represents the capability to run the protocol `r`. The session type environment `e` provides context for any free variables `Var v` in `r`; that is, `r` must be closed in `e`. We discuss `e` in more detail when we explain recursion, and the other operations merely thread it through.

We can now give *send* a type and definition that will work:

```
send  :: a -> Session (Cap e (a :!: r)) (Cap e r) ()
send x = Session (λc -> unsafeWriteUChan c x)
```

Given an `a`, *send* evolves the session from `a :!: r` to `r`. In its implementation, *unsafeWriteUChan* indiscriminately transmits val-

ues of any type over an untyped channel. Thus, if we fail to ensure that the receiving process expects a value of type `a`, things can go very wrong. In §6, we argue that this cannot happen.

Predictably, `recv` requires the capability to receive an `a`, which it then produces:

```
recv  :: Session (Cap e (a :?: r)) (Cap e r) a
recv   = Session unsafeReadUChan
```

We use `close` to discard an exhausted capability, replacing it with `()`. In this implementation, `close` is a run-time no-op.

```
close :: Session (Cap e Eps) () ()
close  = Session (λ_ -> return ())
```

Composing computations. We also need a way to compose `Session` computations. Composing a session from state s_1 to s_2 with a session from state t_1 to t_2 should be permitted only if $s_2 = t_1$. This is precisely the situation that *indexed monads* capture.

Indexed monads (Atkey 2006; Kiselyov 2006), also known as *parameterized monads*, generalize monads to restrict composition of computations. An indexed monad `m i j a` is parameterized by a precondition `i` and postcondition `j`, as well as a result type `a`. Two indexed-monad computations compose only if the postcondition of the first matches the precondition of the second.

```
class IxMonad m where
  (>>>=) :: m i j a -> (a -> m j k b) -> m i k b
  (>>>)  :: m i j a -> m j k b -> m i k b
  m >>> k = m >>>= λ_ -> k
```

An indexed monad's *unit* does not affect the condition:

```
ret    :: a -> m i i a
```

The `IxMonad` instance for `Session` is then straightforward. It threads the implicit channel through and runs the underlying computations in the `IO` monad.

```
instance IxMonad Session where
  ret a    = Session (λ_ -> return a)
  m >>>= k = Session (λc -> do a <- unSession m c
                               unSession (k a) c)
```

We use *io* to lift an arbitrary `IO` computation into `Session`:

```
io    :: IO a -> Session s s a
io m   = Session (λ_ -> m)
```

Because of *io*, this implementation is actually not linear but affine: an `IO` action may raise an exception and terminate the `Session` computation. Provided that exceptions cannot be caught within a `Session`, this does not jeopardize safety in the sense that any messages received will still have the expected representation. Some formulations of session types guarantee that a session, once initiated, will run to completion, but this seems unrealistic for real-world programs. Handling exceptions from within a session remains an open problem.

Alternation. The session actions *sel1*, *sel2*, and *offer* implement alternation. Action *sel1* selects the left side of an "internal choice", thereby replacing a session `r :+: s` with the session `r`; *sel2* selects the right side. On the other side of the channel, *offer* combines a `Session` computation for `r` with a computation

for *s* into a computation that can handle *r* `:&:` *s*. Dynamically, *sel1* sends `True` over the channel, whereas *sel2* sends `False`, and *offer* dispatches on the boolean value received.

```
sel1  :: Session (Cap e (r :+: s)) (Cap e r) ()
sel1   = Session (λc -> unsafeWriteUChan c True)

sel2  :: Session (Cap e (r :+: s)) (Cap e s) ()
sel2   = Session (λc -> unsafeWriteUChan c False)

offer :: Session (Cap e r) u a ->
         Session (Cap e s) u a ->
         Session (Cap e (r :&: s)) u a
offer (Session m1) (Session m2)
       = Session (λc -> do b <- unsafeReadUChan c
                           if b then m1 c else m2 c)
```

Recursion. Session actions *enter*, *zero*, and *suc* implement recursion. Consider the recursive session type

```
Request :!: Rec ((Response :?: Var Z) :&: Eps)
```

from above. After sending a `Request`, we need some way to enter the body of the `Rec`, and upon reaching `Var Z`, we need some way to repeat the body of the `Rec`. We accomplish the former with *enter*, which strips the `Rec` constructor from *r* and pushes *r* onto the stack *e*:

```
enter :: Session (Cap e (Rec r)) (Cap (r, e) r) ()
enter  = Session (λ_ -> return ())
```

In *e*, we maintain a stack of session types for the body of each enclosing `Rec`, representing an environment that closes over *r*.

Upon encountering a variable occurence `Var` *n*, where *n* is a Peano numeral, we restore the *n*th session type from the stack and return the stack to its former state, using *n* expressed with *zero* and *suc*:

```
zero  :: Session (Cap (r, e) (Var Z))
                 (Cap (r, e) r) ()
zero   = Session (λ_ -> return ())

suc   :: Session (Cap (r, e) (Var (S v)))
                 (Cap e (Var v)) ()
suc    = Session (λ_ -> return ())
```

For example, if the current session is `Var (S (S Z))`, then the operation

```
suc >>> suc >>> zero
```

pops two elements from the stack and replaces the current session with the body of the third enclosing `Rec`.

It is worth remarking that this duplication of type and code to pop the stack is not strictly necessary. If we explicitly write *suc* `>>>` *suc* `>>>` *zero*, Haskell's type checker can infer `S (S Z)`. If, on the other hand, the type is already known, then a type class can do the work:[2]

```
class Pop s s' | s -> s' where pop :: Session s s' ()

instance Pop (Cap (r, e) (Var Z)) (Cap (r, e) r)
  where pop = Session (λ_ -> return ())

instance Pop (Cap e (Var v)) (Cap e' r') =>
         Pop (Cap (r, e) (Var (S v))) (Cap e' r')
  where pop = Session (λ_ -> return ())
```

Putting it all together. Finally, we need a way to connect and run sessions.

A `Rendezvous` is a synchronization object that connects the types of two processes at compile time, and then enables their connection by a channel at run time. The `Rendezvous` carries a phantom parameter indicating the protocol to be spoken on the shared implicit channel, and is represented by a homogeneous, typed channel on which the untyped channel for a particular session will later be exchanged. Creating a `Rendezvous` is as simple as creating a new typed channel and wrapping it.

```
newtype Rendezvous r = Rendezvous (TChan UChan)

newRendezvous :: IO (Rendezvous r)
newRendezvous  = newTChan >>= return . Rendezvous
```

To accept a connection request, we need a `Rendezvous` object, and a `Session` computation whose starting session type matches that of the `Rendezvous`. The computation must deplete and close its channel. At run time, *accept* creates a new untyped channel on which the communication will take place and sends it over the `Rendezvous` channel. It then runs the session computation on the new channel.

[2] Note that the definition of the method *pop* is the same for both instances of `Pop`, which suggests that it could be provided as a default method. This would introduce a subtle bug, however, as it would enable defining new instances of `Pop` with arbitrary effect.

```
accept :: Rendezvous r ->
```

```
          Session (Cap () r) () a -> IO a
accept (Rendezvous c) (Session f) = do
  nc <- newUChan
  writeTChan c nc
  f nc
```

To request a connection, the session type of the `Session` computation must be dual to that of the given `Rendezvous`. At run time, *`request`* receives a new, untyped channel from *`accept`* over the `Rendezvous` channel and then runs the computation using the channel.

```
request :: Dual r r' => Rendezvous r ->
           Session (Cap () r') () a -> IO a
request (Rendezvous c) (Session f)
        = readTChan c >>= f
```

3.1 Implicit Channel Examples

In these examples, we use `ixdo` notation for indexed monads, analogous to `do` notation for monads. This syntax is implemented by a preprocessor.

A print server. As an example, we implement a simple print server. The client side of the print server protocol is:

1. Choose either to finish or to continue.
2. Send a string.
3. Go to step 1.

We first implement the server.

```
server = enter >>> loop where
```

```
  loop = offer close
              (ixdo
                 s <- recv
                 io (putStrLn s)
                 zero
                 loop)
```

GHC's type checker can infer that *server*'s session type is `Rec (Eps :&: (String :?: Var Z))`.

The client reads user input, which it sends to the server for printing. When the user tells the client to quit, it sends one more string to the server, tells the server to quit, and closes the channel.

```
client = enter >>> loop 0 where
  loop count = ixdo
    s <- io getLine
    case s of
      "q" -> ixdo
               sel2
               send (show count ++ " lines sent")
               zero; sel1; close
      _   -> ixdo
               sel2; send s
               zero; loop (count + 1)
```

GHC infers the session type `Rec (Eps :+: (String :!: Var Z))` for *client*, which is clearly dual to the type inferred for *server* above.

We run a session by creating a new `Rendezvous`, having the server accept in a new thread, and having the client request in the main thread.

```
runPrintSession = do
  rv <- newRendezvous
```

```
    forkIO (accept rv server)
    request rv client
```

```
recv  :: Channel t -> Session (Cap t e (a :?: r), x) (Cap t e r, x) a
close :: Channel t -> Session (Cap t e Eps, x) x ()
sel1  :: Channel t -> Session (Cap t e (r :+: s), x) (Cap t e r, x) ()
sel2  :: Channel t -> Session (Cap t e (r :+: s), x) (Cap t e s, x) ()
offer :: Channel t -> Session (Cap t e r, x) u a -> Session (Cap t e s, x) u a -> Session (Cap t e (r:&:s), x) u a
enter :: Channel t -> Session (Cap t e (Rec r), x) (Cap t (r, e) r, x) ()
zero  :: Channel t -> Session (Cap t (r, e) (Var Z), x) (Cap t (r, e) r, x) ()
suc   :: Session (Cap t e (Var v), x) (Cap t f s, x) () -> Session (Cap t (r, e) (Var (S v)), x) (Cap t f s, x) ()
```

Figure 1. Types for multiple channel `Session` operations

An example of subtyping. Our implementation provides a form of protocol subtyping. Consider a reimplementation of Gay and Hole's (1999) arithmetic server, which provides two services, addition and negation:

```
server1 = offer
  (ixdo a <- recv
        b <- recv
        send (a + b)
        close)
  (ixdo a <- recv
        send (-a)
        close)
```

The full protocol for *server1* is inferred:

```
(Integer :?: Integer :?: Integer :!: Eps) :&:
(Integer :?: Integer :!: Eps)
```

A second server implements only the negation service:

```
server2 = offer
  close
```

```
(ixdo a <- recv
      send (-a)
      close)
```

Its protocol is inferred as well:

```
Eps :&: (Integer :?: Integer :!: Eps)
```

A particular client may avail itself of only one of the offered services:

```
client' x = ixdo sel2; send x; y <- recv; close; ret y
```

The client's protocol is inferred as `r :+: (a :!: b :?: Eps)`, which unifies with the duals of both servers' protocols. Without the functional dependencies in `Dual`, however, attempting to connect the client with `server2` leads the type checker to complain that there is no instance of `Dual` for `r` and `Eps`; connecting `client` with `server1` also fails to type check. The functional dependency nudges the type checker towards attempting to unify `r` with the corresponding part of either server's type, which then succeeds. As a result, the `client` may be composed with both servers in the same program and never notices the difference.

4. Take *n*: Multiple Channels

Rather than limit ourselves to one implicit channel at a time, it might be more flexible to work with several channels at once. To extend `Session` to handle multiple channels, our first step is to separate the channel itself from the capability to use it for a particular session:

```
newtype Channel t = Channel UChan
data Cap t e r
```

The parameter `t` is a unique tag that ties a given channel to the capability to use it. A `Channel t` is an actual value at run time, while the corresponding `Cap t e r` is relevant only during type-checking. We allow `Channel t` to be aliased freely because a channel is unusable without its capability, and we treat capabilities linearly. As before, the capability also contains a session type environment `e` and a session type `r` that is closed in `e`.

We now index `Session` by a *stack* of capabilities, while underneath the hood, it is just the `IO` monad. `Session` is no longer responsible for maintaining the run-time representation of channels, but instead it keeps track of the compile-time representation of capabilities.

```
newtype Session s s' a = Session { unSession :: IO a }

instance IxMonad Session where
  ret      = Session . return
  m >>>= k = Session (unSession m >>= unSession . k)

io :: IO a -> Session s s a
io  = Session
```

A `Session` computation now carries a stack of capability types, and communication operations manipulate only the top capability on the stack, leaving the rest of the stack unchanged. The `send` operation takes a channel as an argument rather than obtaining it implicitly, and the tag `t` on the channel must match the tag in the capability.

```
send :: Channel t -> a ->
        Session (Cap t e (a :!: r), x)
                (Cap t e r, x) ()
send (Channel c) a = Session (unsafeWriteUChan c a)
```

In the type above, `Cap t e (a :!: r)` is the capability on the top of the stack before the *send*, and `Cap t e r` is the capability after the *send*. Type variable *x* represents the rest of the capability stack, which is unaffected by this operation.

The implementations of the remaining operations are similarly unsurprising. Each differs from the previous section only in obtaining a channel explicitly from its argument rather than implicitly from the indexed monad. Their types may be found in Figure 1. Note that *close* now has the effect of popping the capability for the closed channel from the top of the stack.

Stack manipulation. Channel operations act on the top of the capability stack. Because the capability for the particular channel we wish to use may not be on the top of the stack, we may need to use other capabilities than the top one. The *dig* combinator suffices to select any capability on the stack. Given a `Session` computation that transforms a stack *x* to a stack *x'*, *dig* lifts it to a computation that transforms `(r, x)` to `(r, x')` for any *r*; thus, *n* applications of *dig* will select the *n*th capability on the stack. Note that *dig* has no run-time effect, but merely unwraps and rewraps a `Session` to change the phantom type parameters.

```
dig  :: Session x x' a -> Session (r, x) (r, x') a
dig   = Session . unSession
```

In combination with *swap*, we may generate any desired stack permutation. Since *swap* exchanges the top two capabilities on

the stack, *dig* and *swap* may be combined to exchange any two adjacent capabilities.

```
swap :: Session (r, (s, x)) (s, (r, x)) ()
swap  = Session (return ())
```

One reason we may want to rearrange the stack is to support *forkSession*, which runs a `Session` computation in a new thread, giving to it the entire *visible* stack. Thus, to partition the stack between the current thread and a new thread, we use *dig* and *swap* until all the capabilities for the new thread are below all the capabilities for the current thread. Then we call *forkSession* under sufficiently many *dig*s so that it takes only the desired capabilities with it.

```
forkSession :: Session x () () -> Session x () ()
forkSession (Session c)
              = Session (forkIO c >> return ())
```

For example, to keep the top two capabilities on the stack for the current thread and assign the rest to a new thread `m`, we would use `dig (dig (forkSession m))`.

Making a connection. In the implicit channel case, each *accept* or *request* starts a single `Session` computation that runs to completion. Because we now have multiple channels, we may need to use *accept* and *request* to start new communication sessions during an ongoing `Session` computation. Given a `Rendezvous` and a continuation of matching session type, *accept* creates a new channel/capability pair. It calls the continuation with the channel, pushing the corresponding capability on the top of its stack. The rank-2 type in *accept* ensures that the new `Channel t` and `Cap t () r`

cannot be used with any other capability or channel. In §5 we discuss an alternate formulation that does not require higher-rank polymorphism, but this version here seems more elegant.

```
accept :: Rendezvous r ->
          (forall t. Channel t ->
            Session (Cap t () r, x) y a) ->
          Session x y a
accept (Rendezvous c) f = Session (do
  nc <- newUChan
  writeTChan c nc
  unSession (f (Channel nc)))
```

The *request* function behaves similarly, but as before, it uses the dual session type.

```
request :: Dual r r' =>
           Rendezvous r ->
           (forall t. Channel t ->
             Session (Cap t () r', x) y a) ->
           Session x y a
request (Rendezvous c) f = Session (do
  nc <- readTChan c
  unSession (f (Channel nc)))
```

We may start a `Session` computation from within the IO monad. The type of *runSession* ensures that the computation both begins and ends with no capabilities in the stack.

```
runSession :: Session () () a -> IO a
runSession  = unSession
```

Sending capabilities. Now that we have multiple channels, we

might wonder whether we can send capabilities themselves over a channel. Certainly, but since we do not allow direct access to capabilities, this requires a specialized pair of functions.

```
send_cap :: Channel t ->
            Session (Cap t e (Cap t' e' r' :!: r),
                     (Cap t' e' r', x))
                    (Cap t e r, x) ()
send_cap (Channel c)
          = Session (unsafeWriteUChan c ())

recv_cap :: Channel t ->
            Session (Cap t e (Cap t' e' r' :?: r), x)
                    (Cap t e r, (Cap t' e' r', x)) ()
recv_cap (Channel c) = Session (unsafeReadUChan c)
```

Observe that because capabilities have no run-time existence, the actual value sent over the channel is `()`. This provides synchronization so that the receiving process does not perform channel operations with the capability before the sending process has finished its part. The phantom type parameters to `Session` change to reflect the transmission of the capability.

4.1 An Example with Multiple Channels

As an example, we give an implementation of the Sutherland-Hodgman (1974) reentrant polygon clipping algorithm, which takes a plane and a series of points representing the vertices of a polygon, and produces vertices for the polygon restricted to one side of the plane. Shivers and Might (2006) present a stream transducer implementation, which we follow. Each transducer takes one plane to clip by, and two `Rendezvous` objects for the same proto-

col. It connects on both, and then receives original points on one channel and sends clipped points on the other.

We assume that we have types `Plane` and `Point`, a predicate *above* that indicates whether a given point is on the visible side of a given plane, and a partial function *intersection* that computes where the line segment between two points intersects a plane.

GHC infers all the types in this example.

```
type SendList a = Rec (Eps :+: (a :!: Var Z))

clipper :: Plane -> Rendezvous (SendList Point)
           -> Rendezvous (SendList Point)
           -> Session x x ()
clipper plane inrv outrv =
  accept outrv $ λoc ->
  request inrv $ λic -> ixdo
  let shutdown = ixdo close ic; sel1 oc; close oc
      put pt   = dig $ ixdo
        sel2 oc; send oc pt; zero oc
      -- Attempt to get a point; pass it to yes, or
      -- call no if there are no more:
      get no yes = offer ic no $ ixdo
        pt <- recv ic; zero ic; yes pt
      -- If the line crosses the plane, send the intersection point:
      putCross line =
        maybe (ret ()) put (line `intersect` plane)
      putIfVisible pt   =
        if pt `above` plane then put pt else ret ()
  dig (enter oc)
  enter ic
  get shutdown $ λpt0 ->
    let loop pt = ixdo
          putIfVisible pt
```

```
        get (putcross (pt, pt0) >>> shutdown)
            (λpt' -> ixdo putcross (pt,pt')
                          loop pt')
    in loop pt0
```

We use *sendlist* to send a list of points to the first transducer in the pipeline, and we use *recvlist* to accumulate points produced by the last transducer.

```
sendlist :: [a] -> Rendezvous (SendList a)
         -> Session x x ()
sendlist xs rv = accept rv start where
  start oc = enter oc >>> loop xs where
    loop []     = ixdo sel1 oc; close oc
    loop (x:xs) = ixdo sel2 oc; send oc x
                       zero oc; loop xs

recvlist :: Rendezvous (SendList a) -> Session x x [a]
recvlist rv = request rv start where
  start ic = enter ic >>> loop [] where
    loop acc = offer ic
      (close ic >>> ret (reverse acc))
      (recv ic >>>= λx -> zero ic >>> loop (x : acc))
```

Given a list of planes and a list of points, *clipMany* starts a *clipper* for each plane in a separate thread. It starts *sendlist* a new thread, giving it the list of points and connecting it to the first *clipper*. It then runs *recvlist* in the main thread to gather up the result.

```
clipMany :: [Plane] -> [Point] -> IO [Point]
clipMany planes points = runSession $ ixdo
    rv <- io newRendezvous
```

```
forkSession (sendlist points rv)
let loop []     rv = recvlist rv
    loop (p:ps) rv = ixdo
      rv' <- io newRendezvous
      forkSession (clipper p rv rv')
      loop ps rv'
loop planes rv
```

4.2 Beyond Stack Access for Capabilities

We are not yet satisfied. We have multiple, independently typed channels, but accessing their capabilities by stack position is a pain. With a considerable amount of type class machinery, we can replace position-based capability access with named-based access. In particular, we equip channels with sufficient information so that the type system can automatically search the capability environment for their capabilities. The details of how to do this are beyond the scope of this paper, and we are still exploring designs with different trade-offs. Two approaches seem promising.

One possibility is to add a key to each capability, changing the capability stack into a heterogeneous association list, *à la* HList (Kiselyov et al. 2004) and parameterizing channels by this key as well. New keys are generated by each `accept` and `request`. Capability environment lookup, update, and delete operations are written using type classes with functional dependencies, and the effect of each operation is encoded in its type class context. For example, `recv` and `send` in such a system may have the types:

```
recv :: (Modify t k s (a :?: r) r s') =>
        Channel t k -> Session s s' a
```

```
send :: (Modify t k s (a :!: r) r s') =>
        Channel t k -> a -> Session s s' ()
```

Unfortunately, this approach destroys session type inference.

A second strategy is to include all session information, in addition to a key, as type parameters to `Channel`. We keep a second copy of the type parameters in the capability environment. Session operations take a channel, check that it matches the information in the capability environment, and then update the environment and return an updated channel:

```
recv :: (Update (Channel t n e (a :?: r)) s
                (Channel t n e r) s') =>
        Channel t n e (a :?: r) ->
        Session s s' (a, Channel t n e r)

send :: (Update (Channel t n e (a :!: r)) s
                (Channel t n e r) s') =>
        Channel t n e (a :!: r) -> a ->
        Session s s' (Channel t n e r)
```

In the above example, the channels maintain the session types, and the capability environment ensures that channels are not used improperly. The type class `Update` checks that the given channel type is current and modifies the capability environment to reflect changes to the channel's session type. Keeping session types directly available as parameters to channels restores type inference, but encoding recursion remains problematic.

5. Discussion

5.1 Usability

Whenever an encoding-based library is proposed, the question of usability arises. While the best way to answer that question is with a well-designed usability test suite and real non-expert users, we can make an initial pass at assessing the strengths and weaknesses of our approach.

The main strength of our approach is that the resulting code is very close to that of standard channel-based concurrent programs, with little additional burden. The example in Appendix 4.1 illustrates this point, as we implement a polygon-clipping algorithm using our library in much the way that one might implement it using message-passing concurrency without session types. The only differences involve the use of *accept* and *request* to acquire the ability to communicate over a channel, and the use of *seli* operations to control choices. The rest of the code is merely channel-based communication. That the computation occurs in a monad would be necessary even without our library. Type inference works nicely in this example and many others. The code requires no type annotations to type check, and we provide type signatures only for clarity.

We do, however, suffer many of the problems inherent in the use of phantom types and similar encoding techniques. In particular, type error messages can be obscure and need to be understood in light of the encoding. Nonetheless, many of our error messages are quite comprehensible. For example, if the example *client* in §3.1 attempts to send an `Int` rather than a `String`, GHC complains that it "`Couldn't match expected type 'Int' against inferred type 'String,'`" rather than some obscure message about missing instances. If the client attempts a *recv* when the server is receiving as well, that error mes-

sage is also informative: "`Couldn't match expected type 'a :?: Var Z' against inferred type 'String :!: r.'`"

Other usability problems arise from a balancing act having to do with naming. First, position-based access to capabilities using *dig* and *swap* is more difficult to program than name-based access. Likewise, de Bruijn indices for recursion are likely more difficult to manage than named variables. Finally, the use of binary sums for alternation may also be inferior to name-based alternation among an arbitrary number of cases, especially when representing protocols with multi-way alternation. We have experimented with encoding, for instance, name-based capability access or named-variable recursion using type classes with either functional dependencies or associated types (Chakravarty et al. 2005). However, this tends to yield worse type inference and more cryptic error messages. There is a trade-off, therefore, between the weight of the encoding and the feasibility of type inference and useful type errors.

5.2 Applicability to Other Languages

Our thesis is that session types may be embedded in a general purpose programming language without resorting to exotic language features and techniques. Yet in the previous section, we took advantage of several extensions to Haskell's type system: multi-parameter type classes, functional dependencies, empty datatype declarations, and rank-2 polymorphism. In this section, we discuss which features of our session types implementation are necessary

```
module type DUAL = sig
  type client type server
  val witness : (client, server) dual
```

```
  end

  module Eps : DUAL
    with type client = eps
     and type server = eps

  module Send (T : sig type t end) (R : DUAL) : DUAL
    with type client = (R.client, T.t) send
     and type server = (R.server, T.t) recv

  module Choose (R : DUAL) (S : DUAL) : DUAL
    with type client = (R.client, S.client) choose
     and type server = (R.server, S.server) offer
```

(a) OCaml modules

```
final class Dual<C, S> {
  private Dual() { }

  public static Dual<Eps, Eps> eps
      = new Dual<Eps, Eps>();

  public <A> Dual<Send<A, C>, Recv<A, S>>
  send() {
      return new Dual<Send<A, C>, Recv<A, S>>();
  }

  public <C2, S2> Dual<Choose<C, C2>, Offer<S, S2>>
  choose(Dual<C2, S2> right) {
      right.nullCheck();
      return new Dual<Choose<C, C2>, Offer<S, S2>>();
  }
```

(b) Java 1.5 generics

Figure 2. Fragments of duality proof systems

and which merely convenient, and we argue that a similar implementation is practical in other languages.

Session type duality. Representing duality is essential. In our implementation, we use a type class to check that communicating processes speak dual protocols. The type class uses functional dependencies to propagate information when a type is incompletely specified; as we saw in §3.1, this helps with polymorphism. We could encode protocol subtyping explicitly—this is rather complicated—but with functional dependencies, it comes for free.

Nonetheless, type classes are not strictly necessary for this to work. The `Dual` class and instances encode a small proof system for duality, and Haskell constructs duality proofs where needed. It is possible, however, to encode proofs explicitly, and constructing proof objects can be as simple as writing one side of the protocol in stylized form. For example, using Java 1.5 generics (Gosling et al. 2005), we write

```
Dual<Send<String, Recv<Character, Eps>>,
     Recv<String, Send<Character, Eps>>> =
  Dual.eps.<Char>recv().<String>send();
```

for the same proof. Fragments of two such proof systems may be found in Figure 2.

We have written similar duality proof systems in Standard ML, C#, and Scala. These formulations rely on two essential features:

- Session types and duality theorems are represented by some notion of parameterized type.
- Either abstract types or private constructors prevent arbitrary construction of proof objects.

Java almost fails on the latter count, since `null` is a proof of every theorem, but we consider dynamic `null` checks to be a fact of life in Java. Each session-type factory method performs a `null` check on its receiver, and some check their argument, to ensure that any non-`null` `Session` object is a valid proof. One additional dynamic check before connecting ensures that the session then proceeds without incident.

Notably, it is also possible to embed an implicit duality proof in the indexed monad. In this formulation, we maintain a pair of the current session and its dual at every step. In our Standard ML proof-of-concept implementation of this scheme, a complete session computation has the type $(\gamma * \delta, unit, \alpha)$ *session*, where γ is the computation's own protocol, δ is the dual protocol, and α is the type of value computed by it. Rendezvous objects are parameterized by the protocol as seen from the client side, so when starting a session, *request* ensures that γ matches the rendezvous object, while *accept* checks δ:

```
val request : γ rendezvous ->
              (γ * δ, unit, α) session -> α
val accept  : δ rendezvous ->
              (γ * δ, unit, α) session -> α
```

Indexed monads. Barring a linear or affine type system in the host language—Clean's uniqueness types (Barendsen and Smetsers

1996) may be sufficient—some other means to prevent aliasing of capabilities is required. The indexed monad `Session` accomplishes this in our Haskell implementation of session types.

Indexed monads offer a principled way to embed a variety of substructural type disciplines, and they are reasonably expressed in the same variety of languages that can express duality proofs. Since the type of the *bind* operation is higher order, however, Java's lack of lambda incurs a heavy syntactic burden. As types get larger, explicit type annotations become increasingly burdensome as well.

Multiple concurrent sessions. The solution employed in §4, a heterogeneous stack of capabilities, should work in any of the languages mentioned in the discussion above. All of these languages' type systems are capable of expressing the requisite polymorphic *push* and *pop* operations.

In Haskell, however, we can go a step further (§4.2), by storing capabilities in a heterogeneous record. Accessing capabilities by name rather than stack position is a convenient improvement, but we do not know how to do this without the compile-time computation that type classes provide.

Separate channels and capabilities. In the Haskell implementation with multiple channels, we separate channel values from the capability types that restrict their use. This has the nice property that the capability stack and attendant stack manipulations need have no runtime reality. In a language that supports neither existential nor rank-2 quantification, nor some other way of generating unique tags, it is possible to combine each channel-capability pair into a single value managed by the indexed monad.

6. Formalization

The implementation of session types in §3 and §4 is in terms of unsafe, untyped channels. Yet, we claim that our use of the type system prevents threads from receiving values of unexpected type. We formalize this intuition first by modeling the unsafe channel operations with a core calculus $\lambda^{\mathrm{F}\|\mathrm{F}}$. We add session types to $\lambda^{\mathrm{F}\|\mathrm{F}}$

$$
\begin{array}{rll}
\textit{type variables} & \alpha, \beta, \gamma & \\
\textit{variables} & x, y, z & \\
\\
\textit{program types} & \pi ::= & \tau_1 \parallel \tau_2 \\
\textit{programs} & p ::= & e_1 \parallel e_2 \\
\textit{types} & \tau ::= & 1 \mid \alpha \mid \tau_1 \to \tau_2 \mid \forall \alpha.\tau \mid \chi^k\ \overline{\tau}_k \\
\textit{terms} & e ::= & \iota \mid x \mid \lambda x{:}\tau.e \mid e_1\ e_2 \mid \Lambda\alpha.e \mid e[\tau] \\
\textit{type constructors} & \chi ::= & \mathsf{M}^1 \\
\textit{data constructors} & \iota ::= & \langle\rangle_0^0 \mid \mathsf{unit}_1^1 \mid \mathsf{bind}_2^2 \mid \mathsf{recv}_1^0 \mid \mathsf{send}_1^1 \\
\\
\textit{values} & v ::= & \lambda x{:}\tau.e \mid \Lambda\alpha.e \\
 & \mid & (\iota_m^n)\overline{[\tau]}_k \qquad \text{(where } k \leq m\text{)} \\
 & \mid & (\iota_m^n)\overline{[\tau]}_m\ \overline{v}_k \qquad \text{(where } k \leq n\text{)} \\
\textit{program results} & w ::= & \mathsf{unit}[\tau_1]\ v_1 \parallel \mathsf{unit}[\tau_2]\ v_2 \\
\textit{evaluation contexts} & E ::= & [\,] \mid E\ e \mid v\ E \mid E[\tau] \\
\textit{thread contexts} & T ::= & [\,] \mid \mathsf{bind}\,\overline{[\tau]}_2\ T\ v \\
\\
\textit{type contexts} & \Delta ::= & \alpha_1, \ldots, \alpha_n \\
\textit{value contexts} & \Gamma ::= & x_1{:}\tau_1, \ldots, x_n{:}\tau_n
\end{array}
$$

Figure 3. Syntax of $\lambda^{\mathrm{F}\|\mathrm{F}}$

by means of a library, capturing the essence of our implementation in §3. We then prove that the library enforces the properties we desire for our session-type system. We focus on the single-channel case for simplicity, but this approach should generalize to multiple channels.

A $\lambda^{\mathrm{F}\|\mathrm{F}}$ program is a pair of two threads that reduce concurrently and may communicate via an implicit channel. Each thread is a term in a variant of System F (Girard 1971, 1972; Reynolds 1974) extended with type constructors and several data constructor constants. The syntax of $\lambda^{\mathrm{F}\|\mathrm{F}}$ appears in Figure 3.

Each data constructor ι^n_m is equipped with two arities, which are used in the definition of normal forms. Subscript m is the number of type applications admitted by the constructor, and superscript n is the number of applications until the resulting value is fully saturated. For example, since bind^2_2 has type arity 2 and value arity 2, all of bind^2_2, $\mathsf{bind}^2_2[\tau_1]$, $\mathsf{bind}^2_2[\tau_1][\tau_2]$, $\mathsf{bind}^2_2[\tau_1][\tau_2]\ v_1$, and $\mathsf{bind}^2_2[\tau_1][\tau_2]\ v_1\ v_2$ are syntactic values. For legibility, we generally elide arities in examples and discussion.

The type constructor M^1 with data constructors unit and bind form a monad, which is used to sequence communication between the two threads. As the language is call-by-value, this sequencing may seem redundant, but its purpose will become apparent when we tackle type soundness. Normal forms for programs, denoted by the syntactic metavariable w, are pairs of injected values $\mathsf{unit}[\tau_1]\ v_1 \parallel \mathsf{unit}[\tau_2]\ v_2$.

We write $\mathrm{FTV}(\tau)$ for the free type variables of τ, defined in the standard way. We write $N\{M/X\}$ for the capture-avoiding substitution of M for X in N. We use the notation $\overline{M}_k$ as shorthand for a repetition of indexed syntactic objects, $M_1\ M_2\ \cdots\ M_k$. We consider

both terms and types to be equivalent up to alpha conversion.

We give the static semantics for $\lambda^{\mathrm{F}\|\mathrm{F}}$ in Figure 4. The type system is largely conventional. The rules for typing variables, abstractions, applications, type abstractions, and type applications are as in System F. Each data constructor is given an ordinary type that agrees with its arity. The program typing judgment T-PROG requires that each thread e_i in a program $e_1 \parallel e_2$ have a type $\mathsf{M}\,\tau_i$, in which case the whole program is then given the type $\tau_1 \parallel \tau_2$. Note that no rule *connects* the types between the threads in any way—this is important.

An evaluation context semantics may be found in Figure 5. The reduction relation for threads ($\longrightarrow$) is conventional, but the

TYPES: $\Delta \vdash \tau$

$$\frac{}{\Delta \vdash \tau}\ (\mathrm{FTV}(\tau) \subseteq \Delta)$$

CONSTANTS: $\mathrm{TypeOf}(\iota) = \tau$

$$\mathrm{TypeOf}(\langle\rangle) = 1 \qquad \mathrm{TypeOf}(\mathsf{unit}) = \forall\alpha.\alpha \to \mathsf{M}\,\alpha$$
$$\mathrm{TypeOf}(\mathsf{bind}) = \forall\alpha.\forall\alpha'.\mathsf{M}\,\alpha \to (\alpha \to \mathsf{M}\,\alpha') \to \mathsf{M}\,\alpha'$$
$$\mathrm{TypeOf}(\mathsf{send}) = \forall\alpha.\alpha \to \mathsf{M}\,1 \qquad \mathrm{TypeOf}(\mathsf{recv}) = \forall\alpha.\mathsf{M}\,\alpha$$

TERMS: $\Delta;\Gamma \vdash e : \tau$

$$\frac{}{\Delta;\Gamma \vdash \iota : \mathrm{TypeOf}(\iota)} \qquad \frac{}{\Delta;\Gamma,x{:}\tau \vdash x : \tau}$$

$$\frac{\Delta \vdash \tau_1 \quad \Delta \vdash \tau_2 \quad \Delta;\Gamma,x{:}\tau_1 \vdash e : \tau_2}{\Delta;\Gamma \vdash \lambda x{:}\tau_1.e : \tau_1 \to \tau_2}\ (x \notin \mathrm{dom}\,\Gamma)$$

$$\frac{\Delta;\Gamma \vdash e_1 : \tau' \to \tau \quad \Delta;\Gamma \vdash e_2 : \tau'}{\Delta;\Gamma \vdash e_1\ e_2 : \tau} \qquad \frac{\Delta,\alpha;\Gamma \vdash e : \tau}{\Delta;\Gamma \vdash \Lambda\alpha.e : \forall\alpha.\tau}\ (\alpha \notin \Delta)$$

$$\frac{\Delta \vdash \tau \quad \Delta;\Gamma \vdash e : \forall\alpha.\tau'}{\Delta;\Gamma \vdash e[\tau] : \tau'\{\tau/\alpha\}}$$

PROGRAMS: $\vdash p : \pi$

$$\frac{\vdash e_1 : \mathsf{M}\,\tau_1 \quad \vdash e_2 : \mathsf{M}\,\tau_2}{\vdash e_1 \parallel e_2 : \tau_1 \parallel \tau_2}\ \text{(T-PROG)}$$

Figure 4. Type system for $\lambda^{\mathrm{F}\|\mathrm{F}}$

reduction relation for programs ($\Longrightarrow$) is slightly more interesting. The first two rules are structural: R-THREAD allows threads to step according to ($\longrightarrow$), and R-FLIP means that reductions that apply to a pair of threads also apply to their transposition. Rule R-ID implements M's left identity. Because the type system fails to enforce any interthread properties, we require that rule R-SEND, which implements communication between the threads, perform a runtime check. Only when send and recv agree on the type parameter τ is communication allowed to proceed.

This last bug/feature models the untyped channel primitives *unsafeUChanSend* and *unsafeUChanRecv* that we assume in our Haskell implementation. The dynamic behavior in Haskell differs slightly from that in our model. In Haskell, if communication co-

erces an `Int` to type `Int -> Int`, we are unlikely to notice immediately, but subsequent behavior is badly undefined. For the sake of expedience, we choose in $\lambda^{\mathrm{F}\|\mathrm{F}}$ to get stuck as soon as possible rather than proceed in an inconsistent state.

It should be increasingly clear at this point that $\lambda^{\mathrm{F}\|\mathrm{F}}$ exhibits only a limited form of type soundness. It enjoys subject reduction because of the dynamic check in R-SEND, but the progress lemma is impeded by the presence of *communication faults*.

Definition 6.1 (Stuck Programs)**.** *A program configuration p is **stuck** if it is not a result w and there is no p' such that $p \Longrightarrow p'$.*

Definition 6.2 (Communication Faults)**.** *A program configuration p is **stuck in communication** if it is stuck in one of the following four situations.*

1. *Both threads are attempting to send a message, as in*

$$T_1[\mathsf{send}[\tau_1]\ v_1] \parallel T_2[\mathsf{send}[\tau_2]\ v_2].$$

THREAD REDUCTIONS: $e \longrightarrow e$

$$\overline{(\lambda x{:}\tau.e)\ v \longrightarrow e\{v/x\}} \qquad \overline{(\Lambda\alpha.e)[\tau] \longrightarrow e\{\tau/\alpha\}}$$

$$\frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']}$$

PROGRAM REDUCTIONS: $p \Longrightarrow p$

$$\text{(R-THREAD)}\quad \frac{e_1 \longrightarrow e_1'}{e_2 \parallel e_1 \Longrightarrow e_2 \parallel e_1'} \qquad \text{(R-FLIP)}\quad e_2 \parallel e_1 \Longrightarrow e_2' \parallel e_1'$$

$$\frac{}{e_1 \parallel e_2 \Longrightarrow e_1' \parallel e_2} \qquad \frac{}{e_1 \parallel e_2 \Longrightarrow e_1' \parallel e_2'}$$

(R-ID)

$$\frac{}{T_1[\mathsf{bind}\,\overline{[\tau]}_2\ (\mathsf{unit}[\tau_3]\ v_1)\ v_2] \parallel e \Longrightarrow T_1[v_2\ v_1] \parallel e}$$

(R-SEND)

$$\frac{}{T_1[\mathsf{recv}[\tau]] \parallel T_2[\mathsf{send}[\tau]\ v] \Longrightarrow T_1[\mathsf{unit}[\tau]\ v] \parallel T_2[\mathsf{unit}[1]\ \langle\rangle]}$$

Figure 5. Operational semantics for $\lambda^{\mathrm{F}\parallel\mathrm{F}}$

2. *Both threads are attempting to receive a message, as in*

$$T_1[\mathsf{recv}[\tau_1]] \parallel T_2[\mathsf{recv}[\tau_2]]$$

3. *One thread is attempting to communicate but the other is finished reducing, as in*

$$T_1[\mathsf{recv}[\tau_1]] \parallel \mathsf{unit}[\tau_2]\ v_2,$$
$$T_1[\mathsf{send}[\tau_1]\ v_1] \parallel \mathsf{unit}[\tau_2]\ v_2,$$

   or their transpositions over $\parallel$.

4. *The threads are ready to communicate but their types do not agree, as in*

$$T_1[\mathsf{recv}[\tau_1]] \parallel T_2[\mathsf{send}[\tau_2]\ v_2]$$

   or its transposition, where $\tau_1 \neq \tau_2$.

We have a soundness proof (Wright and Felleisen 1994) of the following theorem:

Theorem 6.3 (Soundness for $\lambda^{\mathrm{F}\|\mathrm{F}}$)**.** *If* $\vdash p : \pi$*, then either:*

- *p diverges,*
- $p \Longrightarrow^* w$ *where* $\vdash w : \pi$*, or*
- *p eventually becomes stuck in communication.*

6.1 The Session Types Library for $\lambda^{\mathrm{F}\|\mathrm{F}}$

We now define a library for $\lambda^{\mathrm{F}\|\mathrm{F}}$ that adds session types and, we claim, a progress lemma. It is a library in the sense that it defines several new type constructors, abstractly, in terms of $\lambda^{\mathrm{F}\|\mathrm{F}}$ types and several new constants in terms of $\lambda^{\mathrm{F}\|\mathrm{F}}$ terms. We now require that programs access the old primitives unit, bind, send, and recv *only* through the library. We describe the library informally for a few paragraphs before making things precise.

Library interface. The library's signature gives the new type constructors with their arities and the new constants with their types:

$$\chi_\ell ::= \underline{\mathsf{S}}^3 \mid \cdot \underline{?} \cdot \mid \cdot \underline{!} \cdot \mid \cdot \underline{\oplus} \cdot \mid \cdot \underline{\&} \cdot \mid \underline{\varepsilon}^0$$

$$\begin{array}{rcl}
\underline{\mathsf{unit}}_\ell & : & \forall\beta.\forall\alpha.\underline{\mathsf{S}}\,\beta\;\beta\;\alpha \\
\underline{\mathsf{bind}}_\ell & : & \forall\beta.\forall\beta'.\forall\beta''.\forall\alpha.\forall\alpha. \\
 & & \quad \underline{\mathsf{S}}\,\beta\;\beta'\;\alpha \to (\alpha \to \underline{\mathsf{S}}\,\beta'\;\beta''\;\alpha') \to \underline{\mathsf{S}}\,\beta\;\beta''\;\alpha' \\
\underline{\mathsf{recv}}_\ell & : & \forall\beta.\forall\alpha.\underline{\mathsf{S}}\,(\alpha\;\underline{?}\;\beta)\;\beta\;\alpha \\
\underline{\mathsf{send}}_\ell & : & \forall\beta.\forall\alpha.\alpha \to \underline{\mathsf{S}}\,(\alpha\;\underline{!}\;\beta)\;\beta\;1 \\
\underline{\mathsf{sel1}}_\ell & : & \forall\beta_1.\forall\beta_2.\underline{\mathsf{S}}\,(\beta_1 \,\underline{\oplus}\, \beta_2)\;\beta_1\;1 \\
\underline{\mathsf{sel2}}_\ell & : & \forall\beta_1.\forall\beta_2.\underline{\mathsf{S}}\,(\beta_1 \,\underline{\oplus}\, \beta_2)\;\beta_2\;1 \\
\underline{\mathsf{offer}}_\ell & : & \forall\beta_1.\forall\beta_2.\forall\beta'.\forall\alpha. \\
 & & \quad \underline{\mathsf{S}}\beta_1\;\beta'\;\alpha \to \underline{\mathsf{S}}\beta_2\;\beta'\;\alpha \to \underline{\mathsf{S}}\,(\beta_1 \,\underline{\&}\, \beta_2)\;\beta'\;\alpha
\end{array}$$

These types correspond to the types given in Haskell in §3.

The library also adds a new type judgment for duality, which corresponds to the Haskell type class `Dual`:

$$\text{(D-EPS)}\quad \frac{}{\underline{\varepsilon} \bowtie \underline{\varepsilon}} \qquad \text{(D-SEND)}\quad \frac{\tau_2 \bowtie \tau_2'}{\tau_1 \mathbin{\underline{!}} \tau_2 \bowtie \tau_1 \mathbin{\underline{?}} \tau_2'} \qquad \text{(D-RECV)}\quad \frac{\tau_2 \bowtie \tau_2'}{\tau_1 \mathbin{\underline{?}} \tau_2 \bowtie \tau_1 \mathbin{\underline{!}} \tau_2'}$$

$$\text{(D-CHOOSE)}\quad \frac{\tau_1 \bowtie \tau_1' \qquad \tau_2 \bowtie \tau_2'}{\tau_1 \mathbin{\underline{\oplus}} \tau_2 \bowtie \tau_1' \mathbin{\underline{\&}} \tau_2'} \qquad \text{(D-OFFER)}\quad \frac{\tau_1 \bowtie \tau_1' \qquad \tau_2 \bowtie \tau_2'}{\tau_1 \mathbin{\underline{\&}} \tau_2 \bowtie \tau_1' \mathbin{\underline{\oplus}} \tau_2'}$$

It redefines the program typing rule R-PROG using the new duality relation to ensure that the threads' session types are dual:

$$\text{(T_ℓ-PROG)}\quad \frac{\vdash e_1 : \underline{\mathsf{S}}\,\tau_1\,\underline{\varepsilon}\,\tau_1' \qquad \vdash e_2 : \underline{\mathsf{S}}\,\tau_2\,\underline{\varepsilon}\,\tau_2' \qquad \tau_1 \bowtie \tau_2}{\vdash_\ell e_1 \parallel e_2 : \tau_1' \parallel \tau_2'}$$

Note that while the premises enforce that τ_1 and τ_2 be dual, these types are not mentioned in the conclusion. This makes the subject reduction lemma easier to state and prove.

Library implementation. The new types and constants are defined in terms of $\lambda^{\mathrm{F}\|\mathrm{F}}$:

$$\underline{\mathsf{S}}\;\beta\;\beta'\;\alpha \triangleq \mathsf{M}\,\alpha \qquad \alpha \mathbin{\underline{?}} \beta \triangleq 1 \qquad \alpha \mathbin{\underline{!}} \beta \triangleq 1$$

$$\underline{\varepsilon} \triangleq 1 \qquad \beta \mathbin{\underline{\oplus}} \beta' \triangleq 1 \qquad \beta \mathbin{\underline{\&}} \beta' \triangleq 1$$

$$\underline{\mathsf{unit}}_\ell \triangleq \Lambda\beta.\,\mathsf{unit} \qquad \underline{\mathsf{bind}}_\ell \triangleq \Lambda\beta.\Lambda\beta'.\Lambda\beta''.\,\mathsf{bind}$$

$$\underline{\mathsf{send}}_\ell \triangleq \Lambda\beta.\,\mathsf{send} \qquad \underline{\mathsf{recv}}_\ell \triangleq \Lambda\beta.\,\mathsf{recv}$$

$$\underline{\mathsf{sel1}}_\ell \triangleq \Lambda\beta_1.\Lambda\beta_2.\,\mathsf{send}[\forall\gamma.\gamma \to \gamma \to \gamma](\Lambda\gamma.\lambda t{:}\gamma.\lambda f{:}\gamma.t)$$

$$\underline{\mathsf{sel2}}_\ell \triangleq \Lambda\beta_1.\Lambda\beta_2.\,\mathsf{send}[\forall\gamma.\gamma \to \gamma \to \gamma](\Lambda\gamma.\lambda t{:}\gamma.\lambda f{:}\gamma.f)$$

$$\begin{aligned}\underline{\mathsf{offer}}_\ell \triangleq\ & \Lambda\beta_1.\Lambda\beta_2.\Lambda\beta'.\Lambda\alpha.\lambda x_1{:}\,\mathsf{M}\,\alpha.\lambda x_2{:}\,\mathsf{M}\,\alpha.\\ & \quad \mathsf{bind}[\forall\gamma.\gamma \to \gamma \to \gamma][\alpha]\\ & \qquad (\mathsf{recv}[\forall\gamma.\gamma \to \gamma \to \gamma])\\ & \qquad (\lambda z{:}(\forall\gamma.\gamma \to \gamma \to \gamma).z[\mathsf{M}\,\alpha]\ x_1\ x_2)\end{aligned}$$

The library's dynamics derive directly from the above definitions.

6.2 A Semantics for the Library

We are now ready to state the principal claim of this section:

Claim. *If a $\lambda^{F\|F}$ program is written using the new library, with no mention of the primitives* unit, bind, recv, *nor* send, *and furthermore, if the program has a type according to the new rule* T_ℓ-PROG, *then the program either converges to a program result or diverges. In particular, well-typed programs written with the library do not have communication faults.*

We formalize this intuition with a new calculus $\lambda_\ell^{\mathrm{F}\|\mathrm{F}}$, by which we give a semantics to the library directly rather than in terms of $\lambda^{\mathrm{F}\|\mathrm{F}}$. The changes to $\lambda_\ell^{\mathrm{F}\|\mathrm{F}}$ from $\lambda^{\mathrm{F}\|\mathrm{F}}$ are summarized in Figure 6. The type constructors (χ) are the same as for the library. The data constructors (ι) are the same as the constants defined in the library, and they are given the same types that they have in the library. Program results (w) and thread contexts (T) are adjusted for the new data constructors.

The type judgment for terms is as for $\lambda^{\mathrm{F}\|\mathrm{F}}$, and we use the new rule T_ℓ-PROG for typing programs. Similarly, the small-step

NEW SYNTAX:

$$\begin{array}{rrcl} \textit{type constructors} & \chi & ::= & \underline{\mathsf{S}}^3 \mid \cdot\,\underline{?}\,\cdot \mid \cdot\,\underline{!}\,\cdot \mid \cdot\,\underline{\oplus}\,\cdot \mid \cdot\,\underline{\&}\,\cdot \mid \underline{\varepsilon}^0 \\ \textit{value constructors} & \iota & ::= & \langle\rangle_0^0 \mid \underline{\mathsf{unit}}_\ell^1 \mid \underline{\mathsf{bind}}_\ell^2 \mid \underline{\mathsf{recv}}_\ell^0 \mid \underline{\mathsf{send}}_\ell^1 \mid \underline{\mathsf{sel1}}_\ell^0 \mid \underline{\mathsf{sel2}}_\ell^0 \mid \underline{\mathsf{offer}}_\ell^2 \\ \textit{program results} & w & ::= & \underline{\mathsf{unit}}_\ell[\tau_1][\tau_1'] \; v_1 \parallel \underline{\mathsf{unit}}_\ell[\tau_2][\tau_2'] \; v_2 \\ \textit{thread contexts} & T & ::= & [\,] \mid \underline{\mathsf{bind}}_\ell\,\overline{[\tau]}_5 \; T \; v \end{array}$$

NEW DYNAMICS:

$$\mathrm{Evolve}([\,],\tau') = [\,] \qquad \mathrm{Evolve}(\underline{\mathsf{bind}}_\ell[\tau]\overline{[\tau]}_4 T\; v, \tau') = \underline{\mathsf{bind}}_\ell[\tau']\overline{[\tau]}_4\; (\mathrm{Evolve}(T,\tau'))\; v$$

(R_ℓ-THREAD)
$$\frac{e_1 \longrightarrow e_1'}{e_1 \parallel e_2 \Longrightarrow_\ell e_1' \parallel e_2}$$

(R_ℓ-FLIP)
$$\frac{e_2 \parallel e_1 \Longrightarrow_\ell e_2' \parallel e_1'}{e_1 \parallel e_2 \Longrightarrow_\ell e_1' \parallel e_2'}$$

(R_ℓ-ID)
$$\frac{}{T_1[\underline{\mathsf{bind}}_\ell\,\overline{[\tau]}_5(\underline{\mathsf{unit}}_\ell\,\overline{[\tau']}_2\; v_1)\; v_2] \parallel e_2 \Longrightarrow_\ell T_1[v_2\; v_1] \parallel e_2}$$

(R_ℓ-SEND)
$$\frac{\mathrm{Evolve}(T_1,\tau_1) = T_1' \qquad \mathrm{Evolve}(T_2,\tau_2) = T_2'}{T_1[\underline{\mathsf{recv}}_\ell[\tau_1][\tau]] \parallel T_2[\underline{\mathsf{send}}_\ell[\tau_2][\tau]v] \Longrightarrow_\ell T_1'[\underline{\mathsf{unit}}_\ell[\tau_1][\tau]\; v] \parallel T_2'[\underline{\mathsf{unit}}_\ell[\tau_2][1]\; \langle\rangle]}$$

(R_ℓ-SELi)
$$\frac{\mathrm{Evolve}(T_i,\tau_i) = T_1' \qquad \mathrm{Evolve}(T_2,\tau_i') = T_2'}{T_1[\underline{\mathsf{offer}}_\ell\,\overline{[\tau]}_4\; v_1\; v_2] \parallel T_2[\underline{\mathsf{sel}}_\ell\,\overline{[\tau']}_2] \Longrightarrow_\ell T_1'[v_i] \parallel T_2'[\underline{\mathsf{unit}}_\ell[\tau_i'][1]\; \langle\rangle]}\,(i \in \{1,2\})$$

Figure 6. Summary of changes for $\lambda_\ell^{\mathrm{F}\|\mathrm{F}}$

relation for threads ($\longrightarrow$) is the same as for $\lambda^{\mathrm{F}\|\mathrm{F}}$, but the small-step relation for programs ($\Longrightarrow$) needs revision. The structural rules R_ℓ-THREAD and R_ℓ-FLIP are unchanged, and R_ℓ-ID is merely updated to reflect the type parameters taken by library operations $\underline{\mathsf{unit}}_\ell$ and $\underline{\mathsf{bind}}_\ell$. The remaining rules are somewhat strange, because we need to adjust the type parameters in thread contexts. To see why this is necessary, consider the configuration

$$T_1\,[\underline{\mathsf{recv}}_\ell[\underline{\varepsilon}][1]] \parallel T_2[\underline{\mathsf{send}}_\ell[\underline{\varepsilon}][1]\langle\rangle].$$

The terms in the holes have types:

$$\vdash \underline{\mathsf{recv}}_\ell[\underline{\varepsilon}][1] : \underline{\mathsf{S}}\,(1\;\underline{?}\;\underline{\varepsilon})\;\underline{\varepsilon}\;1$$

$$\vdash \underline{\mathsf{send}}_\ell[\underline{\varepsilon}][1]\ \langle\rangle : \underline{\mathsf{S}}\,(1\ \underline{!}\ \underline{\varepsilon})\ \underline{\varepsilon}\ 1$$

The configuration takes a step to

$$T_1'[\underline{\mathsf{unit}}_\ell[\underline{\varepsilon}][1]\ \langle\rangle] \parallel T_2'[\underline{\mathsf{unit}}_\ell[\underline{\varepsilon}][1]\ \langle\rangle],$$

and now the terms in the holes have *different* types than before:

$$\vdash \underline{\mathsf{unit}}_\ell[\underline{\varepsilon}][1]\ \langle\rangle : \underline{\mathsf{S}}\ \underline{\varepsilon}\ \underline{\varepsilon}\ 1$$
$$\vdash \underline{\mathsf{unit}}_\ell[\underline{\varepsilon}][1]\ \langle\rangle : \underline{\mathsf{S}}\ \underline{\varepsilon}\ \underline{\varepsilon}\ 1$$

The thread contexts T_1 and T_2 therefore need to be adjusted to accomodate the new types. In a precise sense, this is because $\underline{\mathsf{S}}$ is an indexed monad. We use a function Evolve to update the first type parameter of each $\underline{\mathsf{bind}}_\ell$ in the thread contexts.

Let $\mathscr{L}[\![\cdot]\!]$ be a function that takes $\lambda_\ell^{\mathrm{F}\|\mathrm{F}}$ configurations, terms, and types to $\lambda^{\mathrm{F}\|\mathrm{F}}$ configurations, terms, and types by expanding the library definitions of $\lambda_\ell^{\mathrm{F}\|\mathrm{F}}$ types and constants. For example,

$$\mathscr{L}[\![\,\underline{\mathsf{recv}}_\ell[\underline{\varepsilon}][1] \parallel \underline{\mathsf{send}}_\ell[\underline{\varepsilon}][1]\ \langle\rangle\,]\!] = (\Lambda\beta.\,\mathsf{recv})[1][1] \parallel (\Lambda\beta.\,\mathsf{send})[1][1]\ \langle\rangle.$$

We may think of $\mathscr{L}[\![\cdot]\!]$ as inlining the library's definitions, or as a compiler from $\lambda_\ell^{\mathrm{F}\|\mathrm{F}}$ to $\lambda^{\mathrm{F}\|\mathrm{F}}$.

We have a Wright-Felleisen–style type soundness proof for $\lambda_\ell^{\mathrm{F}\|\mathrm{F}}$:

Lemma 6.4 (Soundness of $\lambda_\ell^{\mathrm{F}\|\mathrm{F}}$)**.** *If $\vdash_\ell p : \pi$ then either p diverges or $p \Longrightarrow_\ell^* w$ where $\vdash_\ell w : \pi$.*

We also proved an agreement lemma between $\lambda_\ell^{\mathrm{F}\|\mathrm{F}}$ and $\lambda^{\mathrm{F}\|\mathrm{F}}$:

Lemma 6.5 (Agreement)**.**

- *Compilation preserves types: If* $\vdash_\ell p : \pi$ *in* $\lambda_\ell^{F\|F}$*, then* $\vdash \mathscr{L}[\![p]\!] : \mathscr{L}[\![\pi]\!]$ *in* $\lambda^{F\|F}$*.*
- *Compilation preserves convergence: If* $p \Longrightarrow_\ell^* w$*, then there is some* w' *such that* $\mathscr{L}[\![p]\!] \Longrightarrow^* w'$*.*
- *Compilation preserves divergence: If p diverges in* $\lambda_\ell^{F\|F}$*, then* $\mathscr{L}[\![p]\!]$ *diverges in* $\lambda^{F\|F}$*.*

Together, these yield a soundness theorem about the $\lambda^{\mathrm{F}\|\mathrm{F}}$ session types library.

Theorem 6.6 (Library Soundness)**.** *If* $\vdash_\ell p : \pi$ *in* $\lambda_\ell^{F\|F}$*, then in* $\lambda^{F\|F}$ *either* $\mathscr{L}[\![p]\!]$ *diverges or* $\mathscr{L}[\![p]\!] \Longrightarrow^* w$ *where* $\vdash w : \mathscr{L}[\![\pi]\!]$*.*

7. Conclusion and Future Work

We have demonstrated that session types may be embedded in a variety of polymorphic programming languages, not merely as primitives in dedicated core calculi. With reasonable requirements on the host language, we provide[3] all the principal features of session types in a usable library. Yet much remains to be done in making session types practical for real-world use.

Several language features present problems or opportunities when combined with session types. How to combine session types with exceptions is an open question, for example. Raising exceptions is not a problem if we allow capabilities to be affine rather than linear, but it is unclear how exceptions may be caught and

[3] Literate Haskell is available on the web at `http://www.ccs.neu.edu/~tov/session-types/`.

communication resumed safely. One potential solution—and perhaps a profitable opportunity in its own right—could be to combine session types with software transactional memory (Harris et al. 2005). It might also be fruitful to integrate session types with CML-style events.

It would be interesting to investigate how some of our implementation techniques may be applied toward other embedded type systems. Indexed monads, in particular, seem especially promising. They are able to encode a variety of substructural type systems, including linear, affine, relevance, and ordered logics, and they allow *à la carte* selection of structural rules for particular types and operations. Indexed monads may also be useful in the embedding of effect systems, as in Kiselyov's (2007) Haskell implementation of Asai and Kameyama's (2007) polymorphic delimited continuations.

With the increasing prevalence of concurrent and distributed systems, more and better technology is needed to specify and check the behavior of communicating processes. Session types have the potential to play an important part in this story, and we believe this paper represents a step toward their wider availability.

Acknowledgments

We wish to thank Ryan Culpepper, Elizabeth Magner, Stevie Strickland, and Sam Tobin-Hochstadt for their helpful comments, and Alec Heller in particular for his sharp eye and perpetual encouragement.

References

J. Armstrong. Getting Erlang to talk to the outside world. In *Proc. 2002 ACM SIGPLAN workshop on Erlang*, pages 64–72. ACM Press, 2002.

K. Asai and Y. Kameyama. Polymorphic delimited continuations. In *Programming Languages and Systems*, volume 4807 of *Lecture Notes in Computer Science*, pages 239–254. Springer-Verlag, 2007.

R. Atkey. Parameterized notions of computation. In *Proc. Workshop on Mathematically Structured Functional Programming (MSFP'06)*. BCS, 2006.

E. Barendsen and S. Smetsers. Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical Structures in Computer Science*, 6(6):579–612, 1996.

M. M. T. Chakravarty, G. Keller, S. Peyton Jones, and S. Marlow. Associated types with class. In *Proc. 32nd Annual ACM Symposium on Principles of Programming Languages (POPL'05)*, pages 1–13. ACM Press, 2005.

R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Proc. 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01)*. ACM Press, 2001.

M. Dezani-Ciancaglini, N. Yoshida, A. Ahern, and S. Drossopolou. A distributed object-oriented language with session types. In *Proc. Symposium on Trustworthy Global Computing*, volume 3706 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.

M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopolou. Session types for object-oriented languages. In *Proc. European Conference on Object-Oriented Programming (ECOOP'06)*. Springer-Verlag, 2006.

M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based

communication in Singularity OS. In *Proc. 1st ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys'2006)*, pages 177–190. ACM Press, 2006.

S. J. Gay and M. J. Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2/3):191–225, 2005.

S. J. Gay and M. J. Hole. Types and subtypes for client-server interactions. In *Proc. 8th European Symposium on Programming (ESOP'99)*, volume 1576 of *Lecture Notes in Computer Science*, pages 74–90. Springer-Verlag, 1999.

S. J. Gay and V. T. Vasconcelos. Asynchronous functional session types. Technical Report 2007–251, Department of Computing, University of Glasgow, May 2007.

J.-Y. Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In J. E. Fenstad, editor, *Proc. Second Scandinavian Logic Symposium*, pages 63–92. North-Holland, 1971.

J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VI, 1972.

J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java™ Language Specification*. Addison Wesley, 3rd edition, 2005.

T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable memory transactions. In *Proc. ACM SIGPLAN Symposium on on Principles and Practice of Parallel Programming (PPoPP'05)*. ACM Press, 2005.

K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *Proc. European Symposium on Programming Languages and Systems*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer-Verlag, 1998.

M. P. Jones. Type classes with functional dependencies. In *Programming*

Languages and Systems, volume 1782 of *Lecture Notes in Computer Science*, pages 230–244. Springer-Verlag, 2000.

O. Kiselyov. Simple variable-state 'monad'. Mailing list message, December 2006. URL `http://www.haskell.org/pipermail/haskell/2006-December/018917.html`.

O. Kiselyov. Genuine shift/reset in Haskell98. Mailing list message, December 2007. URL `http://www.haskell.org/pipermail/haskell/2007-December/020034.html`.

O. Kiselyov, R. Lammel, and K. Schupke. Strongly typed heterogeneous collections. In *Proc. ACM SIGPLAN Workshop on Haskell (Haskell'04)*, pages 96–107. ACM Press, 2004.

M. Neubauer and P. Thiemann. An implementation of session types. In *Proc. 7th International Symposium on Practical Aspects of Declarative Languages (PADL'04)*, volume 3057 of *Lecture Notes in Computer Science*, pages 56–70, 2004.

S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proc. 23rd Annual ACM Symposium on Principles of Programming Languages (POPL'96)*, pages 295–308. ACM Press, 1996.

S. Peyton Jones, M. P. Jones, and E. Meijer. Type classes: An exploration of the design space, 1997.

R. Pucella and A. Heller. Capability-based calculi for session types. Unpublished manuscript, 2008.

J. H. Reppy. CML: A higher concurrent language. In *Proc. 1991 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'91)*, volume 26, pages 293–305. ACM Press, 1991.

J. C. Reynolds. Towards a theory of type structure. In *Proc. Colloque sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer-Verlag, 1974.

O. Shivers and M. Might. Continuations and transducer composition. In

Proc. 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'06), pages 295–307. ACM Press, 2006.

I. E. Sutherland and G. W. Hodgman. Reentrant polygon clipping. *Communications of the ACM*, 17(1):32–42, January 1974.

A. Vallecillo, V. T. Vasconcelos, and A. Ravara. Typing the behavior of objects and components using session types. In *Proc. International Workshop on Foundations of Coordination Languages and Software Architectures*, volume 68(3) of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2003.

V. T. Vasconcelos, S. J. Gay, and A. Ravara. Typechecking a multithreaded functional language with session types. *Theoretical Computer Science*, 368(1–2):64–87, 2006.

A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.