15-312 Recitation Notes #13

Joshua Dunfield
Carnegie Mellon University

November 20, 2002

Review of the π-calculus

π-calculus actions:

$$\begin{array}{rcll} \pi & ::= & \overline{x}\langle y \rangle & \text{Send } y \text{ on channel } x \\ & | & x(y) & \text{Receive } y \text{ on channel } x \\ & | & \tau & \text{Silent action} \end{array}$$

Process expressions:

$$\begin{array}{rcll} P & ::= & \pi.P & \text{Take action } \pi\text{, continue with } P \\ & | & 0 & \text{Finished} \\ & | & P_1 + \cdots + P_n & \text{Alternation} \\ & | & (P_1 \mid P_2) & \text{Parallel} \\ & | & \mathsf{new}\; a\; P & \text{Binding} \\ & | & !P & \text{Replication} \end{array}$$

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 `bool`, the constructors `true` and `false`, and the construct `if(e,e1,e2)`.

The standard encoding is

$$\begin{aligned} \mathsf{bool} &= \forall t.t \to t \to t \\ \mathtt{true} &= \lambda t.\lambda f.t \\ \mathtt{false} &= \lambda t.\lambda f.f \\ \mathtt{if}(e, e_1, e_2) &= e\; e_1\; e_2 \end{aligned}$$

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

$$\begin{aligned} \mathit{True}(\ell) &= \ell(t, f).\overline{t} \\ \mathit{False}(\ell) &= \ell(t, f).\overline{f} \\ \mathit{If}(\ell, P, Q) &= \mathsf{new}\ (t, f)\ \big(\overline{\ell}\langle t, f\rangle \mid (t.P + f.Q)\big) \end{aligned}$$

Example:

$$\begin{aligned} \mathit{If}(\ell, P, Q) \mid \mathit{True}(\ell) &= \big(\mathsf{new}\ (t, f)\ (\overline{\ell}\langle t, f\rangle \mid (t.P + f.Q))\big) \mid \ell(t, f).\overline{t} \\ &\to^* \big(t.P + f.Q\big) \mid \overline{t} \\ &\to^* P \end{aligned}$$

Encoding the natural numbers

Just as `bool` is essentially the datatype

```
datatype bool =
    true
  | false
```

the natural numbers are essentially the datatype

```
datatype nat =
    Z              (* zero *)
  | S of nat       (* successor *)
```

Like `bool`, `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 `Z` constructor, like the constructors of `bool`, takes no arguments. Thus, its encoding is analogous to the encoding of `true` and `false`.

$$\begin{aligned} Z(\ell) &= \ell(z, s).\,\overline{z} \\ S(\ell, n) &= \ell(z, s).\,\overline{s}\langle n\rangle \end{aligned}$$

On the other hand, the constructor `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 *zero* and *one* are channel names; a natural number is manipulated by sending a z and an s to one of these channels.

$$\begin{aligned} &\ Z(\mathit{zero}) \\ &\mid S(\mathit{one}, \mathit{zero}) \\ &\mid \overline{\mathit{one}}\langle p, q\rangle \\ &\mid p().\,\texttt{print "."} \\ &\mid q(n).\,(\texttt{print "*";}\ \overline{n}\langle p, q\rangle) \end{aligned}$$

By the definitions above, this is equivalent to

$$\begin{aligned} & zero(z,s).\,\overline{z} \\ \mid\ & one(z,s).\,\overline{s}\langle zero\rangle \\ \mid\ & \overline{one}\langle p,q\rangle \\ \mid\ & p().\,\texttt{print "."} \\ \mid\ & q(n).\,(\texttt{print "*";}\ \overline{n}\langle p,q\rangle) \end{aligned}$$

It's quite easy to run this set of processes by hand; the result should be that `*.` is printed.

What happens if we also have $S(two, one)$ and do $\overline{two}\langle p, q\rangle$? 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:

$$!\,q(n).\,(\texttt{print "*";}\ \overline{n}\langle p,q\rangle)$$

Now, by the rules of structural equivalence, we can make as many copies of $q(n).\,(\texttt{print "*";}\ \overline{n}\langle p,q\rangle)$ 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 $one(z,s).\,\overline{s}\langle zero\rangle.\,0$), 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 $\texttt{S}(n)$ (the successor of itself), but we can easily do so in the π-calculus:

$$!\,S(inf,\ inf) \quad = \quad !\,inf(z,s).\,\overline{s}\langle inf\rangle$$

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

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