Shared-Variable Concurrency

Parallel Composition (or Concurrency Composition)

Syntax:

$$(\text{Comm})\ c ::= \ldots \mid c \parallel c$$

Note we allow nested parallel composition, e.g., $(c_0\,;(c_1 \parallel c_2))$.

Parallel Composition (or Concurrency Composition)

Syntax:

$$(\text{Comm})\ c ::= \ldots \mid c \parallel c$$

Note we allow nested parallel composition, e.g., $(c_0\,\mathbf{;}\,(c_1 \parallel c_2))$.

Operational Semantics:

$$\frac{(c_0, \sigma) \longrightarrow (c_0', \sigma')}{(c_0 \parallel c_1, \sigma) \longrightarrow (c_0' \parallel c_1, \sigma')} \qquad \frac{(c_1, \sigma) \longrightarrow (c_1', \sigma')}{(c_0 \parallel c_1, \sigma) \longrightarrow (c_0 \parallel c_1', \sigma')}$$

$$\frac{}{(\mathbf{skip} \parallel \mathbf{skip}, \sigma) \longrightarrow (\mathbf{skip}, \sigma)} \qquad \frac{(c_i, \sigma) \longrightarrow (\mathbf{abort}, \sigma') \quad i \in \{0, 1\}}{(c_0 \parallel c_1, \sigma) \longrightarrow (\mathbf{abort}, \sigma')}$$

We have to use small-step semantics (instead of big-step semantics) to model concurrency.

Examples (1)

$$\begin{array}{l} y := x + 1\,\mathbf{;} \\ x := y + 1 \end{array} \quad \Big\| \quad \begin{array}{l} y := x + 1\,\mathbf{;} \\ x := x + 1 \end{array}$$

Suppose initially $\sigma(x) = \sigma(y) = 0$. What are the possible results?

$(1) y = 1, x = 2$; $(2) y = 1, x = 3$; $(3) y = 3, x = 3$; $(4) y = 2, x = 3$

Examples (2)

Search for a non-negative result of the function f:

$$
\begin{array}{l}
k := -1\,\mathbf{;} \\[1ex]
(\mathbf{newvar}\ i := 0\ \mathbf{in\ while}\ i \leq n \wedge k = -1\ \mathbf{do} \\
\quad \mathbf{if}\ f(i) \geq 0\ \mathbf{then}\ k := i\ \mathbf{else}\ i := i + 2 \\[1ex]
\parallel \mathbf{newvar}\ i := 1\ \mathbf{in\ while}\ i \leq n \wedge k = -1\ \mathbf{do} \\
\quad \mathbf{if}\ f(i) \geq 0\ \mathbf{then}\ k := i\ \mathbf{else}\ i := i + 2)
\end{array}
$$

Examples (2)

Search for a non-negative result of the function f:

$$
\begin{array}{l}
k := -1\,\mathbf{;} \\[6pt]
(\mathbf{newvar}\ i := 0\ \mathbf{in\ while}\ i \leq n \wedge k = -1\ \mathbf{do} \\
\quad \mathbf{if}\ f(i) \geq 0\ \mathbf{then}\ k := i\ \mathbf{else}\ i := i + 2 \\[6pt]
\parallel \mathbf{newvar}\ i := 1\ \mathbf{in\ while}\ i \leq n \wedge k = -1\ \mathbf{do} \\
\quad \mathbf{if}\ f(i) \geq 0\ \mathbf{then}\ k := i\ \mathbf{else}\ i := i + 2)
\end{array}
$$

A problematic version:

$$
\begin{array}{l}
k := -1\,\mathbf{;} \\[6pt]
(\mathbf{newvar}\ i := 0\ \mathbf{in\ while}\ i \leq n \wedge k = -1\ \mathbf{do} \\
\quad \mathbf{if}\ f(i) \geq 0\ \mathbf{then\ print}(i)\,\mathbf{;}\,\mathbf{print}(f(i))\ \mathbf{else}\ i := i + 2 \\[6pt]
\parallel \mathbf{newvar}\ i := 1\ \mathbf{in\ while}\ i \leq n \wedge k = -1\ \mathbf{do} \\
\quad \mathbf{if}\ f(i) \geq 0\ \mathbf{then\ print}(i)\,\mathbf{;}\,\mathbf{print}(f(i))\ \mathbf{else}\ i := i + 2)
\end{array}
$$

Conditional Critical Regions

We could use a critical region to achieve mutual exclusive access of shared variables.

Syntax:

$$(\text{Comm}) \quad c \; ::= \ldots \mid \textbf{await } b \textbf{ then } \hat{c}$$

where $\hat{c}$ is a sequential command (a command with no **await** and parallel composition).

Conditional Critical Regions

We could use a critical region to achieve mutual exclusive access of shared variables.

Syntax:

$$(\text{Comm})\ \ c \ ::= \ldots \mid \textbf{await}\ b\ \textbf{then}\ \hat{c}$$

where $\hat{c}$ is a sequential command (a command with no **await** and parallel composition).

Semantics:

$$\frac{[\![b]\!]_{boolexp}\,\sigma = \textbf{true} \qquad (\hat{c}, \sigma) \longrightarrow^* (\textbf{skip}, \sigma')}{(\textbf{await}\ b\ \textbf{then}\ \hat{c}, \sigma) \longrightarrow (\textbf{skip}, \sigma')}$$

$$\frac{[\![b]\!]_{boolexp}\,\sigma = \textbf{false}}{(\textbf{await}\ b\ \textbf{then}\ \hat{c}, \sigma) \longrightarrow (\textbf{skip}\,;\textbf{await}\ b\ \textbf{then}\ \hat{c}, \sigma)}$$

The second rule gives us a “busy-waiting” semantics. If we eliminate that rule, the thread will be blocked when the condition does not hold.

Achieving Mutual Exclusion

$$
\begin{array}{l}
k := -1\,\mathbf{;}\\[1ex]
(\mathbf{newvar}\ i := 0\ \mathbf{in\ while}\ i \leq n \wedge k = -1\ \mathbf{do}\\
\quad (\mathbf{if}\ f(i) \geq 0\ \mathbf{then}\ (\mathbf{await}\ busy = 0\ \mathbf{then}\ busy := 1)\,\mathbf{;}\\
\qquad\qquad\qquad\qquad\quad \mathbf{print}(i)\,\mathbf{;}\,\mathbf{print}(f(i))\,\mathbf{;}\,busy := 0\\
\qquad\qquad\quad \mathbf{else}\ i := i + 2)\\[1ex]
\parallel \mathbf{newvar}\ i := 1\ \mathbf{in\ while}\ i \leq n \wedge k = -1\ \mathbf{do}\\
\quad (\mathbf{if}\ f(i) \geq 0\ \mathbf{then}\ (\mathbf{await}\ busy = 0\ \mathbf{then}\ busy := 1)\,\mathbf{;}\\
\qquad\qquad\qquad\qquad\quad \mathbf{print}(i)\,\mathbf{;}\,\mathbf{print}(f(i))\,\mathbf{;}\,busy := 0\\
\qquad\qquad\quad \mathbf{else}\ i := i + 2))
\end{array}
$$

Atomic Blocks

A syntactic sugar:

$$\textbf{atomic}\{c\} \stackrel{\text{def}}{=} \textbf{await true then } c$$

We may also use the short-hand notation $\langle c \rangle$.

Semantics:

$$\frac{(c, \sigma) \longrightarrow^* (\textbf{skip}, \sigma')}{(\textbf{atomic}\{c\}, \sigma) \longrightarrow (\textbf{skip}, \sigma')}$$

It gives the programmer control over the size of atomic actions.

Deadlock

$$
\begin{array}{l|l}
\textbf{await } \mathit{busy}0 = 0 & \textbf{await } \mathit{busy}1 = 0 \\
\quad \textbf{then } \mathit{busy}0 := 1; & \quad \textbf{then } \mathit{busy}1 := 1; \\
\textbf{await } \mathit{busy}1 = 0 & \textbf{await } \mathit{busy}0 = 0 \\
\quad \textbf{then } \mathit{busy}1 := 1; & \quad \textbf{then } \mathit{busy}0 := 1; \\
\ldots & \ldots \\
\mathit{busy}0 := 0; & \mathit{busy}0 := 0; \\
\mathit{busy}1 := 0; & \mathit{busy}1 := 0;
\end{array}
$$

Fairness

$$
\begin{array}{l}
k := -1; \\
(\textbf{newvar } i := 0 \textbf{ in while } k = -1 \textbf{ do} \\
\quad \textbf{if } f(i) \geq 0 \textbf{ then } k := i \textbf{ else } i := i + 2 \\
\parallel \textbf{newvar } i := 1 \textbf{ in while } k = -1 \textbf{ do} \\
\quad \textbf{if } f(i) \geq 0 \textbf{ then } k := i \textbf{ else } i := i + 2)
\end{array}
$$

Suppose $f(i) < 0$ for all even number i. Then there's an infinite execution in the form of:

$$\ldots \longrightarrow (c_1 \parallel c', \sigma_1) \longrightarrow (c_2 \parallel c', \sigma_2) \longrightarrow \ldots \longrightarrow (c_n \parallel c', \sigma_n) \longrightarrow \ldots$$

An execution of concurrent processes is *unfair* if it does not terminate but, after some finite number of steps, there is an unterminated process that never makes a transition.

Fairness — More Examples

A fair execution of the following program would always terminate:

$$\textbf{newvar}\ y := 0\ \textbf{in}\ (x := 0\textbf{;}((\textbf{while}\ y = 0\ \textbf{do}\ x := x + 1) \parallel y := 1))$$

Fairness — More Examples

A fair execution of the following program would always terminate:

$$\textbf{newvar}\ y := 0\ \textbf{in}\ (x := 0\textbf{;}((\textbf{while}\ y = 0\ \textbf{do}\ x := x + 1) \parallel y := 1))$$

Stronger fairness is needed to rule out infinite execution of the following program:

$$
\begin{array}{l}
\textbf{newvar}\ y := 0\ \textbf{in} \\
\quad (x := 0\textbf{;} \\
\quad\ ((\textbf{while}\ y = 0\ \textbf{do}\ x := 1 - x) \parallel (\textbf{await}\ x = 1\ \textbf{then}\ y := 1)) \\
\quad)
\end{array}
$$

Trace Semantics

Can we give a denotational semantics to concurrent programs?

Trace Semantics

Can we give a denotational semantics to concurrent programs?
Here we use *transition traces* to model the execution of programs.

Execution of (c_0, σ_0) in a concurrent setting:

$$(c_0, \sigma_0) \longrightarrow (c_1, \sigma'_0), (c_1, \sigma_1) \longrightarrow (c_2, \sigma'_1), \ldots, (c_{n-1}, \sigma_{n-1}) \longrightarrow (\mathbf{skip}, \sigma'_{n-1})$$

The gap between σ'_i and σ_{i+1} reflects the intervention of the environment (other threads).
It could be infinite if (c_0, σ_0) does not terminate:

$$(c_0, \sigma_0) \longrightarrow (c_1, \sigma_1), (c_1, \sigma'_1) \longrightarrow (c_2, \sigma_2), \ldots$$

We omit the commands to get a transition trace:

$$(\sigma_0, \sigma'_0), (\sigma_1, \sigma'_1), \ldots, (\sigma_{n-1}, \sigma'_{n-1})$$

or $$(\sigma_0, \sigma'_0), (\sigma_1, \sigma'_1), \ldots$$

Interference-Free Traces

A trace $(\sigma_0, \sigma'_0), (\sigma_1, \sigma'_1), \ldots, (\sigma_{n-1}, \sigma'_{n-1})$ (or $(\sigma_0, \sigma'_0), (\sigma_1, \sigma'_1), \ldots)$ is said to be *Interference-Free* iff $\forall i.\, \sigma'_i = \sigma_{i+1}$.

Operations over Traces

We use τ to represent individual transition traces, and $\mathcal{T}$ for a set of traces.

$$
\begin{array}{lll}
\epsilon & & \text{empty trace} \\
\tau_1 +\!\!+ \tau_2 & \stackrel{\text{def}}{=} & \text{concatenation of } \tau_1 \text{ and } \tau_2 \\
 & & \quad \tau_1 \text{ if } \tau_1 \text{ is infinite.} \\
\mathcal{T}_1 \,;\, \mathcal{T}_2 & \stackrel{\text{def}}{=} & \{\tau_1 +\!\!+ \tau_2 \mid \tau_1 \in \mathcal{T}_1 \text{ and } \tau_2 \in \mathcal{T}_2\} \\
\mathcal{T}^0 & \stackrel{\text{def}}{=} & \{\epsilon\} \\
\mathcal{T}^{n+1} & \stackrel{\text{def}}{=} & \mathcal{T} \,;\, \mathcal{T}^n \\
\mathcal{T}^* & \stackrel{\text{def}}{=} & \bigcup_{n=0}^{\infty} \mathcal{T}^n \\
\mathcal{T}^\omega & \stackrel{\text{def}}{=} & \{\tau_0 +\!\!+ \tau_1 +\!\!+ \ldots \mid \tau_i \in \mathcal{T}\}
\end{array}
$$

Note the difference between $\mathcal{T}^*$ and $\mathcal{T}^\omega$.

Trace Semantics — First Try

$$\mathcal{T}[\![x := e]\!] = \Big\{(\sigma, \sigma') \mid \sigma' = \sigma\{x \leadsto [\![e]\!]_{intexp}\,\sigma\}\Big\}$$

$$\mathcal{T}[\![\mathbf{skip}]\!] = \{(\sigma, \sigma) \mid \sigma \in \Sigma\}$$

$$\mathcal{T}[\![c_0 \,\mathbf{;}\, c_1]\!] = \mathcal{T}[\![c_0]\!] \,\mathbf{;}\, \mathcal{T}[\![c_1]\!]$$

$$\mathcal{T}[\![\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2]\!] = (\mathcal{B}[\![b]\!] \,\mathbf{;}\, \mathcal{T}[\![c_1]\!]) \cup (\mathcal{B}[\![\neg b]\!] \,\mathbf{;}\, \mathcal{T}[\![c_2]\!])$$

$$\text{where } \mathcal{B}[\![b]\!] = \{(\sigma, \sigma) \mid [\![b]\!]_{boolexp}\,\sigma = \mathbf{true}\}$$

$$\mathcal{T}[\![\mathbf{while}\ b\ \mathbf{do}\ c]\!] = ((\mathcal{B}[\![b]\!] \,\mathbf{;}\, \mathcal{T}[\![c]\!])^* \,\mathbf{;}\, \mathcal{B}[\![\neg b]\!]) \cup (\mathcal{B}[\![b]\!] \,\mathbf{;}\, \mathcal{T}[\![c]\!])^\omega$$

Trace Semantics (cont'd)

How to give semantics to **newvar** $x := e$ **in** c?

Definition: *local-global*$(x, e, \tau, \hat{\tau})$ iff the following are true (suppose $\tau = (\sigma_0, \sigma'_0), (\sigma_1, \sigma'_1), \ldots$ and $\hat{\tau} = (\hat{\sigma}_0, \hat{\sigma}'_0), (\hat{\sigma}_1, \hat{\sigma}'_1), \ldots$):

- they have the same length;
- for all $x' \neq x$, $\sigma_i\, x' = \hat{\sigma}_i\, x'$ and $\sigma'_i\, x' = \hat{\sigma}'_i\, x'$;
- for all i, $\sigma_{i+1}\, x = \sigma'_i\, x$;
- for all i, $\hat{\sigma}_i\, x = \hat{\sigma}'_i\, x$;
- $\sigma_0\, x = [\![e]\!]_{intexp}\, \hat{\sigma}_0$.

$$\mathcal{T}[\![\textbf{newvar}\, x := e \textbf{ in } c]\!] = \{\hat{\tau} \mid \tau \in \mathcal{T}[\![c]\!] \text{ and } \textit{local-global}(x, e, \tau, \hat{\tau})\}$$

Fair Interleaving

We view a trace τ as a function mapping indices to the corresponding transitions.

Definition: *fair-merge*(τ_1, τ_2, τ) iff there exist functions $f \in \mathrm{dom}(\tau_1) \to \mathrm{dom}(\tau)$ and $g \in \mathrm{dom}(\tau_2) \to \mathrm{dom}(\tau)$ such that the following are true:

- f and g are *monotone injections*:

$$i < j \Longrightarrow (f\, i < f\, j) \wedge (g\, i < g\, j)$$

- $\mathrm{ran}(f) \cap \mathrm{ran}(g) = \emptyset$ and $\mathrm{ran}(f) \cup \mathrm{ran}(g) = \mathrm{dom}(\tau)$;
- $\forall i.\, \tau_1(i) = \tau(f\, i) \wedge \tau_2(i) = \tau(g\, i)$

Then $\mathcal{T}_{fair}[\![c_1 \parallel c_2]\!] =$
$\{\tau \mid \exists \tau_1 \in \mathcal{T}_{fair}[\![c_1]\!], \tau_2 \in \mathcal{T}_{fair}[\![c_2]\!].\ \textit{fair-merge}(\tau_1, \tau_2, \tau)\}$

Unfair Interleaving

Definition: *unfair-merge*(τ_1, τ_2, τ) if one of the following are true:

- *fair-merge*(τ_1, τ_2, τ)
- τ_1 is infinite and there exist τ_2' and τ_2'' such that $\tau_2 = \tau_2' +\!\!+ \tau_2''$ and *fair-merge*(τ_1, τ_2', τ)
- τ_2 is infinite, and there exist τ_1' and τ_1'' such that $\tau_1 = \tau_1' +\!\!+ \tau_1''$ and *fair-merge*(τ_1', τ_2, τ)

$$\begin{aligned}&\mathcal{T}_{unfair}[\![c_1 \parallel c_2]\!] \\ &\quad = \{\tau \mid \exists \tau_1 \in \mathcal{T}_{unfair}[\![c_1]\!], \tau_2 \in \mathcal{T}_{unfair}[\![c_2]\!].\ \textit{unfair-merge}(\tau_1, \tau_2, \tau)\}\end{aligned}$$

Trace Semantics for **await**

$$
\begin{aligned}
&\mathcal{T}[\![\textbf{await}\ b\ \textbf{then}\ c]\!] \ = \\
&\quad (\mathcal{B}[\![\neg b]\!]\ \mathbf{;}\, \mathcal{T}[\![\textbf{skip}]\!])^{*}\ \mathbf{;} \\
&\qquad \{(\sigma, \sigma') \mid [\![b]\!]_{boolexp}\, \sigma = \textbf{true} \\
&\qquad\qquad \text{and there exist } \sigma_0', \sigma_1, \sigma_1', \ldots, \sigma_n \text{ such that} \\
&\qquad\qquad\quad (\sigma, \sigma_0'), (\sigma_1, \sigma_1'), \ldots, (\sigma_n, \sigma') \in \mathcal{T}[\![c]\!] \\
&\qquad\qquad\qquad \text{and it is Interference-Free.}\} \\
&\quad \cup (\mathcal{B}[\![\neg b]\!]\ \mathbf{;}\, \mathcal{T}[\![\textbf{skip}]\!])^{\omega}
\end{aligned}
$$

Trace Semantics (cont'd)

The semantics is equivalent to the following:

$$
\begin{aligned}
\mathcal{T}[\![c]\!] \stackrel{\text{def}}{=} & \\
& \{(\sigma_0, \sigma'_0), \ldots, (\sigma_n, \sigma'_n) \mid \\
& \quad \text{there exist } c_0, \ldots, c_n \text{ such that } c_0 = c, \\
& \qquad \forall i \in [0, n-1].\, (c_i, \sigma_i) \longrightarrow (c_{i+1}, \sigma'_i), \\
& \qquad \text{and } (c_n, \sigma_n) \longrightarrow (\mathbf{skip}, \sigma'_n)\} \\
& \cup \{(\sigma_0, \sigma'_0), (\sigma_1, \sigma'_1), \ldots \mid \\
& \qquad \text{there exist } c_0, c_1, \ldots \text{ such that } c_0 = c, \\
& \qquad\quad \text{and for all } i, (c_i, \sigma_i) \longrightarrow (c_{i+1}, \sigma'_i)\}
\end{aligned}
$$

Problem with This Semantics

The trace semantics we just defined is not abstract enough.
It distinguishes the following programs (which should be viewed equivalent):

$$x := x+1$$

$$x := x+1\ \mathbf{;}\ \mathbf{skip}$$

$$\mathbf{skip}\ \mathbf{;}\ x := x+1$$

Also consider the following two programs:

$$x := x+1\ \mathbf{;}\ x := x+1$$

$$(x := x+1\ \mathbf{;}\ x := x+1)\ \mathbf{choice}\ x := x+2$$

Stuttering and Mumbling

$$\frac{}{\tau \prec \tau} \qquad \frac{}{\tau \prec (\sigma,\sigma),\tau} \qquad \frac{}{(\sigma,\sigma'),(\sigma',\sigma''),\tau \prec (\sigma,\sigma''),\tau}$$

$$\frac{\tau \prec \tau' \quad \tau' \prec \tau''}{\tau \prec \tau''} \qquad \frac{\tau \prec \tau'}{(\sigma,\sigma'),\tau \prec (\sigma,\sigma'),\tau'}$$

$$\mathcal{T}^{\dagger} \overset{\text{def}}{=} \{\tau \mid \tau \in \mathcal{T} \text{ or } \exists \tau' \in \mathcal{T}.\, \tau' \prec \tau\}$$

$$\mathcal{T}^{*}[\![c]\!] \overset{\text{def}}{=} (\mathcal{T}[\![c]\!])^{\dagger}$$

Stuttering and Mumbling (cont'd)

The new semantics $\mathcal{T}^*[\![c]\!]$ is equivalent to the following:

$$
\begin{array}{l}
\mathcal{T}[\![c]\!] \stackrel{\text{def}}{=} \\
\quad \{(\sigma_0, \sigma'_0), \ldots, (\sigma_n, \sigma'_n) \mid \\
\qquad \text{there exist } c_0, \ldots, c_n \text{ such that } c_0 = c, \\
\qquad\quad \forall i \in [0, n-1].\, (c_i, \sigma_i) \longrightarrow^* (c_{i+1}, \sigma'_i), \\
\qquad\quad \text{and } (c_n, \sigma_n) \longrightarrow^* (\textbf{skip}, \sigma'_n)\} \\
\quad \cup \{(\sigma_0, \sigma'_0), (\sigma_1, \sigma'_1), \ldots \mid \\
\qquad\quad \text{there exist } c_0, c_1, \ldots \text{ such that } c_0 = c, \\
\qquad\qquad \forall i.\, (c_i, \sigma_i) \longrightarrow^* (c_{i+1}, \sigma'_i), \\
\qquad\qquad \text{and for infinitely many } i \geq 0, (c_i, \sigma_i) \longrightarrow^+ (c_{i+1}, \sigma'_i)\}
\end{array}
$$