# 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:

$$\mathbf{data}\ \mathit{Session}\ (s :: [*])\ a = \mathit{Session}\ \{\mathit{getProcess} :: \mathit{IO}\ a\}$$

This wraps the *IO* monad in a binary type constructor *Session* with deconstructor $\mathit{getProcess} :: \mathit{Session}\ s\ a \to \mathit{IO}\ a$ 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.

$$\begin{array}{l} \mathit{run} :: \mathit{Session}\ `[\,]\ a \to \mathit{IO}\ a \\ \mathit{run} = \mathit{getProcess} \end{array}$$

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: $`[c :\!\to S, d :\!\to T, ...]$. 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:

$$\begin{array}{l} \quad \text{-- Session types} \\ \mathbf{data}\ a :!\ s \\ \mathbf{data}\ a :?\ s \\ \mathbf{data}\ \mathit{End} \end{array}$$

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

$$\begin{array}{l} \mathbf{type}\ \mathit{family}\ \mathit{Dual}\ s\ \mathbf{where} \\ \quad \mathit{Dual}\ \mathit{End} = \mathit{End} \\ \quad \mathit{Dual}\ (t :!\ s) = t :?\ (\mathit{Dual}\ s) \\ \quad \mathit{Dual}\ (t :?\ s) = t :!\ (\mathit{Dual}\ s) \\ \quad \mathit{Dual}\ (\mathit{Sel}\ l\ s\ t) = (\mathit{Dual}\ s) :\&\ (\mathit{Dual}\ t) \\ \quad \mathit{Dual}\ (\mathit{s1} :\&\ \mathit{s2}) = \mathit{Sel}\ \mathit{Sup}\ (\mathit{Dual}\ \mathit{s1})\ (\mathit{Dual}\ \mathit{s2}) \end{array}$$

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].

$$\begin{array}{l} \mathbf{data}\ \mathit{ChanNameSymbol} = X \mid Y \mid Z \mid C \mid D \mid \mathit{ForAll} \quad \text{-- reserved} \\ \mathbf{data}\ \mathit{ChanName} = \mathit{Ch}\ \mathit{ChanNameSymbol} \mid \mathit{Op}\ \mathit{ChanNameSymbol} \end{array}$$

*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]

$$\mathbf{data}\ \mathit{Channel}\ (n :: \mathit{ChanName}) = \mathit{forall}\ a \circ \mathit{Channel}\ (C.\mathit{Chan}\ a)\ \mathbf{deriving}\ \mathit{Typeable}$$

## 1.1 $\pi$-calculus part

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

$$\mathit{send} :: \mathit{Channel}\ c \to t \to \mathit{Session}\ `[\,c :\to t :!\ \mathit{End}\,]\ ()$$

$$\mathit{recv} :: \mathit{Channel}\ c \to \mathit{Session}\ `[\,c :\to t :?\ \mathit{End}\,]\ t$$

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

$$\begin{aligned}
&\mathbf{data}\ \mathit{Ping} = \mathit{Ping}\ \mathbf{deriving}\ \mathit{Show}\\
&\mathbf{data}\ \mathit{Pong} = \mathit{Pong}\ \mathbf{deriving}\ \mathit{Show}\\
&\mathit{foo}\ (c :: \mathit{Channel}\ (\mathit{Ch}\ C)) = \mathbf{do}\ \mathit{send}\ c\ \mathit{Ping}\\
&\qquad\qquad\qquad\qquad\qquad\quad x \leftarrow \mathit{recv}\ c\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathit{return}\ ((x + 1) :: \mathit{Int})
\end{aligned}$$

This function is of type:

$$\mathit{foo} :: \mathit{Channel}\ (\mathit{Ch}\ C) \to \mathit{Session}\ `[\,\mathit{Ch}\ C :\to (\mathit{Ping} :!\ (\mathit{Int} :?\ \mathit{End}))\,]\ \mathit{Int}$$

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:

$$\begin{aligned}
&\mathit{new} :: (\mathit{Duality}\ s\ c) \Rightarrow\\
&\qquad ((\mathit{Channel}\ (\mathit{Ch}\ c), \mathit{Channel}\ (\mathit{Op}\ c)) \to \mathit{Session}\ s\ b)\\
&\quad \to \mathit{Session}\ (\mathit{Del}\ (\mathit{Ch}\ c)\ (\mathit{Del}\ (\mathit{Op}\ c)\ s))\ b
\end{aligned}$$

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:

$$
\begin{aligned}
&\mathit{foo2} = \mathit{new}\ (\lambda(c :: (\mathit{Channel}\ (\mathit{Ch}\ C)), c' :: (\mathit{Channel}\ (\mathit{Op}\ C))) \rightarrow \\
&\quad \mathbf{do}\ \mathit{Ping} \leftarrow \mathit{recv}\ c' \\
&\qquad \mathit{send}\ c\ \mathit{Ping} \\
&\qquad \mathit{return}\ ())
\end{aligned}
$$

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

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

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:

$$
\begin{aligned}
&\mathit{server}\ c = \mathbf{do}\ \mathit{Ping} \leftarrow \mathit{recv}\ c \\
&\qquad\qquad\quad \mathit{print}\ \texttt{"Server: Got a ping"} \\
&\mathit{process} = \mathit{new}\ (\lambda(c, c') \rightarrow \mathit{par}\ (\mathit{send}\ c\ \mathit{Ping})\ (\mathit{server}\ c'))
\end{aligned}
$$

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

### 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:

$$\mathbf{data}\ \mathit{DelgS}\ s$$

Channels can then be sent with *chSend* primitive:

$$\mathit{chSend} :: \mathit{Channel}\ c \rightarrow \mathit{Channel}\ d \rightarrow \mathit{Session}\ `[\,c :\rightarrow (\mathit{DelgS}\ s) :!\ \mathit{End}, d :\rightarrow s\,]\ ()$$

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:

$$
\begin{aligned}
&\mathit{chRecv} :: \mathit{Channel}\ c \rightarrow (\mathit{Channel}\ d \rightarrow \mathit{Session}\ s\ a) \rightarrow \\
&\qquad\qquad \mathit{Session}\ (\mathit{UnionS}\ `[\,c :\rightarrow (\mathit{DelgS}\ (\mathit{Lookup}\ s\ d)) :?\ (\mathit{Lookup}\ s\ c)]\ (\mathit{Del}\ d\ s))\ a
\end{aligned}
$$

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:

$$
\begin{aligned}
&\mathit{server2}\ c = \mathit{chRecv}\ c \\
&\quad (\lambda(d :: \mathit{Channel}\ (\mathit{Ch}\ D)) \rightarrow \mathit{send}\ d\ \mathit{Ping})
\end{aligned}
$$

(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:

$$\mathit{server2} :: \mathit{Channel}\ c \to \mathit{Session}\ `[c{:\to}(\mathit{DelgS}\ (\mathit{Ping}{:}!\mathit{End}){:}?\mathit{Lookup}\ `[\texttt{'Ch '}\ D{:\to}(\mathit{Ping}{:}!\mathit{End})]\ c)]\ ()$$

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 $d'$

$$\begin{array}{l} \mathit{client2}\ (c :: \mathit{Channel}\ (\mathit{Ch}\ C)) = \\ \quad \mathit{new}\ (\lambda(d :: (\mathit{Channel}\ (\mathit{Ch}\ D)), d') \to \\ \qquad \mathbf{do}\ \mathit{chSend}\ c\ d \\ \qquad\quad \mathit{Ping} \leftarrow \mathit{recv}\ d' \\ \qquad\quad \mathit{print}\ \texttt{"Client: got a ping"}) \end{array}$$

This has inferred type:

$$\mathit{client2} :: \mathit{Dual}\ s \sim (\mathit{Ping}{:}?\mathit{End}) \Rightarrow \mathit{Channel}\ (\texttt{'Ch '}\ C) \to \mathit{Session}\ `[\texttt{'Ch '}\ C{:\to}(\mathit{DelgS}\ s{:}!\mathit{End})]\ ()$$

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

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

$$\mathit{process2} = \mathit{new}\ (\lambda(c, c') \to \mathit{par}\ (\mathit{client2}\ c)\ (\mathit{server2}\ c'))$$

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*

$$\mathbf{data}\ \mathit{Abs}\ t\ s = \mathit{forall}\ c \circ \mathit{Abs}\ (\mathit{Proxy}\ s)\ (\mathit{Channel}\ c \to \mathit{Session}\ (\mathit{UnionS}\ s\ `[c :\to t])\ ())$$

The *Abs* data constructor takes a function of type $(\mathit{Channel}\ c \to \mathit{Session}\ (\mathit{UnionS}\ s\ `[c{:\to} t])\ ())$, that is, a function from some channel $c$ to a *Session* environment $s$ where $c :\to t$ is a member). Since *UnionS* is a non-injective function we also need a type annotation that explains exactly what is the remaining channel- this is $\mathit{Proxy}\ s$ (I'll show an example in the moment). This returns a result $\mathit{Abs}\ t\ s$ 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

$appH :: Abs\ t\ s \rightarrow Channel\ c \rightarrow Session\ (UnionS\ s\ `[c :\rightarrow t])\ ()$
$appH\ (Abs\ \_\ k)\ c = \textbf{let}\ (Session\ s) = k\ (unsafeCoerce\ c)\ \textbf{in}\ Session\ s$

Whatever concrete name was used for the channel in the abstracted process is replaced by the channel name here.

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)) = do
  let f = Abs (Proxy :: (Proxy `[])) (λc → send c Ping)
  appH f c
```

This simply has type $client4 :: Channel\ (\texttt{'Ch '}\ C) \rightarrow Session\ `[\texttt{'Ch '}\ C :\rightarrow (Ping :!\ End)]\ ()$. We can then interfact with this in a usual straightforwad way.

```
process4 = new (λ(c, c′) → (client4 c) `par` (do {x ← recv c′; print x}))
```

A more complicated example reuses the abstraction in the client with different channels:

```
client5 (c :: Channel (Ch C)) (d :: (Channel (Ch D))) (x :: (Channel (Ch X))) = do
  let f = Abs (Proxy :: (Proxy `[(Ch X) :→ Pong :! End]))
    (λ(c :: (Channel (Ch D))) → do send c Ping
      send x Pong)
  appH f c
  appH f d
```

```
process5 = new (λ(c :: Channel (Ch C), c′) →
  new (λ(x :: Channel (Ch X), x′) →
    new (λ(d :: Channel (Ch D), d′) →
      (client5 c d x) `par`
        do v ← recv c′
          print v
          v ← recv x′
          print v
          v ← recv d′
          print v
          v ← recv x′
          print v
      )))
```

Dimtris example

$$
\begin{array}{l}
\mathit{client6}\ (c :: (\mathit{Channel}\ (\mathit{Ch}\ C)))\ (d :: (\mathit{Channel}\ (\mathit{Ch}\ D))) = \\
\quad \mathbf{do}\ \mathbf{let}\ f = \mathit{Abs}\ (\mathit{Proxy} :: \mathit{Proxy}\ `[(\mathit{Ch}\ C) :\!\to \mathit{Int}\ :!\ \mathit{End}]) \\
\qquad (\lambda(z :: (\mathit{Channel}\ (\mathit{Ch}\ Z))) \to (\mathit{send}\ z\ 42\ `\mathit{par}`\ \mathit{send}\ c\ 7)) \\
\qquad \mathit{appH}\ f\ d
\end{array}
$$

$$
\begin{array}{l}
\mathit{process6} = \mathit{new}\ (\lambda(c :: (\mathit{Channel}\ (\mathit{Ch}\ C)), c') \to \\
\quad \mathit{new}\ (\lambda(d :: (\mathit{Channel}\ (\mathit{Ch}\ D)), d') \to \\
\qquad \mathbf{do}\ \mathit{client6}\ c\ d \\
\qquad\quad \mathit{v1} \leftarrow \mathit{recv}\ c' \\
\qquad\quad \mathit{v2} \leftarrow \mathit{recv}\ d' \\
\qquad\quad \mathit{return}\ (\mathit{v1} + \mathit{v2})))
\end{array}
$$

## 1.3 Branching and choice

To encode branching and choice, we introduce binary branch/select (from which more complicated branch/select can be encoded) with two labels:

$$
\begin{array}{l}
\mathbf{data}\ \mathit{Left} \\
\mathbf{data}\ \mathit{Right} \\
\mathbf{data}\ \mathit{Sup}
\end{array}
$$

$$
\begin{array}{l}
\mathbf{data}\ \mathit{Label}\ l\ \mathbf{where} \\
\quad \mathit{LeftL} :: \mathit{String} \to \mathit{Label}\ \mathit{Left} \\
\quad \mathit{RightL} :: \mathit{String} \to \mathit{Label}\ \mathit{Right}
\end{array}
$$

The label data constructors *LeftL* and *RightL* also take string parameters for convenience (to act as comments in the code).

Note that whilst 'Sup' is a viable type-level label, there is no way to construct a label value with this type index. This is used for subtyping, where *Sup* represents a selection type which is a supertype.

Selection and branching session types are provided by the following two type constructors respectively:

$$
\begin{array}{l}
\mathbf{data}\ \mathit{Sel}\ l\ s\ t \\
\mathbf{data}\ s :\&\ t
\end{array}
$$

Select then has the type:

$$\mathit{select} :: \mathit{Channel}\ c \to \mathit{Label}\ l \to \mathit{Session}\ `[c :\!\to \mathit{Sel}\ l\ \mathit{End}\ \mathit{End}]\ ()$$

The idea is that, given a channel $c$, and a label $l$, then a session is created with a select session type for label $l$. Any computations that get composed after that use $c$ will add their session types into branch corresponding to the label. For example:

$$\begin{array}{l} \mathit{foo3}\ (c :: (\mathit{Channel}\ (\mathit{Ch}\ C))) = \\ \quad \mathbf{do}\ \mathit{select}\ c\ (\mathit{LeftL}\ \texttt{"l"}) \\ \qquad v \leftarrow \mathit{recv}\ c \\ \qquad \mathit{send}\ c\ (42 :: \mathit{Int}) \end{array}$$

*foo3* has the inferred type:

$$\mathit{foo3} :: \mathit{Channel}\ (\texttt{'Ch '}\ C) \to \mathit{Session}\ \text{`}[\texttt{'Ch '}\ C{:}{\to}\mathit{Sel}\ \mathit{Left}\ (t{:}?(\mathit{Int}{:}!\mathit{End}))\ \mathit{End}]\ ()$$

That is, we see that after selecting the left branch, then $c$ is used to receive some $t$ and then send an *Int*.

Branching then has the following type:

$$\begin{array}{l} \mathit{branch} :: ((\mathit{Del}\ c\ s1){\sim}(\mathit{Del}\ c\ s2)) \Rightarrow \\ \quad \mathit{Channel}\ c \to (\mathit{Label}\ \mathit{Left} \to \mathit{Session}\ s1\ a) \\ \qquad \to (\mathit{Label}\ \mathit{Right} \to \mathit{Session}\ s2\ a) \\ \qquad \to \mathit{Session}\ (\mathit{UnionS}\ (\mathit{Del}\ c\ s1)\ \text{`}[c :\to ((\mathit{Lookup}\ s1\ c) : \&\ (\mathit{Lookup}\ s2\ c))])\ a \end{array}$$

This is a bit more complicated. The first parameter is the channel over which a choice is being offered. Then come two continuations, the process if the left branch is taken and the process if the right branch is taken. Each gives a session environment $s1$ and $s2$ but apart from a session type for $c$, these must be equal (shown by the constraint $(\mathit{Del}\ c\ s1){\sim}(\mathit{Del}\ c\ s2)$. Finally, the returned session is that of $(\mathit{Del}\ c\ s1)$ unioned with $c$ mapping to the $(\mathit{Lookup}\ s1\ c) : \&$ $(\mathit{Lookup}\ s2\ c)$, i.e., the branching pair of the session types for $c$ in the left and right branches.

Here's an example:

$$\begin{array}{l} \mathit{process7} = \mathit{new}\ (\lambda(c :: (\mathit{Channel}\ (\mathit{Ch}\ C)), c') \to \\ \quad \mathbf{do}\ \{\mathit{select}\ c\ (\mathit{LeftL}\ \texttt{""}); \mathit{send}\ c\ 42\} \\ \qquad \text{`}\mathit{par}\text{`}\ \mathit{branch}\ c'\ (\lambda(\mathit{LeftL}\ \texttt{""}) \to \mathbf{do}\ \{v \leftarrow \mathit{recv}\ c'; \mathit{print}\ v\}) \\ \qquad\quad (\lambda(\mathit{RightL}\ \texttt{""}) \to \mathbf{do}\ \{\mathit{return}\ (); \mathit{return}\ ()\})) \end{array}$$

Then *run process7* yields 42 as expected.

$$\begin{array}{l} \mathit{selSupL} :: \mathit{Session}\ \text{`}[c :\to \mathit{Sel}\ l\ s\ \mathit{End}]\ () \to \mathit{Session}\ \text{`}[c :\to \mathit{Sel}\ \mathit{Sup}\ s\ t]\ () \\ \mathit{selSupL}\ s = \mathit{Session}\ \$\ \mathit{getProcess}\ s \end{array}$$

$$\begin{array}{l} \mathit{selSupR} :: \mathit{Session}\ \text{`}[c :\to \mathit{Sel}\ l\ \mathit{End}\ s]\ () \to \mathit{Session}\ \text{`}[c :\to \mathit{Sel}\ \mathit{Sup}\ t\ s]\ () \\ \mathit{selSupR}\ s = \mathit{Session}\ \$\ \mathit{getProcess}\ s \end{array}$$

# 2 Hotel booking scenario

$$\begin{array}{l} p :: (?\mathit{room} :: \mathit{String}, ?\mathit{credit} :: \mathit{Int}) \Rightarrow \\ \quad \mathit{Channel}\ (\mathit{Ch}\ X) \to \mathit{Abs}\ (\mathit{Int} :!\ (\mathit{End} : \&\ \mathit{End}))\ \text{`}[(\mathit{Ch}\ X) :\to \mathit{String} :!\ (\mathit{Int} :?\ (\mathit{Sel}\ \mathit{Sup}\ (\mathit{Int} :!\ \mathit{End})\ \mathit{End}) \end{array}$$

$$
\begin{array}{l}
p\ x = Abs\ (Proxy :: Proxy\ `[(Ch\ X) :\!\to String :!\ (Int :?\ (Sel\ Sup\ (Int :!\ End)\ End))]) \\
\quad (\lambda(y :: (Channel\ (Ch\ Y))) \to \\
\qquad \mathbf{do}\ send\ x\ ?\ room \\
\qquad\quad quote \leftarrow recv\ x \\
\qquad\quad send\ y\ quote \\
\qquad\quad branch\ y\ (\lambda(LeftL\ \texttt{"accept"}) \to selSupL\ \$\ \mathbf{do}\ select\ x\ (LeftL\ \texttt{"accept"}) \\
\qquad\qquad send\ x\ ?\ credit) \\
\qquad\qquad (\lambda(RightL\ \texttt{"reject"}) \to selSupR\ \$\ select\ x\ (RightL\ \texttt{"reject"})))
\end{array}
$$

{-foofoo (s1 :: (Channel (Ch Y))) (s2 :: (Channel (Ch Z))) (h1 :: (Channel (Ch C))) (h2 :: (Channel (