1 A brief 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:

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

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.

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

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:

$$\begin{aligned}&\quad\text{-- Session types}\\&\textbf{data}\ a :!\ s\\&\textbf{data}\ a :?\ s\\&\textbf{data}\ \mathit{End}\end{aligned}$$

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

$$\begin{aligned}&\textbf{type}\ \mathit{family}\ \mathit{Dual}\ s\ \textbf{where}\\&\quad \mathit{Dual}\ \mathit{End} = \mathit{End}\\&\quad \mathit{Dual}\ (t :!\ s) = t :?\ (\mathit{Dual}\ s)\\&\quad \mathit{Dual}\ (t :?\ s) = t :!\ (\mathit{Dual}\ s)\end{aligned}$$

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].

$$\begin{aligned}&\textbf{data}\ \mathit{ChanNameSymbol} = X \mid Y \mid Z \mid C \mid D\\&\textbf{data}\ \mathit{ChanName} = \mathit{Ch}\ \mathit{ChanNameSymbol} \mid \mathit{Op}\ \mathit{ChanNameSymbol}\end{aligned}$$

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]

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

1.1 π-calculus part

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

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

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

These both take a named channel $\mathit{Channel}\ c$ and return a *Session* computation indexed by the session environment $\text{`}[c :\to S]$ 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:

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

This function is of type:

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

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:

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

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:

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

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'

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

This has inferred type:

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

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.

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

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

1.2 λ-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*

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

The *Abs* data constructor takes a function of type $forall\ c \circ (Channel\ c \to Session\ (UnionS\ s\ `[c :\to t])\ ())$, that is, a function from *universally quanitifed* channel name c to a *Session* environment s where $c :\to t$ 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

$$\begin{array}{l} \mathit{appH} :: \mathit{Abs}\ t\ s \to \mathit{Channel}\ c \to \mathit{Session}\ (\mathit{UnionS}\ s\ `[c :\to t])\ () \\ \mathit{appH}\ (\mathit{Abs}\ \_\ k)\ c = k\ c \end{array}$$

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

$$\begin{array}{l} \mathit{client4}\ (c :: \mathit{Channel}\ (\mathit{Ch}\ C)) = \mathbf{do} \\ \quad \mathbf{let}\ f = \mathit{Abs}\ (\mathit{Proxy} :: (\mathit{Proxy}\ `[\,]))\ (\lambda c \to \mathit{send}\ c\ \mathit{Ping}) \\ \quad \mathit{appH}\ f\ c \end{array}$$

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.

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