Review of the $\pi$-calculus

$\pi$-calculus actions:

\[
\begin{align*}
\pi & ::= \pi(y) & \text{Send } y \text{ on channel } x \\
& | x(y) & \text{Receive } y \text{ on channel } x \\
& | \tau & \text{Silent action}
\end{align*}
\]

Process expressions:

\[
\begin{align*}
P & ::= \pi.P & \text{Take action } \pi, \text{ continue with } P \\
& | 0 & \text{Finished} \\
& | P_1 + \cdots + P_n & \text{Alternation} \\
& | (P_1 \mid P_2) & \text{Parallel} \\
& | \text{new } a \ P & \text{Binding} \\
& | !P & \text{Replication}
\end{align*}
\]

Recall the structural equivalence $!P \equiv P \mid !P$.

Warm-up: Encoding booleans

Suppose we had to program in MinML without a boolean type (in fact, without any base types), and therefore without an if-then-else construct. Can we encode booleans using only functions? We need to encode the type $\text{bool}$, the constructors $\text{true}$ and $\text{false}$, and the construct $\text{if}(e, e_1, e_2)$.

The standard encoding is

\[
\begin{align*}
\text{bool} &= \forall t. t \to t \to t \\
\text{true} &= \lambda t. \lambda f. t \\
\text{false} &= \lambda t. \lambda f. f \\
\text{if}(e, e_1, e_2) &= e \ e_1 \ e_2
\end{align*}
\]

It’s not too far a leap from the above to an encoding in the $\pi$-calculus. Just as $\lambda t. \lambda f. t$ selects its first argument, we can write $\ell(t, f). t$ to select (write to)
the first component of the pair \((t, f)\) written to the channel \(\ell\).

\[
\begin{align*}
\text{True}(\ell) & = \ell(t, f).\overline{1} \\
\text{False}(\ell) & = \ell(t, f).\overline{0} \\
\text{If}(\ell, P, Q) & = \text{new } (t, f) \ (\overline{1}(t, f) \mid (t.P + f.Q))
\end{align*}
\]

Example:

\[
\begin{align*}
\text{If}(\ell, P, Q) \mid \text{True}(\ell) & = \left(\text{new } (t, f) \ (\overline{1}(t, f) \mid (t.P + f.Q))\right) \mid \ell(t, f).\overline{1} \\
& \rightarrow^* (t.P + f.Q) \mid \overline{1} \\
& \rightarrow^* P
\end{align*}
\]

**Encoding the natural numbers**

Just as \texttt{bool} is essentially the datatype

\[
\text{datatype bool =}
\begin{align*}
& \text{true} \\
& \mid \text{false}
\end{align*}
\]

the natural numbers are essentially the datatype

\[
\text{datatype nat =}
\begin{align*}
& Z \quad \text{(* zero *)} \\
& \mid S \text{ of nat} \quad \text{(* successor *)}
\end{align*}
\]

Like \texttt{bool}, \texttt{nat} has two constructors, so a process that “is” a natural number will read two values—the first telling it what to do if it is zero, the second what to do if it is the successor of something. The \texttt{Z} constructor, like the constructors of \texttt{bool}, takes no arguments. Thus, its encoding is analogous to the encoding of \texttt{true} and \texttt{false}.

\[
\begin{align*}
Z(\ell) & = \ell(z, s).\overline{3} \\
S(\ell, n) & = \ell(z, s).\overline{s}(n)
\end{align*}
\]

On the other hand, the constructor \texttt{S} is not nullary. So instead of transmitting nothing along the channel \(s\), it transmits \(n\), which is (a channel to) the number it is the successor of.

**Example.** Suppose we have the following processes. Note that \texttt{zero} and \texttt{one} are channel names; a natural number is manipulated by sending a \texttt{z} and an \texttt{s} to one of these channels.

\[
\begin{align*}
& Z(\text{\texttt{zero}}) \\
& \mid S(\text{\texttt{one}}, \text{\texttt{zero}}) \\
& \mid \overline{\text{\texttt{and}}}(p, q) \\
& \mid p().\text{print } "." \\
& \mid q(n).\text{(print } "*": \overline{n}(p, q))
\end{align*}
\]
By the definitions above, this is equivalent to
\[
\begin{align*}
\text{zero}(z, s) &. \overline{z} \\
\text{one}(z, s) &. \pi(\text{zero}) \\
\text{one}(p, q) & \\
p(.). \text{print } "." \\
q(n). (\text{print } "\ast"; \pi(p, q)) 
\end{align*}
\]

It’s quite easy to run this set of processes by hand; the result should be that 
\ast. is printed.

What happens if we also have \(S(\text{two}, \text{one})\) and do \(\text{two}(p, q)\)? We might expect the output "\ast\ast.". However, the process receiving along \(q\) is “used up” the first time it’s run, so we will deadlock trying to send to a channel \(q\) that has no receiver! The solution is to use replication:
\[
!q(n). (\text{print } "\ast"; \pi(p, q))
\]

Now, by the rules of structural equivalence, we can make as many copies of 
\(q(n). (\text{print } "\ast"; \pi(p, q))\) as we need.

Observe that a similar phenomenon arises if we try to use a number more than once. In the example above, as soon as we send to \text{one}, that process steps to 0 (strictly speaking we should have written \(\text{one}(z, s). \pi(\text{zero}). 0\)), so by the rules of structural equivalence, it vanishes into thin air. Again the solution is simply to put a ! before any “object” we might wish to use more than once.

As an interesting example of this, SML does not let you declare a number \(n\) to be \(S(n)\) (the successor of itself), but we can easily do so in the π-calculus:
\[
!S(\text{inf}, \text{inf}) = !\text{inf}(z, s). \pi(\text{inf})
\]

If we send \(p\) and \(q\) (as above) to \text{inf}, we will forever print asterisks.

**Given in second recitation but omitted here:**  \(\text{succ}\) and \(\text{add}\) (untested).