CS 4110

Programming Languages & Logics

Lecture 36
Concurrency

3 December 2014

Announcements

- None!

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.

IMP with Parallel Composition

As a first step, let's extend IMP with a new a parallel composition command:

IMP with Parallel Composition

As a first step, let's extend IMP with a new a parallel composition command:

$$\begin{array}{rcl}
a & ::= & x \mid n \mid a_1 + a_2 \\
b & ::= & \textbf{true} \mid \textbf{false} \mid a_1 < a_2 \\
c & ::= & \textbf{skip} \mid x := a \mid c_1 ; c_2 \mid \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \mid \textbf{while}\ b\ \textbf{do}\ c \\
 & \mid & c_1 \mid\mid c_2
\end{array}$$

Operational Semantics

and extend the small-step operational semantics with the following rules for $c_1 \mid\mid c_2$, which interleave the execution of c_1 and c_2:

$$\frac{\langle \sigma, c_1 \rangle \rightarrow \langle \sigma', c_1' \rangle}{\langle \sigma, c_1 \mid\mid c_2 \rangle \rightarrow \langle \sigma', c_1' \mid\mid c_2 \rangle}$$

Operational Semantics

and extend the small-step operational semantics with the following rules for $c_1 \mid\mid c_2$, which interleave the execution of c_1 and c_2:

$$\frac{\langle \sigma, c_1 \rangle \rightarrow \langle \sigma', c_1' \rangle}{\langle \sigma, c_1 \mid\mid c_2 \rangle \rightarrow \langle \sigma', c_1' \mid\mid c_2 \rangle}$$

$$\frac{\langle \sigma, c_2 \rangle \rightarrow \langle \sigma .' c_2' \rangle}{\langle \sigma, c_1 \mid\mid c_2 \rangle \rightarrow \langle \sigma', c_1 \mid\mid c_2' \rangle}$$

Operational Semantics

and extend the small-step operational semantics with the following rules for $c_1 \mid\mid c_2$, which interleave the execution of c_1 and c_2:

$$\frac{\langle \sigma, c_1 \rangle \rightarrow \langle \sigma', c_1' \rangle}{\langle \sigma, c_1 \mid\mid c_2 \rangle \rightarrow \langle \sigma', c_1' \mid\mid c_2 \rangle}$$

$$\frac{\langle \sigma, c_2 \rangle \rightarrow \langle \sigma.' c_2' \rangle}{\langle \sigma, c_1 \mid\mid c_2 \rangle \rightarrow \langle \sigma', c_1 \mid\mid c_2' \rangle}$$

$$\langle \sigma, \textbf{skip} \mid\mid \textbf{skip} \rangle \rightarrow \langle \sigma, \textbf{skip} \rangle$$

Operational Semantics

and extend the small-step operational semantics with the following rules for $c_1 \mid\mid c_2$, which interleave the execution of c_1 and c_2:

$$\frac{\langle \sigma, c_1 \rangle \rightarrow \langle \sigma', c_1' \rangle}{\langle \sigma, c_1 \mid\mid c_2 \rangle \rightarrow \langle \sigma', c_1' \mid\mid c_2 \rangle}$$

$$\frac{\langle \sigma, c_2 \rangle \rightarrow \langle \sigma.' c_2' \rangle}{\langle \sigma, c_1 \mid\mid c_2 \rangle \rightarrow \langle \sigma', c_1 \mid\mid c_2' \rangle}$$

$$\langle \sigma, \textbf{skip} \mid\mid \textbf{skip} \rangle \rightarrow \langle \sigma, \textbf{skip} \rangle$$

Note that the rules for parallel compositions $c_1 \mid\mid c_2$ allow either sub-command to take a step; two sub-commands can interleave read and write operations involving the same store.

Process Calculi

In the 1970s, Tony Hoare, Robin Milner, and others correctly observed that in the future, computers would have multiple computing cores, but each would have its own independent store.

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.

π-calculus Syntax

The π-calculus is a minimal formalism that attempts to capture the essence message-passing concurrency

π-calculus Syntax

The π-calculus is a minimal formalism that attempts to capture the essence message-passing concurrency

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 attempts to capture the essence message-passing concurrency

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 attempts to capture the essence message-passing concurrency

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 attempts to capture the essence message-passing concurrency

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 attempts to capture the essence message-passing concurrency

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 attempts to capture the essence message-passing concurrency

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

Reaction

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

Reaction

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

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

Reaction

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

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

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

Reaction

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

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

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

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

Reaction

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

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

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

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

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

Structural Congruence

Definition (Congruence)

An equivalence relation $\mathcal{S}$ is a *congruence* if $P \mathrel{\mathcal{S}} Q$ implies $C[P] \mathrel{\mathcal{S}} C[Q]$ for every context C.

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 the scope extrusion axiom: $P \mid \nu x.\ Q \equiv \nu x.\ P \mid Q$.

Example

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

Example

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

Example

$$!x(u).\bar{x}\langle succ\ u \rangle$$

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:

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

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

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

Polyadic π-calculus

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

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

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

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

Encoding Recursion

Idea: Suppose we want to encode P where $A(\vec{x}) \triangleq P_A$.

- Pick a 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 $\nu a.\ \ ((\!|P|\!)\ \mid\ !a(\vec{x}).(\!|P_A|\!))$

Example: Buffer

Consider a recursive definition of a simple buffer:

$$\begin{aligned} B(l, r) &\triangleq r(x).C\langle x, l, r\rangle \\ C(x, l, r) &\triangleq \bar{l}\langle x\rangle.B\langle l, r\rangle \end{aligned}$$

Example: Buffer

Consider a recursive definition of a simple buffer:

$$\begin{aligned} B(l, r) &\triangleq r(x).C\langle x, l, r\rangle \\ C(x, l, r) &\triangleq \bar{l}\langle x\rangle.B\langle l, r\rangle \end{aligned}$$

When encoded this becomes

$$\nu b.\ \nu c.\ \left(\overline{b}\langle l, r\rangle \mid\ !b(l, r).r(x).\bar{c}\langle x, l, r\rangle \mid\ !c(x, l, r).\bar{l}\langle x\rangle.\overline{b}\langle l, r\rangle\right)$$