

# 1 A brief overview of the effect-system based encoding of the session-typed $\pi + \lambda$ -calculus into Haskell

The basis of the  $\pi + \lambda$  encoding in Haskell is a *graded monad* which is used to track session information. This is encoded via the data type:

```
data Session (s :: [*]) a = Session {getProcess :: IO a}
```

This wraps the *IO* monad in a binary type constructor *Session* with deconstructor *getProcess* :: *Session s a* → *IO a* and with a tag *s* used for type-level session information. In practise, we only need *getProcess* internally, so this can be hidden. We define a type-refined version of *getProcess* which allows us to run a computation only when the session environment is empty, that is, the process is closed with respect to channels.

```
run :: Session '[] a → IO a
run = getProcess
```

We can therefore run any session which will evaluate everything inside of the *IO* monad and actually performing the communication/spawning/etc.

Type-level session information will take the form of a list of mappings from channel names to session types, written like: `'[c :→ S, d :→ T, ...]`. This list will get treated as a set when we compose computations together, that is there are no duplicate mappings of some channel variable *c*, and the ordering will be normalise (this is a minor point and shouldn't affect too much here).

Session types are defined by the following type constructors:

```
-- Session types
data a !: s
data a :? s
data End
```

Duality of session type is then defined as a simple type-level function:

```
type family Dual s where
  Dual End = End
  Dual (t !: s) = t :? (Dual s)
  Dual (t :? s) = t !: (Dual s)
```

We define a (finite) set of channel name symbols *ChanNameSymbol* [this can be generalised away, but for some slightly subtle reasons mostly to do with CloudHaskell internals I have avoided the generalisation for the moment].

```
data ChanNameSymbol = X | Y | Z | C | D
data ChanName = Ch ChanNameSymbol | Op ChanNameSymbol
```

*ChanName* thus can describe the dual end of a channel with *Op*. These are just names for channels. Channels themselves comprise an encapsulated Concurrent Haskell channel [todo: convert to a Cloud Haskell channel]

**data** *Channel* (*n* :: *ChanName*) = forall *a* ◦ *Channel* (*C.Chan a*) **deriving** *Typeable*

## 1.1 $\pi$ -calculus part

We can now define the core primitives for send and receive, which have types:

*send* :: *Channel c* → *t* → *Session* '[*c* :→ *t* ! End] ()

*recv* :: *Channel c* → *Session* '[*c* :→ *t* ? End] *t*

These both take a named channel *Channel c* and return a *Session* computation indexed by the session environment '[*c* :→ *S*]' where *S* is either a send or receive action (terminated by *End*). These can then be composed using the **do**-notation, which sequentially composes session information. For example:

```
data Ping = Ping deriving Show
data Pong = Pong deriving Show
foo (c :: Channel (Ch C)) = do send c Ping
                               x ← recv c
                               return ((x + 1) :: Int)
```

This function is of type:

*foo* :: *Channel* (*Ch C*) → *Session* '[*Ch C* :→ (*Ping* ! (*Int* ? End))] *Int*

describing the session channel behaviour for *C*.

I've given an explicit name to the channel *c* here via a type signature, which names it as *Ch C*. This isn't strictly necessary here, but it leads to a huge simplification in the inferred type.

The *new* combinator then models  $\nu$ , which takes a function mapping from a pair of two channels names *Ch c* and *Op C* to a session with behaviour *s*, and creates a session where any mention to *Ch c* or *Op c* is removed:

```
new :: (Duality s c) ⇒
      ((Channel (Ch c), Channel (Op c)) → Session s b)
      → Session (Del (Ch c) (Del (Op c) s)) b
```

That is, the channels *Ch c* and *Op c* are only in scope for *Session s b*.

The *Duality* predicate asks whether the session environment *s* contains dual session types for channel *Ch C* and its dual *Op c*.

The session type encoding here is for an asynchronous calculus. In which case, the following is allowed:

```

foo2 = new (λ(c :: (Channel (Ch C)), c' :: (Channel (Op C))) →
  do Ping ← recv c'
  send c Ping
  return ())

```

To use channels properly, we need parallel composition. This is given by:

$$par :: (Disjoint\ s\ t) \Rightarrow Session\ s\ () \rightarrow Session\ t\ () \rightarrow Session\ (UnionS\ s\ t)\ ()$$

The binary predicate *Disjoint* here checks that *s* and *t* do not contain any of the same channels. *UnionS* takes the disjoint union of the two environments.

We can now define a complete example with communication:

```

server c = do Ping ← recv c
            print "Server: Got a ping"
process = new (λ(c, c') → par (send c Ping) (server c'))

```

Which we can run with *run process* getting "Server: Got a ping". Note that the types here are completely inferred, giving *process* :: *Session* '[]' ().

### 1.1.1 Delegation

So far we have dealt with only first-order channels (in the sense that they can pass only values and not other channels). We introduce a “delegate” type to wrap the session types of channels being passed:

```
data DelgS s
```

Channels can then be sent with *chSend* primitive:

$$chSend :: Channel\ c \rightarrow Channel\ d \rightarrow Session\ '[c : \rightarrow (DelgS\ s) :! End, d : \rightarrow s]\ ()$$

i.e., we can send a channel *d* with session type *s* over *c*.

The dual of this is a little more subtle. Receiving a delegated channel is given by combinator, which is not a straightforward monadic function, but takes a function as an argument:

$$chRecv :: Channel\ c \rightarrow (Channel\ d \rightarrow Session\ s\ a) \rightarrow Session\ (UnionS\ '[c : \rightarrow (DelgS\ (Lookup\ s\ d)) :? (Lookup\ s\ c)] (Del\ d\ s))\ a$$

Given a channel *c*, and a computation which binds channel *d* to produces behaviour *c*, then this is provided by receiving *d* over *c*. Thus the resulting computation is the union of *c* mapping to the session type of *d* in the session environment *s*, composed with the *s* but with *d* deleted (removed).

Here is an example using delegation. Consider the following process *server2* which receives a channel *d* on *c*, and then seds a ping on it:

```

server2 c = chRecv c
            (λ(d :: Channel (Ch D)) → send d Ping)

```

(Note, I have had to include explicit types to give a concrete name to the channel  $d$ , this is an unfortunate artefact of the current encoding, but not too bad from a theoretical perspect).

The type of *server2* is inferred as:

$server2 :: Channel\ c \rightarrow Session\ '[c : \rightarrow (DelgS\ (Ping : !End) : ?Lookup\ '[ 'Ch\ ' D : \rightarrow (Ping : !End)]\ c)]\ ()$

We then define a client to interact with this that binds  $d$  (and its dual  $d'$ ), then sends  $d$  over  $c$  and waits to receive a ping on  $d'$

```
client2 (c :: Channel (Ch C)) =
  new (\(d :: (Channel (Ch D)), d') →
    do chSend c d
      Ping ← recv d'
      print "Client: got a ping")
```

This has inferred type:

$client2 :: Dual\ s \sim (Ping : ?End) \Rightarrow Channel\ ('Ch\ ' C) \rightarrow Session\ '[ 'Ch\ ' C : \rightarrow (DelgS\ s : !End)]\ ()$

The type constraint says that the dual of  $s$  is a session that receives a *Ping*, so  $s$  is  $Ping : !End$ .

We then compose *server2* and *client2* in parallel, binding the channels  $c$  and its dual  $c'$  to give to client and server.

$process2 = new\ (\lambda(c, c') \rightarrow par\ (client2\ c)\ (server2\ c'))$

This type checks and can be then run (*run process2*) yielding "Client: got a ping".

## 1.2 $\lambda$ -part

Since we are studying the  $\pi + \lambda$ -calculus, we can abstract over channels with linear functions. So far we have abstracted over channels, but not in an *operational sense*- think of this more as let-binding style substitution (cut). We now introduce linear functions which can abstract over channels (and the session types of those channels, which the previous form of abstraction above **doesn't do**, it just abstracts over names, not the session types associated with their names).

First, we abstract functions via a type constructor *Abs*

**data**  $Abs\ t\ s = Abs\ (Proxy\ s)\ (forall\ c \circ (Channel\ c \rightarrow Session\ (UnionS\ s\ '[c : \rightarrow t])\ ()))$

The *Abs* data constructor takes a function of type  $forall\ c \circ (Channel\ c \rightarrow Session\ (UnionS\ s\ '[c : \rightarrow t])\ ())$ , that is, a function from *universally quantified* channel name  $c$  to a *Session* environment  $s$  where  $c : \rightarrow t$  is a member). Since *UnionS* is a non-injective function we also need a (trivial) type annotation that explains exactly what is the remaining channel- this is *Proxy s* (I'll shown an

example in the moment). This returns a result  $Abs\ t\ s$  which describes a function which takes some channel with session type  $t$  and has session environment  $s$ , cf.

$$\frac{\Delta, c : T \vdash C : \Diamond}{\Delta \vdash \lambda c. C : T \multimap \Diamond}$$

This can then be applied by the following primitive

$appH :: Abs\ t\ s \rightarrow Channel\ c \rightarrow Session\ (UnionS\ s\ '[c : \rightarrow t])\ ()$   
 $appH\ (Abs\ _\ k)\ c = k\ c$

Thus, given a linear session function  $Abs\ t\ s$  and some channel  $c$  then we get a session with environment  $s$  and a mapping  $c : \rightarrow t$ . Here's an example: a client abstract over a channel, and then applies it within the same process:

$client4\ (c :: Channel\ (Ch\ C)) = \mathbf{do}$   
 $\mathbf{let}\ f = Abs\ (Proxy :: (Proxy\ '[[]]))\ (\lambda c \rightarrow send\ c\ Ping)$   
 $appH\ f\ c$

This simply has type  $client4 :: Channel\ ('Ch\ '\ C) \rightarrow Session\ '[ 'Ch\ '\ C : \rightarrow (Ping :! End)]\ ()$ . We can then interact with this in a usual straightforward way.

$process4 = new\ (\lambda(c, c') \rightarrow (client4\ c)\ 'par'\ (\mathbf{do}\ \{x \leftarrow recv\ c'; print\ x\}))$