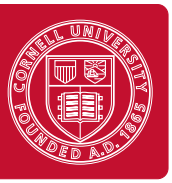
CS 4110 – Programming Languages and Logics
Lectures #26-27: Concurrency



Based on material by Stephen Chong, Greg Morrisett, Andrew Myers, George Necula, and Radu Rugina

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 steps of computation may happen simultaneously.

1 IMP with Parallel Composition

A simple way to add concurrency to a language is via a parallel composition operator. For example, we can add the $c_1 \mid\mid c_2$ command to IMP,

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

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:

$$\frac{\langle \sigma, c_1 \rangle \to \langle \sigma', c_1' \rangle}{\langle \sigma, c_1 \mid\mid c_2 \rangle \to \langle \sigma', c_1' \mid\mid c_2 \rangle} \qquad \frac{\langle \sigma, c_2 \rangle \to \langle \sigma.' c_2' \rangle}{\langle \sigma, c_1 \mid\mid c_2 \rangle \to \langle \sigma', c_1 \mid\mid c_2' \rangle} \qquad \frac{}{\langle \sigma, \textbf{skip} \mid\mid \textbf{skip} \rangle \to \langle \sigma, \textbf{skip} \rangle}$$

Note that the rules for parallel compositions $c_1 \mid\mid c_2$ allow either sub-command to take a step. In particular, the two sub-commands can interleave read and write operations involving the same store. This models a simple *shared memory* form of concurrency.

2 Communicating Sequential Processes

In the 1970s, Tony Hoare (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 so on.

CSP is organized around several new constructs for communicating information between processes. We assume a fixed collection of *channels*

$$\alpha, \beta, \gamma \in \textbf{Channels}$$

and add new constructs $\alpha\,!\,a$ for sending and $\alpha?x$ for receiving messages on channels. Intuitively, if one part of a process sends a value on a channel and another process is waiting to receive, the two will interact and the value will be transmitted from one to the other. We also add a restriction operation $c \backslash \alpha$ that allows a process c to hide a channel α. This is useful for encoding internal communication within a process. Finally, we add *guards* to the language. These are based on a language construct due to Dijkstra. Intuitively, a guarded command $b \to c$ can "fire" to c if b is true and a composition of guards $g_1 \,[\!]\, g_2$ can non-deterministically "fire" to any of the commands in g_1 or g_2 whose boolean guard evaluates to true.

2.1 Syntax

The syntax of CSP is as follows. (This presentation is a slight simplification of the language given in Winskel, Chapter 14.)

b	::=	...	boolean expressions
a	::=	...	arithmetic expressions
c	::=	**skip**	skip
	|	$x := a$	assignment
	|	$\alpha?x$	receive
	|	$\alpha\,!\,a$	send
	|	$c\backslash\alpha$	restriction
	|	$c_1 \mid\mid c_2$	parallel composition
	|	$c_1; c_2$	sequential composition
	|	**if** g **fi**	guarded conditional
	|	**do** g **od**	guarded loop
g	::=	$b \to c$	command guard
	|	$b \wedge \alpha?x \to c$	receive guard
	|	$b \wedge \alpha\,!\,a \to c$	send guard
	|	$g_1 [\!] g_2$	guard composition

To ensure that all communication between processes happens via sends and receives and not via shared memory we assume a simple well-formedness condition: in every parallel composition $c_1 \mid\mid c_2$, the set of locations referenced in c_1 and c_2 must be disjoint.

2.2 Operational Semantics for Commands

We formalize the operational semantics of CSP using a small-step transition relation between configurations of the form $\langle \sigma, c \rangle$. To keep track of the sends and receives performed by processes, we annotate the arrow with a label λ, writing $\langle \sigma, c \rangle \xrightarrow{\lambda} \langle \sigma', c' \rangle$ to indicate that the transition has an observable event λ. The syntax of labels λ is given by the following grammar:

$$\lambda ::= \epsilon \mid \alpha\,!\,n \mid \alpha?n$$

By convention we omit the label if it is ϵ, writing $\langle \sigma, c \rangle \to \langle \sigma', c' \rangle$ instead of $\langle \sigma, c \rangle \xrightarrow{\epsilon} \langle \sigma', c' \rangle$.

The operational semantics rules are given by the axioms and rules below. The most important rules are for the parallel composition of a send and receive, which allows two processes composed in parallel to communicate a value over a channel. Some of the rules use the standard large-step semantics for arithmetic and boolean expressions in IMP. Others use an auxiliary relation that captures the small-step semantics for guards, which is defined below.

$$\frac{\langle \sigma, a \rangle \Downarrow n}{\langle \sigma, x := a \rangle \to \langle \sigma[x \mapsto n], \textbf{skip} \rangle}$$

$$\frac{}{\langle \sigma, \alpha?x \rangle \xrightarrow{\alpha?n} \langle \sigma[x \mapsto n], \textbf{skip} \rangle} \qquad \frac{\langle \sigma, a \rangle \Downarrow n}{\langle \sigma, \alpha\,!\,a \rangle \xrightarrow{\alpha\,!\,n} \langle \sigma, \textbf{skip} \rangle}$$

$$\frac{\langle \sigma, c_1 \rangle \xrightarrow{\lambda} \langle \sigma, c_1' \rangle}{\langle \sigma, c_1; c_2 \rangle \xrightarrow{\lambda} \langle \sigma, c_1'; c_2 \rangle} \qquad \frac{}{\langle \sigma, \textbf{skip}; c_2 \rangle \to \langle \sigma, c_2 \rangle}$$

$$\frac{\langle \sigma, g \rangle \overset{\lambda}{\rightsquigarrow} \langle \sigma', c \rangle}{\langle \sigma, \textbf{if } g \textbf{ fi} \rangle \xrightarrow{\lambda} \langle \sigma', c \rangle} \qquad \frac{\langle \sigma, g \rangle \overset{\lambda}{\rightsquigarrow} \langle \sigma', c \rangle}{\langle \sigma, \textbf{do } g \textbf{ od} \rangle \xrightarrow{\lambda} \langle \sigma', c; \textbf{do } g \textbf{ od} \rangle} \qquad \frac{\langle \sigma, g \rangle \rightsquigarrow \textbf{fail}}{\langle \sigma, \textbf{do } g \textbf{ od} \rangle \to \langle \sigma, \textbf{skip} \rangle}$$

$$\frac{\langle \sigma, c_1 \rangle \xrightarrow{\lambda} \langle \sigma', c_1' \rangle}{\langle \sigma, c_1 \mid\mid c_2 \rangle \xrightarrow{\lambda} \langle \sigma', c_1' \mid\mid c_2 \rangle} \qquad \frac{\langle \sigma, c_2 \rangle \xrightarrow{\lambda} \langle \sigma', c_2' \rangle}{\langle \sigma, c_1 \mid\mid c_2 \rangle \xrightarrow{\lambda} \langle \sigma', c_1 \mid\mid c_2' \rangle} \qquad \frac{}{\langle \sigma, \textbf{skip} \mid\mid \textbf{skip} \rangle \to \langle \sigma, \textbf{skip} \rangle}$$

$$\frac{\langle \sigma, c_1 \rangle \xrightarrow{\alpha?n} \langle \sigma', c_1' \rangle \qquad \langle \sigma, c_2 \rangle \xrightarrow{\alpha!n} \langle \sigma, c_2' \rangle}{\langle \sigma, c_1 \mid\mid c_2 \rangle \to \langle \sigma', c_1' \mid\mid c_2' \rangle} \qquad \frac{\langle \sigma, c_1 \rangle \xrightarrow{\alpha!n} \langle \sigma, c_1' \rangle \qquad \langle \sigma, c_2 \rangle \xrightarrow{\alpha?n} \langle \sigma', c_2' \rangle}{\langle \sigma, c_1 \mid\mid c_2 \rangle \to \langle \sigma', c_1' \mid\mid c_2' \rangle}$$

$$\frac{\langle \sigma, c \rangle \xrightarrow{\lambda} \langle \sigma', c' \rangle}{\langle \sigma, c \backslash \alpha \rangle \xrightarrow{\lambda} \langle \sigma', c' \backslash \alpha \rangle} \ \lambda \neq \alpha?n \text{ and } \lambda \neq \alpha\,!\,n \qquad \frac{}{\langle \sigma, \textbf{skip} \backslash \alpha \rangle \to \langle \sigma, \textbf{skip} \rangle}$$

3 Operational Semantics for Guards

The small-step operational semantics for guards, written $\langle \sigma, c \rangle \overset{\lambda}{\rightsquigarrow} \langle \sigma', c' \rangle$ or $\langle \sigma, c \rangle \overset{\lambda}{\rightsquigarrow} \textbf{fail}$, is a binary relation from configurations to configurations or the special value **fail**. It non-deterministically evaluates a guarded command to a configuration $\langle \sigma, c \rangle$ if the boolean expression b in one of its sub-guards of the form $b \to c$ or $b \wedge \alpha\,!\,a \to c$ or $b \wedge \alpha?x \to c$ evaluates to **true**, and produces **fail** if no such sub-guard can be found.

$$\frac{\langle \sigma, b \rangle \Downarrow \textbf{true}}{\langle \sigma, b \to c \rangle \rightsquigarrow \langle \sigma, c \rangle} \qquad \frac{\langle \sigma, b \rangle \Downarrow \textbf{false}}{\langle \sigma, b \to c \rangle \rightsquigarrow \textbf{fail}}$$

$$\frac{\langle \sigma, b \rangle \Downarrow \textbf{true}}{\langle \sigma, b \wedge \alpha?x \to c \rangle \overset{\alpha?n}{\rightsquigarrow} \langle \sigma[x \mapsto n], c \rangle} \qquad \frac{\langle \sigma, b \rangle \Downarrow \textbf{false}}{\langle \sigma, b \wedge \alpha?x \to c \rangle \rightsquigarrow \textbf{fail}}$$

$$\frac{\langle \sigma, b \rangle \Downarrow \textbf{true} \qquad \langle \sigma, a \rangle \Downarrow n}{\langle \sigma, b \wedge \alpha\,!\,a \to c \rangle \overset{\alpha!n}{\rightsquigarrow} \langle \sigma, c \rangle} \qquad \frac{\langle \sigma, b \rangle \Downarrow \textbf{false}}{\langle \sigma, b \wedge \alpha\,!\,a \to c \rangle \rightsquigarrow \textbf{fail}}$$

$$\frac{\langle \sigma, g_1 \rangle \overset{\lambda}{\rightsquigarrow} \langle \sigma', c \rangle}{\langle \sigma, g_1 \,[\!]\, g_2 \rangle \overset{\lambda}{\rightsquigarrow} \langle \sigma', c \rangle} \qquad \frac{\langle \sigma, g_2 \rangle \overset{\lambda}{\rightsquigarrow} \langle \sigma', c \rangle}{\langle \sigma, g_1 \,[\!]\, g_2 \rangle \overset{\lambda}{\rightsquigarrow} \langle \sigma', c \rangle} \qquad \frac{\langle \sigma, g_1 \rangle \rightsquigarrow \textbf{fail} \qquad \langle \sigma, g_2 \rangle \rightsquigarrow \textbf{fail}}{\langle \sigma, g_1 \,[\!]\, g_2 \rangle \overset{\lambda}{\rightsquigarrow} \textbf{fail}}$$

3.1 Examples

To get a taste for programming in CSP, let us consider a few examples.

1. Here is a simple process that computes the max of two variables and sends it on a channel m:

$$\textbf{if } (x \leq y \rightarrow m\,!\,y) \,[\!]\, (y \leq x \rightarrow m\,!\,x) \textbf{ fi}$$

   Note that the two tests, which overlap, could be evaluated concurrently on an appropriate machine.

2. Now consider a process that repeatedly receives a value on α and transmits it on β:

$$\textbf{do } (\textbf{true} \wedge \alpha?x \rightarrow \beta\,!\,x) \textbf{ od}$$

   This implements a buffer of size one. This is useful as communication in CSP is synchronous. Using this buffer, a process that needs to send a value on α can continue without waiting for another process to receive it.

3. The next example illustrates a process that implements a buffer of size two:

$$(\textbf{do } (\textbf{true} \wedge \alpha?x \rightarrow \beta\,!\,x) \textbf{ od} \;||\; \textbf{do } (\textbf{true} \wedge \beta?y \rightarrow \gamma\,!\,y) \textbf{ od})\backslash\beta$$

   Note the use of restriction to hide the internal channel β.

4. Next consider the following two processes:

$$\textbf{if } (\textbf{true} \wedge \alpha?x \rightarrow c_1) \,[\!]\, (\textbf{true} \wedge \beta?x \rightarrow c_2) \textbf{ fi}$$
$$\textbf{if } (\textbf{true} \rightarrow \alpha?x; c_1) \,[\!]\, (\textbf{true} \rightarrow \beta?x; c_2) \textbf{ fi}$$

   These processes appear similar but they behave quite differently. Why?

5. Here is a process that merges two channels α and β onto a single channel γ:

$$\textbf{do } (\textbf{true} \wedge \alpha?x \rightarrow \gamma\,!\,x) \,[\!]\, (\textbf{true} \wedge \beta?x \rightarrow \gamma\,!\,x) \textbf{ od}$$

6. This process filters a channel α, only preserving the positive values:

$$\begin{array}{l}\textbf{do} \\ \quad \textbf{true} \wedge \alpha?x \rightarrow \\ \qquad \textbf{if } (x > 0 \rightarrow \beta\,!\,x) \,[\!]\, (x \leq 0 \rightarrow \textbf{skip}) \textbf{ fi} \\ \textbf{od}\end{array}$$

7. Here is a process that implements a division operator:

$$\begin{array}{l}\textbf{do} \\ \quad \textbf{true} \wedge in?x \rightarrow \\ \quad \textbf{true} \wedge in?y \rightarrow \\ \qquad q := 0; \\ \qquad r := x; \\ \qquad \textbf{do } (r \geq y \rightarrow \\ \qquad\qquad r := r - y; \\ \qquad\qquad q := q + 1) \\ \qquad \textbf{od}; \\ \qquad out\,!\,q; \\ \qquad out\,!\,r \\ \textbf{od}\end{array}$$

8. What does this process do? (Hint: it's like the buffer we saw before but...)

$$\begin{array}{l}\textbf{do true} \wedge \alpha?x \rightarrow \\ \qquad (\beta\,!\,x \rightarrow \alpha?x \rightarrow \textbf{skip}) \\ \textbf{od}\end{array}$$

9. What about this one?

$$\begin{array}{l}\textbf{do} \\ \quad \textbf{true} \wedge \alpha?x \rightarrow \\ \qquad \textbf{do} \\ \qquad\quad \textbf{true} \wedge \alpha?y \rightarrow \\ \qquad\qquad \beta\,!\,x \rightarrow x := y \\ \qquad \textbf{od} \\ \textbf{od}\end{array}$$

10. As a final example, here is a simple encoding of Dijkstra's dining philosophers in CSP (note this is *not* a solution—it can easily deadlock!):

$$phil_i = get_i\,!\,i; get_{(i+1)\%n}\,!\,i; nom\,!\,i; put_i\,!\,i; put_{(i+1)\%n}\,!\,i$$

$$fork_i = \textbf{do}\ (\textbf{true} \wedge get_i?x \rightarrow put_i?x)\ \textbf{od}$$

$$eat = \textbf{do}\ (\textbf{true} \wedge nom?x \rightarrow \textbf{skip})\ \textbf{od}$$

$$phil_0 \,||\, phil_1 \,||\, phil_2 \,||\, phil_3 \,||\, phil_4 \,||\, fork_0 \,||\, fork_1 \,||\, fork_2 \,||\, fork_3 \,||\, fork_4 \,||\, eat$$