1 The \( \pi \)-calculus

\[
\pi \ ::= \ \tau \mid x(y) \mid [x = y] \pi \\
M, N \ ::= \ 0 \mid \pi.P \mid M + N \\
P \ ::= \ M \mid P_1 | P_2 \mid \nu x.P \ | !P
\]

Polyadic encoding and recursion were also covered in the previous lecture.

2 Encoding Booleans

\[
\text{True}(l) \triangleq l(t,f).\bar{t} \\
\text{False}(l) \triangleq l(t,f).\bar{f} \\
\text{Cond}(P,Q)(l) \triangleq \nu t, f(l(t,f).(t.P + f.Q))
\]

3 Encoding Lists

\[
\text{Nil}(k) \triangleq k(n,c).\bar{k} \\
\text{Cons}(V,L)(k) \triangleq \nu v, l(k(n,c).\bar{c}(v,l) \mid V(v) \mid L(l)) \\
\text{IsNil}(L)(r) \triangleq \nu l, n, c(L(l) \mid \bar{l}(n,v).(n.\text{True}(r) + c(n,t).\text{False}(r)))
\]

Introduce a case form:

\[
\text{case } l \text{ of } \\
\text{Nil}? \Rightarrow P \\
\text{Cons}? \Rightarrow Q
\]

3.1 Destructive List Copy

\[
\text{Copy}(l, m) \triangleq \text{case } l \text{ of } \\
\text{Nil}? \Rightarrow \text{Nil}(m) \\
\text{Cons?(h,t)} \Rightarrow \nu m'(m(n,c).\bar{c}(n,m') \mid \text{Copy}(r,m'))
\]
3.2 Destructive List Join

\[
\text{Join}(k, l, m) \triangleq \text{case } k \text{ of }
\]
\[
\quad \text{Nil? } \Rightarrow \text{Copy}(l, m)
\]
\[
\quad \text{Cons?(h, t) } \Rightarrow \nu m'(m(n, c).c\langle n, m' \rangle \mid \text{Join}(t, l, m'))
\]

A \textit{Double} operation cannot be defined as follows because \text{Join} is destructive

\[
\text{Double}(l, m) \triangleq \text{Join}(l, m)
\]

4 Encoding Persistent Datatypes

We put a ! in front of processes to turn them into servers create arbitrary numbers of the original process. This prevents their destruction after sending or receiving a message. This is generally used on “connectors” – the processes that acts as “cons cell” equivalents. The List encoding can be rewritten as:

\[
\text{Nil}(k) \triangleq \nu k(n, c).\tilde{k}
\]
\[
\text{Cons}(V, L)(k) \triangleq \nu v, l(!k(n, c).c\langle v, l \rangle \mid V\langle v \rangle \mid L\langle l \rangle)
\]
\[
\text{IsNil}(L)(r) \triangleq \nu l, n, c(L\langle l \rangle \mid l\langle n, v \rangle \mid n.\text{True}(r) + c(n, t).\text{False}(r))
\]

5 Reference Cells for Mutable Data

\[
\text{NullRef}(r) \triangleq r(g, s, t, i).(s(v).Ref\langle v, r \rangle + t.\text{NullRef}\langle r \rangle + i(b).\text{True}(b))
\]
\[
\text{Ref}(v, r) \triangleq r(g, s, t, i).\tilde{g}(v) + s(v').Ref\langle v', r \rangle + t.\text{NullRef}\langle r \rangle + i(b).\text{False}(b))
\]

6 Encoding the \(\lambda\)-calculus

We can express the syntax of the pure untyped \(\lambda\)-calculus as:

\[
e ::= x \mid e_1 e_2 \mid \lambda x. e
\]

The semantics can be expressed as:

\[
\frac{e_1 \rightarrow e'_1}{e_1 e_2 \rightarrow e'_1 e_2} \quad \frac{(\lambda x. e_1)e_2 \rightarrow e_1[e_2/x]}{}
\]

The above semantics can be encoded in the \(\pi\)-calculus as:

\[
\overline{x}(u) \triangleq \bar{x}\langle u \rangle
\]
\[
\overline{\lambda x. e}(u) \triangleq u(x, y).\overline{e}\langle y \rangle
\]
\[
\overline{e_1, e_2}(u) \triangleq v y(\overline{e_1}\langle y \rangle \mid v x(\tilde{g}(x, u) \mid !x(w).\overline{e_2}\langle w \rangle))
\]