1 Introduction

A fundamental limitation of CCS is that the communication structure of a process is fixed. For example, it is easy to show that the set \( \{ \alpha \mid P \equiv P' \} \) is finite. The \( \pi \)-calculus is a similar, but more expressive calculus that addresses this deficiency.

2 Syntax

\[
P ::= 0 \quad \text{Inert}
\]

\[
| \quad x(y).P \quad \text{Receive}
\]

\[
| \quad \pi(y).P \quad \text{Send}
\]

\[
| \quad P_1 | P_2 \quad \text{Parallel composition}
\]

\[
| \quad \nu x. \ P \quad \text{Restriction}
\]

\[
| \quad !P \quad \text{Replication}
\]

Compared to CCS, note that instead of simply interacting on a named channel, we can now communicate channel names! In addition, \( \pi \)-calculus does not have summation or top-level definitions. Again, we work up to \( \alpha \) equivalence for restrictions.

3 Labeled Transition System

Structural congruence is defined as follows:

\[
P|Q \equiv Q|P
\]

\[
(P|Q)|R \equiv P|(Q|R)P|0 \equiv P
\]

\[
\nu x. \ 0 \equiv 0
\]

\[
\nu x. \ (P|Q) \equiv (\nu x. P)|Q \text{ if } x \notin \text{fv}(Q)
\]

\[
!P \equiv !P|P
\]

Reduction is defined as follows:

\[
\pi(y). \ P|x(z).Q \rightarrow P|Q\{y/z\}
\]

\[
P \rightarrow P'
\]

\[
P|Q \rightarrow P'|Q
\]

\[
P \rightarrow P'
\]

\[
\nu x. \ P \rightarrow \nu x. \ P'
\]

\[
P \equiv Q \quad Q \rightarrow Q' \quad Q' \equiv P'
\]

\[
P' \rightarrow P'
\]

The definition of the labeled transition system is left as an exercise.
4 Programming in the \(\pi\)-calculus

The rest of this lecture will explore how we can implement various programming constructs in \(\pi\)-calculus.

4.1 Polyadic Communication

Although \(\pi\)-calculus send/receive are unary, we can encode polyadic communication as follows:

\[
\begin{align*}
\l(x_1, \ldots, x_n) & \triangleq \nu p. \l(p). \l(\overline{x_1}) \ldots \overline{x_n} \\
\l(y_1, \ldots, y_n) & \triangleq \l(p). \l(p(y_1)) \ldots \l(p(y_n))
\end{align*}
\]

Intuitively, this encoding works by first creating a fresh channel name \(p\), sending \(p\) along \(\l\), and then sending the actual names \(x_1, \ldots, x_n\) along \(p\). The use of a fresh channel ensures that multiple senders and receivers will not interfere with each other.

4.2 Booleans

We can encode booleans as processes that receive names of \(t\) and \(f\) channels, and then send on the corresponding channel.

\[
\begin{align*}
\text{True}(b) & \triangleq \!b(t, f).\overline{t} \\
\text{False}(b) & \triangleq \!b(t, f).\overline{f} \\
\text{Cond}(P; Q)(b) & \triangleq \nu t. f(\overline{b}(t, f). (t(). P + f(). Q))
\end{align*}
\]

Note that 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.

4.3 Internal Choice

Although \(\pi\)-calculus does not have summation, we can encode a limited form of “internal” choice:

\[
P \oplus Q \triangleq \nu c. (\overline{c}(). P \mid c(). Q)
\]

4.4 References

We can also encode mutable references as processes.

\[
\begin{align*}
\text{Ref}(r, w, i) & \triangleq \nu l. \l(i) \mid \text{Read}(l, r) \mid \text{Write}(l, r) \\
\text{Read}(l, r) & \triangleq \nu l. !r(c). \l(v). (\overline{c}(). \overline{l}(v)) \\
\text{Write}(l, w) & \triangleq \nu l. !w(c, v'). \l(v). (\overline{c}(). \overline{l}(v'))
\end{align*}
\]
Finally, we can encode the $\lambda$-calculus into the $\pi$-calculus as follows:

\[
\begin{align*}
\llbracket x \rrbracket (p) & \triangleq \pi(p) \\
\llbracket \lambda x. e \rrbracket (p) & \triangleq p(x, q).\llbracket e \rrbracket (q) \\
\llbracket e_1, e_2 \rrbracket (p) & \triangleq \nu q(\llbracket e_1 \rrbracket (q) | \nu y(\pi y, p) | ! y(r), \llbracket e_2 \rrbracket (r))
\end{align*}
\]

Note the similarity to continuation-passing style.