1 Overview of the effect-system based encoding of the session-typed π+λ-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:

```
  -- 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)
  Dual (Sel l s t) = (Dual s) :& (Dual t)
  Dual (s1 :& s2) = Sel Sup (Dual s1) (Dual s2)
```

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 | ForAll   -- reserved
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]

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

1.1 π-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:

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

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 $\mathit{Ch}\ C$. This isn't strictly necessary here, but it leads to a huge simplification in the inferred type.

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

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

That is, the channels $\mathit{Ch}\ c$ and $\mathit{Op}\ c$ are only in scope for $\mathit{Session}\ s\ b$.

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

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

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

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:

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

(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:

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

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'

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

This has inferred type:

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

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 λ-part

Since we are embedding the $\pi + \lambda$-calculus, we can abstract over channels with linear functions. So far we have defined functions which take particular names channels as arguments, but we have not abstracted over channels names. We now introduce linear functions which can abstract over channels (and the session types of those channels).

We abstract functions via a type constructor *Abs*

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

The *Abs* data constructor should be considered as abstract [it can be hidden]. Instead, we provide the follow constructor for (linear) abstractions:

$$\begin{aligned} &\mathit{absL} :: (\mathit{Proxy}\ s) \to (\mathit{Channel}\ c \to \mathit{Session}\ (\mathit{UnionS}\ s\ `[c :\to t])\ a) \to \mathit{Session}\ s\ (\mathit{Abs}\ t\ a) \\ &\mathit{absL}\ p\ f = \mathit{Session}\ (P.\mathit{return}\ \$\ \mathit{Abs}\ p\ f) \end{aligned}$$

The *absL* constructor takes a function of type $(\mathit{Channel}\ c \to \mathit{Session}\ (\mathit{UnionS}\ s\ `[c{:}\to t])\ a)$, 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 behaviour - this is $\mathit{Proxy}\ s$ (I'll show an example in the moment). This returns a result $\mathit{Session}\ s\ (\mathit{Abs}\ t\ a)$ which describes a function which takes some channel with session type t, returns a result of type a, and is embedded in a session with environment s, cf.

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

The main different here is that we can actual return a result (of type a), rather than just being a process $\diamond$. Dom: We need to decide whether we want to keep the ability for a value to returns from an abstraction, but for the moment it makes the hotel example easier (see Section 2).

These functions can then be applied by the following primitive:

$$\mathit{appL} :: \mathit{Abs}\ t\ a \to \mathit{Channel}\ c \to \mathit{Session}\ `[c :\to t]\ a$$

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 $\mathit{Abs}\ t\ a$ and some channel c then we get a session with mapping $c :\to t$. Here's an example: a client abstract over a channel, and then applies it within the same process:

$$
\begin{array}{l}
\mathit{client4}\ (c :: \mathit{Channel}\ (\mathit{Ch}\ C)) = \\
\quad \textbf{do}\ f \leftarrow \mathit{absL}\ (\mathit{Proxy} :: (\mathit{Proxy}\ `[]))\ (\lambda c \to \mathit{send}\ c\ \mathit{Ping}) \\
\qquad \mathit{appL}\ f\ c
\end{array}
$$

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

$$\mathit{process4} = \mathit{new}\ (\lambda(c, c') \to (\mathit{client4}\ c)\ `\mathit{par}`\ (\textbf{do}\ \{x \leftarrow \mathit{recv}\ c'; \mathit{print}\ x\}))$$

A more complicated example creates a closure over an other channel x:

$$
\begin{array}{l}
\mathit{client5}\ (c :: \mathit{Channel}\ (\mathit{Ch}\ C))\ (x :: (\mathit{Channel}\ (\mathit{Ch}\ X))) = \\
\quad \textbf{do}\ f \leftarrow \mathit{absL}\ (\mathit{Proxy} :: (\mathit{Proxy}\ `[(\mathit{Ch}\ X) :\to \mathit{Pong} :!\ \mathit{End}])) \\
\qquad\qquad\qquad (\lambda(c :: (\mathit{Channel}\ (\mathit{Ch}\ D))) \to \textbf{do}\ \mathit{send}\ c\ \mathit{Ping} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{send}\ x\ \mathit{Pong}) \\
\qquad \mathit{appL}\ f\ c \\
\qquad\ \text{-- appL f d -- not allowed due to linearity...}
\end{array}
$$

The type is inferred type (although, note, we had to do some explicit typing with the *Proxy*), as:

$$
\begin{array}{l}
\mathit{client5} :: \mathit{Channel}\ (\texttt{'Ch '}\ C) \\
\qquad\quad \to \mathit{Channel}\ (\texttt{'Ch '}\ X) \\
\qquad\quad \to \mathit{Session}\ `[\texttt{'Ch '}\ C :\to (\mathit{Ping} :!\ \mathit{End}), \texttt{'Ch '}\ X :\to (\mathit{Pong} :!\ \mathit{End})]\ ()
\end{array}
$$

We can then interact with this process in the expected way:

$$
\begin{aligned}
&\mathit{process5} = \mathit{new}\ (\lambda(c :: \mathit{Channel}\ (\mathit{Ch}\ C), c') \rightarrow\\
&\qquad\qquad \mathit{new}\ (\lambda(x :: \mathit{Channel}\ (\mathit{Ch}\ X), x') \rightarrow\\
&\qquad\qquad\quad (\mathit{client5}\ \ c\ x)\ \text{`}\mathit{par}\text{`}\\
&\qquad\qquad\qquad \mathbf{do}\ v \leftarrow \mathit{recv}\ c'\\
&\qquad\qquad\qquad\quad \mathit{print}\ v\\
&\qquad\qquad\qquad\quad v \leftarrow \mathit{recv}\ x'\\
&\qquad\qquad\qquad\quad \mathit{print}\ v))
\end{aligned}
$$

Where *run process5* prints `Ping` then `Pong`. Dom: TODO: should we also include a non-linear function application/abstraction for completeneess with HO?

Dimtris' example:

$$
\begin{aligned}
&\mathit{client6}\ (c :: (\mathit{Channel}\ (\mathit{Ch}\ C)))\ (d :: (\mathit{Channel}\ (\mathit{Ch}\ D))) =\\
&\quad \mathbf{do}\ f \leftarrow \mathit{absL}\ (\mathit{Proxy} :: \mathit{Proxy}\ \text{`}[(\mathit{Ch}\ C) :\!\rightarrow \mathit{Int} :!\ \mathit{End}])\\
&\qquad\qquad\qquad (\lambda(z :: (\mathit{Channel}\ (\mathit{Ch}\ Z))) \rightarrow (\mathit{send}\ z\ 42\ \text{`}\mathit{par}\text{`}\ \mathit{send}\ c\ 7))\\
&\qquad \mathit{appL}\ f\ d
\end{aligned}
$$

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

run process6 returns 49 as expected.

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:

data *Left*
data *Right*
data *Sup*

data *Label l* **where**
  LeftL :: *String* → *Label Left*
  RightL :: *String* → *Label Right*

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:

$$\textbf{data}\ \mathit{Sel}\ l\ s\ t$$
$$\textbf{data}\ s : \&\ t$$

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:

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

foo3 has the inferred type:

$$\begin{aligned}
&\mathit{foo3} :: \mathit{Channel}\ (\texttt{'Ch '}\ C) \\
&\quad \to \mathit{Session}\ `[\texttt{'Ch '}\ C :\to \mathit{Sel}\ \mathit{Left}\ (t :?\ (\mathit{Int} :!\ \mathit{End}))\ \mathit{End}]\ ()\ |
\end{aligned}$$

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:

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

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 $(\mathit{Del}\ c\ \mathit{s1}) \sim (\mathit{Del}\ c\ \mathit{s2})$). Finally, the returned session is that of $(\mathit{Del}\ c\ \mathit{s1})$ unioned with c mapping to the $(\mathit{Lookup}\ \mathit{s1}\ c) : \&\ (\mathit{Lookup}\ \mathit{s2}\ c)$, i.e., the branching pair of the session types for c in the left and right branches.

Here's an example:

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

Then *run process7* yields 42 as expected.

In order to take super types on selections, we define the following coercions that use the axiom **end** $<:$ S to introduce an arbtirary supertype for **end** on the left-hande side of a selection or right:

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

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

These are useful in the hotel example:

2 Hotel booking scenario

The P_{xy} process is encoded using two layers of abstraction to abstract over y and x (in that order). We use Haskell's implicit parameters feature to insert *room* and *credit* information, which are abstract here. P_{xy} is defined via p:

```
p :: (?room :: String, ?credit :: Int) ⇒
   Session ‘[] (Abs (Int :! (End : & End))
               (Abs (String :! (Int :? (Sel Sup (Int :! End) End))) ()))
p = absL Proxy (λ(y :: (Channel (Ch Y))) →
      absL Proxy (λ(x :: (Channel (Ch X))) →
        do send x (?room)
           quote ← recv x
           send y quote
           branch y (λ(LeftL "accept") → selSupL $ do select x (LeftL "accept")
                                                      send x (?credit))
                (λ(RightL "reject") → selSupR $ select x (RightL "reject"))))
```

[Don't let the question marks ? here confused, they are nothing to do with sending, this just marks them as 'dynamic'/implicit parameters for Haskell]

The client is then:

```
client (s1 :: (Channel (Ch Y))) (s2 :: (Channel (Ch Z))) =
   new (λ(h1 :: (Channel (Ch C)), h1′) →
   new (λ(h2 :: (Channel (Ch D)), h2′) →
     (do p0 ← p
         p1 ← p
         ph1 ← appL p0 h1   -- implements λx.P_{x,y}{h1/y}
         ph2 ← appL p1 h2   -- implements λx.P_{x,y}{h2/y}
         send s1 ph1
         send s2 ph2)
   ‘par‘ (do x ← recv h1′
             y ← recv h2′
             if (x ⩽ y) then do selSupL (select h1′ (LeftL "accept"))
```

$$
\begin{array}{ll}
 & \mathit{selSupR}\ (\mathit{select}\ h2'\ (\mathit{RightL}\ \texttt{"reject"})) \\
\textbf{else do} & \mathit{selSupR}\ (\mathit{select}\ h1'\ (\mathit{RightL}\ \texttt{"reject"})) \\
 & \mathit{selSupL}\ (\mathit{select}\ h2'\ (\mathit{LeftL}\ \texttt{"acccept"})))))
\end{array}
$$

Which has its type inferred as:

$$
\begin{array}{l}
\mathit{client} \\
\quad :: (?\mathit{credit} :: \mathit{Int}, ?\mathit{room} :: \mathit{String}) \Rightarrow \\
\qquad \mathit{Channel}\ (\texttt{'Ch '}\ Y) \\
\qquad \to \mathit{Channel}\ (\texttt{'Ch '}\ Z) \\
\qquad \to \mathit{Session} \\
\qquad\quad `[\texttt{'Ch '}\ Y \\
\qquad\qquad :\to (\mathit{Abs}\ (\mathit{String}\ :!\ (\mathit{Int}\ :?\ \mathit{Sel}\ \mathit{Sup}\ (\mathit{Int}\ :!\ \mathit{End})\ \mathit{End}))\ ()\ :!\ \mathit{End}), \\
\qquad\quad\ \ \texttt{'Ch '}\ Z \\
\qquad\qquad :\to (\mathit{Abs}\ (\mathit{String}\ :!\ (\mathit{Int}\ :?\ \mathit{Sel}\ \mathit{Sup}\ (\mathit{Int}\ :!\ \mathit{End})\ \mathit{End}))\ ()\ :!\ \mathit{End})] \\
\qquad\quad ()
\end{array}
$$