Review of the $\pi$-calculus

$\pi$-calculus actions:

$$\pi ::= \pi(y) \quad \text{Send } y \text{ on channel } x$$

$$\mid x(y) \quad \text{Receive } y \text{ on channel } x$$

$$\mid \tau \quad \text{Silent action}$$

Process expressions:

$$P ::= \pi.P \quad \text{Take action } \pi, \text{ continue with } P$$

$$\mid 0 \quad \text{Finished}$$

$$\mid P_1 + \cdots + P_n \quad \text{Alternation}$$

$$\mid (P_1 | P_2) \quad \text{Parallel}$$

$$\mid \text{new } a \ P \quad \text{Binding}$$

$$\mid !P \quad \text{Replication}$$

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

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

$$\text{bool} = \forall t. t \to t \to t$$

$$\text{true} = \lambda t. \lambda f. t$$

$$\text{false} = \lambda t. \lambda f. f$$

$$\text{if}(e,e_1,e_2) = e \; e_1 \; e_2$$

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 $\ell(t,f).\overline{t}$ to select (write to)
the first component of the pair \((t, f)\) written to the channel \(\ell\).

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

Example:

\[
If(\ell, P, Q) \mid True(\ell) = (\text{new} \ (t, f) \ (\overline{I}(t, f) \mid (t.P + f.Q))) \mid \ell(t, f).\overline{I} \\
\rightarrow^* \ (t.P + f.Q) \mid \overline{I} \\
\rightarrow^* \ P
\]

### Encoding the natural numbers

Just as \texttt{bool} is essentially the datatype

\[
\texttt{datatype bool =} \\
\quad \texttt{true} \\
\quad \texttt{| false}
\]

the natural numbers are essentially the datatype

\[
\texttt{datatype nat =} \\
\quad \texttt{Z} \quad \texttt{(* zero *)} \\
\quad \texttt{| S of nat} \quad \texttt{(* successor *)}
\]

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}.

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

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 \(z\) and an \(s\) to one of these channels.

\[
\begin{align*}
Z(\texttt{zero}) \\
| S(\texttt{one, zero}) \\
| \overline{\texttt{print}(p, q)} \\
| p(). \texttt{print } \texttt{"."} \\
| q(n). (\texttt{print } \texttt{"*"}; \overline{\texttt{n시설}}(p, q))
\end{align*}
\]
By the definitions above, this is equivalent to

\[
\begin{align*}
zero(z, s). \overline{z} \\
\mid one(z, s). \overline{\pi(zero)} \\
\mid one(p, q) \\
\mid p(). \texttt{print } "." \\
\mid q(n). (\texttt{print } "*"; \overline{\pi(p, q)})
\end{align*}
\]

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 \(S(two, one)\) and do \(two⟨p, q⟩\)? We might expect the output \("**\)." 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:

\[
!q(n). (\texttt{print } "*"; \overline{\pi(p, q)})
\]

Now, by the rules of structural equivalence, we can make as many copies of \(q(n). (\texttt{print } "*"; \overline{\pi(p, q)})\) 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 \(one\), that process steps to 0 (strictly speaking we should have written \(one(z, s). \overline{\pi(zero)}. 0\), 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 \(\pi\)-calculus:

\[
! S(inf, inf) = ! inf(z, s). \overline{\pi(inf)}
\]

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

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