CS 4110

Programming Languages & Logics

Lecture 36
Concurrency

Concurrency

All of the languages we have seen so far in this course have been sequential, performing one step of computation at a time.

In the next few lectures we will consider languages where multiple threads of execution may be interleaved simultaneously.

These languages can be used to model computations that execute on parallel and distributed architectures.

Process Calculi

In the 1970s, Tony Hoare, Robin Milner, and others (correctly) observed that in the future, computers with shared-nothing architectures communicating by sending messages to each other would be important.

Hoare's Communicating Sequential Processes were an early and highly-influential language that capture a *message passing* form of concurrency.

Many languages have built on CSP including Milner's CCS and π-calculus, Petri nets, and others. We're going to look at π-calculus, which is a minimal core calculus in the style of the λ-calculus.

π-calculus Syntax

The π-calculus is a minimal formalism that models capture the "essence" of concurrency based on message-passing:

π-calculus Syntax

The π-calculus is a minimal formalism that models capture the “essence” of concurrency based on message-passing:

The key constructs are based on the ability to interact by sending and receiving channel names:

π-calculus Syntax

The π-calculus is a minimal formalism that models capture the "essence" of concurrency based on message-passing:

The key constructs are based on the ability to interact by sending and receiving channel names:

$$x, y, z \quad \in \quad \mathcal{N} \qquad\qquad \textit{Names}$$

π-calculus Syntax

The π-calculus is a minimal formalism that models capture the “essence” of concurrency based on message-passing:

The key constructs are based on the ability to interact by sending and receiving channel names:

$$
\begin{array}{rcll}
x, y, z & \in & \mathcal{N} & \textit{Names} \\
\pi & ::= & \tau \mid \bar{x}\langle y\rangle \mid x(y) \mid [x = y]\,\pi & \textit{Prefixes}
\end{array}
$$

π-calculus Syntax

The π-calculus is a minimal formalism that models capture the "essence" of concurrency based on message-passing:

The key constructs are based on the ability to interact by sending and receiving channel names:

$$
\begin{array}{rcll}
x, y, z & \in & \mathcal{N} & \textit{Names} \\
\pi & ::= & \tau \mid \bar{x}\langle y \rangle \mid x(y) \mid [x = y]\,\pi & \textit{Prefixes} \\
M, N & ::= & \mathbf{0} \mid \pi.P \mid M + M & \textit{Summations}
\end{array}
$$

π-calculus Syntax

The π-calculus is a minimal formalism that models capture the “essence” of concurrency based on message-passing:

The key constructs are based on the ability to interact by sending and receiving channel names:

$$
\begin{array}{rcll}
x, y, z & \in & \mathcal{N} & \textit{Names} \\
\pi & ::= & \tau \mid \bar{x}\langle y \rangle \mid x(y) \mid [x = y]\,\pi & \textit{Prefixes} \\
M, N & ::= & \mathbf{0} \mid \pi.P \mid M + M & \textit{Summations} \\
P, Q, R & ::= & M \mid P_1 \mid P_2 \mid \nu x.\ P \mid !P & \textit{Processes}
\end{array}
$$

π-calculus Syntax

The π-calculus is a minimal formalism that models capture the "essence" of concurrency based on message-passing:

The key constructs are based on the ability to interact by sending and receiving channel names:

$$\begin{array}{rcll} x, y, z & \in & \mathcal{N} & \textit{Names} \\ \pi & ::= & \tau \mid \bar{x}\langle y\rangle \mid x(y) \mid [x = y]\,\pi & \textit{Prefixes} \\ M, N & ::= & \mathbf{0} \mid \pi.P \mid M + M & \textit{Summations} \\ P, Q, R & ::= & M \mid P_1 \mid P_2 \mid \nu x.\ P \mid\ !P & \textit{Processes} \end{array}$$

In examples, we will often appreviate $\pi.0$ as π

Examples

$$a(x).\bar{b}\langle x\rangle \mid \nu z.\,(\bar{a}\langle z\rangle)$$

Examples

$$a(x) + b(x) \mid \nu z.\, (\bar{a}\langle z \rangle + \bar{b}\langle z \rangle)$$

Reaction

$$\frac{}{\tau.P + M \to P}\ \textsc{R-Tau}$$

Reaction

$$\frac{}{\tau.P + M \to P}\ \textsc{R-Tau}$$

$$\frac{}{(\overline{x}\langle y\rangle.P_1 + M_1) \mid (x(z).P_2 + M_2) \to P_1 \mid P_2\{y/z\}}\ \textsc{R-React}$$

Reaction

$$\frac{}{\tau.P + M \to P}\ \textsc{R-Tau}$$

$$\frac{}{(\bar{x}\langle y\rangle.P_1 + M_1) \mid (x(z).P_2 + M_2) \to P_1 \mid P_2\{y/z\}}\ \textsc{R-React}$$

$$\frac{P_1 \to P_1'}{P_1 \mid P_2 \to P_1' \mid P_2}\ \textsc{R-Par}$$

Reaction

$$\frac{}{\tau.P + M \to P}\ \textsc{R-Tau}$$

$$\frac{}{(\bar{x}\langle y\rangle.P_1 + M_1) \mid (x(z).P_2 + M_2) \to P_1 \mid P_2\{y/z\}}\ \textsc{R-React}$$

$$\frac{P_1 \to P_1'}{P_1 \mid P_2 \to P_1' \mid P_2}\ \textsc{R-Par}$$

$$\frac{P \to P'}{\nu x.\ P \to \nu x.\ P'}\ \textsc{R-Res}$$

Reaction

$$\frac{}{\tau.P + M \to P}\ \textsc{R-Tau}$$

$$\frac{}{(\overline{x}\langle y\rangle.P_1 + M_1) \mid (x(z).P_2 + M_2) \to P_1 \mid P_2\{y/z\}}\ \textsc{R-React}$$

$$\frac{P_1 \to P_1'}{P_1 \mid P_2 \to P_1' \mid P_2}\ \textsc{R-Par}$$

$$\frac{P \to P'}{\nu x.\ P \to \nu x.\ P'}\ \textsc{R-Res}$$

$$\frac{P \equiv P' \qquad P' \to Q' \qquad Q' \equiv Q}{P \to Q}\ \textsc{R-Struct}$$

Structural Congruence

Definition (Structural Congruence)

$$[x = x]\,\pi.P \equiv \pi.P \qquad !P \equiv P \mid !P$$

$$M_1 + (M_2 + M_3) \equiv (M_1 + M_2) + M_3 \qquad M_1 + M_2 \equiv M_2 + M_1$$

$$P_1 \mid (P_2 \mid P_3) \equiv (P_1 \mid P_2) \mid P_3 \qquad P_1 \mid P_2 \equiv P_2 \mid P_1$$

$$M + \mathbf{0} \equiv M \qquad P \mid \mathbf{0} \equiv P$$

$$\nu x.\ \nu y.\ P \equiv \nu y.\ \nu x.\ P \qquad \nu x.\ \mathbf{0} \equiv \mathbf{0}$$

$$\nu x.\ P_1 \mid P_2 \equiv P_1 \mid (\nu x.\ P_2),\ \text{if } x \notin \mathrm{FV}(P_1)$$

Structural Congruence

Theorem (Standard Form)

Each process is structurally congruent to one of the form

$$\nu\vec{x}.\ (M_1 \mid \ldots \mid M_j \mid !P_1 \mid \ldots \mid !P_k)$$

where each P_i is also in standard form.

Structural Congruence

Theorem (Standard Form)

Each process is structurally congruent to one of the form

$$\nu\vec{x}.\ (M_1 \mid \ldots \mid M_j \mid !P_1 \mid \ldots \mid !P_k)$$

where each P_i is also in standard form.

Proof (sketch): repeatedly use α-conversion and scope extrusion: $P \mid \nu x.\ Q \equiv \nu x.\ P \mid Q$.

Programming in the π-calculus

Just as with λ-calculus, we can encode richer data structures and computations using the π-calculus primitives.

Polyadic π-Calculus

The send and receive primitives are monadic—they communicate a single name over a given channel. It is often useful to be able to send several names.

Polyadic π-Calculus

The send and receive primitives are monadic—they communicate a single name over a given channel. It is often useful to be able to send several names.

We can try to encode polyadic sends and receives as follows:

$$\begin{aligned} \overline{x}\langle y_1, \ldots, y_k \rangle.P &\triangleq \overline{x}\langle y_1 \rangle. \ldots .\overline{x}\langle y_k \rangle.P \\ x(z_1, \ldots, z_k).P &\triangleq x(z_1). \ldots .\overline{x}\langle z_k \rangle.P \end{aligned}$$

Polyadic π-Calculus

The send and receive primitives are monadic—they communicate a single name over a given channel. It is often useful to be able to send several names.

We can try to encode polyadic sends and receives as follows:

$$\begin{aligned} \overline{x}\langle y_1, \ldots, y_k \rangle.P &\triangleq \overline{x}\langle y_1 \rangle. \ldots .\overline{x}\langle y_k \rangle.P \\ x(z_1, \ldots, z_k).P &\triangleq x(z_1). \ldots .\overline{x}\langle z_k \rangle.P \end{aligned}$$

But unfortunately this doesn't work... why?

Polyadic π-calculus

To obtain an encoding that works correctly, we can create a fresh name and communicate the values over that channel:

$$\overline{x}\langle y_1, \ldots, y_k \rangle.P \triangleq \nu w.\ (\overline{x}\langle w \rangle.\overline{w}\langle y_1 \rangle.\ldots.\overline{w}\langle y_k \rangle).P$$

where $w \notin \mathrm{FV}(P)$

$$x(z_1, \ldots, z_k).P \triangleq x(w).w(z_1).\ldots.\overline{w}\langle z_k \rangle.P$$

Polyadic π-calculus

To obtain an encoding that works correctly, we can create a fresh name and communicate the values over that channel:

$$\begin{array}{rcl} \overline{x}\langle y_1, \ldots, y_k\rangle.P & \triangleq & \nu w.\ (\overline{x}\langle w\rangle.\overline{w}\langle y_1\rangle.\ldots.\overline{w}\langle y_k\rangle).P \\ \multicolumn{3}{l}{\text{where } w \notin \mathrm{FV}(P)} \\ x(z_1, \ldots, z_k).P & \triangleq & x(w).w(z_1).\ldots.\overline{w}\langle z_k\rangle.P \end{array}$$

Using this (adequate) encoding, we will freely use polyadic sends and receives in examples.

$$\frac{}{(\overline{\vec{x}}\langle\vec{y}\rangle.P_1 + M_1) \mid (\vec{x}(\vec{z}).P_2 + M_2) \rightarrow P_1 \mid P_2\{\vec{y}/\vec{z}\}}\ \textsc{R-PolyReact}$$

We'll use the notation $\overline{x}.P$ and $x.P$ for 0-ary sends and receives.

Encoding Recursive Definitions

Idea: Suppose we want to support recursive definitions.

We'll write $A(\vec{x}) \triangleq P_A$ for the definition of A, and $A\langle\vec{y}\rangle$ for an instantiation of A with $\vec{y}$.

- Pick a fresh name a to stand for A.
- Let $(\!|Q|\!)$ stand for Q with occurrences of $A\langle\vec{z}\rangle$ replaced by $\overline{a}\langle\vec{z}\rangle$.
- Produce $\nu a.\ \left((\!|P_A|\!) \mid\ !a(\vec{x}).(\!|P_A|\!)\right)$

Example: Encoding Booleans

Idea: encode a boolean value *b* as a process that receives two channels *t* and *f* on the channel *l* where the boolean is “located” and then signals on the corresponding channel

$$True(l) \quad \triangleq \quad l(t, f).\overline{t}$$

Example: Encoding Booleans

Idea: encode a boolean value b as a process that receives two channels t and f on the channel l where the boolean is "located" and then signals on the corresponding channel

$$\begin{array}{rcl} True(l) & \triangleq & l(t, f).\overline{t} \\ False(l) & \triangleq & l(t, f).\overline{f} \end{array}$$

Example: Encoding Booleans

Idea: encode a boolean value *b* as a process that receives two channels *t* and *f* on the channel *l* where the boolean is “located” and then signals on the corresponding channel

$$\begin{array}{rcl} \mathit{True}(l) & \triangleq & l(t,f).\overline{t} \\ \mathit{False}(l) & \triangleq & l(t,f).\overline{f} \\ \mathit{Cond}(P,Q)(l) & \triangleq & \nu t,f.\ (\overline{l}\langle t,f\rangle.(t.P + f.Q)) \end{array}$$

Example: Encoding Naturals

Idea: encode a natural number value n as a process that receives two channels s and z on the channel c where the number is "located" and then signals on s n times terminated by z

Example: Encoding Naturals

Idea: encode a natural number value *n* as a process that receives two channels *s* and *z* on the channel *c* where the number is “located” and then signals on *s* *n* times terminated by *z*

$$Zero(c) \quad \triangleq \quad c(s, z).\bar{z}$$

Example: Encoding Naturals

Idea: encode a natural number value n as a process that receives two channels s and z on the channel c where the number is "located" and then signals on s n times terminated by z

$$\begin{array}{rcl} Zero(c) & \triangleq & c(s,z).\bar{z} \\ Succ(n)(c) & \triangleq & c(s,z).\,\bar{s}.\,\bar{n}\langle s,z\rangle \end{array}$$

Encoding Lists

Idea: encode a list l as a process that receives two channels c and n on the channel l where the list is “located” and then signals on c with each value of the list, terminated by n

Encoding Lists

Idea: encode a list l as a process that receives two channels c and n on the channel l where the list is "located" and then signals on c with each value of the list, terminated by n

$$Nil(l) \triangleq l(n, c).\overline{n}$$

Encoding Lists

Idea: encode a list l as a process that receives two channels c and n on the channel l where the list is “located” and then signals on c with each value of the list, terminated by n

$$
\begin{aligned}
Nil(l) &\triangleq l(n, c).\bar{n} \\
Cons(H, T)(l) &\triangleq \nu h, t.\ (l(n, c).\bar{c}\langle h, t\rangle \mid H\langle h\rangle \mid T\langle t\rangle)
\end{aligned}
$$

Encoding Lists

Idea: encode a list l as a process that receives two channels c and n on the channel l where the list is “located” and then signals on c with each value of the list, terminated by n

$$
\begin{array}{rcl}
\mathit{Nil}(l) & \triangleq & l(n, c).\bar{n} \\
\mathit{Cons}(H, T)(l) & \triangleq & \nu h, t.\ (l(n, c).\bar{c}\langle h, t\rangle \mid H\langle h\rangle \mid T\langle t\rangle) \\
\mathit{IsNil}(L)(r) & \triangleq & \nu l, n, c.\ (L\langle l\rangle \mid \bar{l}\langle n, c\rangle.(n.\mathit{True}\langle r\rangle + c(h, t).\mathit{False}\langle r\rangle))
\end{array}
$$

Pattern Matching

We can encode pattern matching on lists

$$
\begin{array}{l}
\text{case } l \text{ of} \\
\quad Nil? \Rightarrow P \\
\quad Cons?(h, t) \Rightarrow Q
\end{array}
$$

Pattern Matching

We can encode pattern matching on lists

$$\begin{array}{l} \text{case } l \text{ of} \\ \quad \mathit{Nil}? \Rightarrow P \\ \quad \mathit{Cons}?(h, t) \Rightarrow Q \end{array}$$

Idea: send fresh channels n and c to l and test which it signals on:

Pattern Matching

We can encode pattern matching on lists

$$
\begin{array}{l}
\text{case } l \text{ of} \\
\quad Nil? \Rightarrow P \\
\quad Cons?(h, t) \Rightarrow Q
\end{array}
$$

Idea: send fresh channels n and c to l and test which it signals on:

$$\nu n, c.\, \overline{l}\langle n, c\rangle\; n.P + c(h, t).Q$$

Destructive Operations

$$
\begin{array}{lll}
\mathit{Copy}\langle l, m\rangle & \triangleq & \text{case } l \text{ of} \\
& & \mathit{Nil}? \Rightarrow \mathit{Nil}\langle m\rangle \\
& & \mathit{Cons}?(h, t) \Rightarrow \nu t'.\,(m(n, c).\bar{c}\langle h, t'\rangle \mid \mathit{Copy}\langle t, t'\rangle)
\end{array}
$$

Destructive Operations

$$
\begin{aligned}
\mathit{Copy}\langle l, m\rangle \quad &\triangleq \quad \text{case } l \text{ of} \\
&\quad \mathit{Nil}? \Rightarrow \mathit{Nil}\langle m\rangle \\
&\quad \mathit{Cons}?(h, t) \Rightarrow \nu t'.\,(m(n, c).\bar{c}\langle h, t'\rangle \mid \mathit{Copy}\langle t, t'\rangle)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Join}\langle k, l, m\rangle \quad &\triangleq \quad \text{case } k \text{ of} \\
&\quad \mathit{Nil}? \Rightarrow \mathit{Copy}\langle l, m\rangle \\
&\quad \mathit{Cons}?(h, t) \Rightarrow \nu t'.\,(m(n, c).\bar{c}\langle h, t'\rangle \mid \mathit{Join}\langle t, l, t'\rangle)
\end{aligned}
$$

Encoding Persistent Datatypes

We can put a ! in front of processes to turn them into servers create arbitrary numbers of the original process

$$
\begin{aligned}
Nil(l) &\triangleq !l(n, c).\bar{n} \\
Cons(H, T)(l) &\triangleq \nu h, t.\ (!l(n, c).\bar{c}\langle h, t\rangle \mid H\langle h\rangle \mid T\langle t\rangle)
\end{aligned}
$$

This causes the list to still exist after sending or receiving a message

Encoding λ-calculus

$$
\begin{aligned}
[\![x]\!](u) &\triangleq \bar{x}\langle u\rangle \\
[\![\lambda x.\, e]\!](u) &\triangleq u(x, y).[\![e]\!](y) \\
[\![e_1\; e_2]\!](u) &\triangleq \nu y.\, ([\![e_1]\!](y) \mid \nu x.\, (\bar{y}\langle x, u\rangle \mid\; !x(w).[\![e_2]\!](w)))
\end{aligned}
$$

Bisimulation

When are two processes equal?

One the most important contributions of research on π calculus has been the development of the notion of *bisimulation*:

$$\begin{tikzcd} & P \arrow[d, "a"'] & \\ & Q \arrow[dl, "b"'] \arrow[dr, "c"] & \\ R & & S \end{tikzcd}$$

Bisimulation

When are two processes equal?

One the most important contributions of research on π calculus has been the development of the notion of *bisimulation*: