

Shared-Variable Concurrency

Xinyu Feng

Nanjing University

12/14/2018

Parallel Composition (or Concurrency Composition)

Syntax:

$$(comm) c ::= \dots \mid c_0 \parallel c_1 \mid \dots$$

Note we allow nested parallel composition, e.g.,

$$(c_0 ; (c_1 \parallel c_2)) \parallel c_3.$$

Operational Semantics:

$$\frac{(c_0, \sigma) \longrightarrow (c'_0, \sigma')}{(c_0 \parallel c_1, \sigma) \longrightarrow (c'_0 \parallel c_1, \sigma')}$$

$$\frac{(c_1, \sigma) \longrightarrow (c'_1, \sigma')}{(c_0 \parallel c_1, \sigma) \longrightarrow (c_0 \parallel c'_1, \sigma')}$$

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

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

Example:

$$\begin{array}{l} y := x + 1 ; \\ x := y + 1 \end{array} \quad \parallel \quad \begin{array}{l} y := x + 1 ; \\ x := x + 1 \end{array}$$

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) $y = 2, x = 3$

Two commands c_0 and c_1 are said to *interfere* if:

$$(fv(c_0) \cap fa(c_1)) \cup (fv(c_1) \cap fa(c_0)) \neq \emptyset$$

If c_0 and c_1 interfere, we say there are *race conditions* (or *races*) in $c_0 \parallel c_1$.

When c_0 and c_1 do not interfere, nor terminate by failure, the concurrent composition $c_0 \parallel c_1$ is determinate.

Another Example

A *benign race*:

```
k := -1;  
  
(newvar i := 0 in while i ≤ n ∧ k = -1 do  
  if f(i) ≥ 0 then k := i else i := i + 2  
  
|| newvar i := 1 in while i ≤ n ∧ k = -1 do  
  if f(i) ≥ 0 then k := i else i := i + 2)
```

A problematic version:

```
k := -1;  
  
(newvar i := 0 in while i ≤ n ∧ k = -1 do  
  if f(i) ≥ 0 then print(i) ; print(f(i)) else i := i + 2  
  
|| newvar i := 1 in while i ≤ n ∧ k = -1 do  
  if f(i) ≥ 0 then print(i) ; print(f(i)) else i := i + 2)
```

Conditional Critical Regions

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

Syntax:

$$(comm) \ c ::= \mathbf{await} \ b \ \mathbf{then} \ \hat{c}$$

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

Semantics:

$$\frac{\llbracket b \rrbracket_{boolexp} \sigma = \mathbf{true} \quad (\hat{c}, \sigma) \longrightarrow^* (\mathbf{Skip}, \sigma')}{(\mathbf{await} \ b \ \mathbf{then} \ \hat{c}, \sigma) \longrightarrow (\mathbf{Skip}, \sigma')}$$

$$\frac{\llbracket b \rrbracket_{boolexp} \sigma = \mathbf{false}}{(\mathbf{await} \ b \ \mathbf{then} \ \hat{c}, \sigma) \longrightarrow (\mathbf{Skip} ; \mathbf{await} \ b \ \mathbf{then} \ \hat{c}, \sigma)}$$

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

```
 $k := -1;$ 
```

```
(newvar  $i := 0$  in while  $i \leq n \wedge k = -1$  do  
  (if  $f(i) \geq 0$  then (await  $busy = 0$  then  $busy := 1$ );  
    print( $i$ ); print( $f(i)$ );  $busy := 0$   
    else  $i := i + 2$ )
```

```
|| newvar  $i := 1$  in while  $i \leq n \wedge k = -1$  do  
  (if  $f(i) \geq 0$  then (await  $busy = 0$  then  $busy := 1$ );  
    print( $i$ ); print( $f(i)$ );  $busy := 0$   
    else  $i := i + 2$ ))
```

Atomic Blocks

A syntactic sugar:

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

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

Semantics:

$$\frac{(c, \sigma) \longrightarrow^* (\mathbf{Skip}, \sigma')}{(\mathbf{atomic}\{c\}, \sigma) \longrightarrow (\mathbf{Skip}, \sigma')}$$

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

Reynolds uses $\mathbf{crit\ } c$ instead of $\mathbf{atomic}\{c\}$.

Deadlock

```
await busy0 = 0
  then busy0 := 1;
await busy1 = 0
  then busy1 := 1;    ||
...
busy0 := 0;
busy1 := 0;

await busy1 = 0
  then busy1 := 1;
await busy0 = 0
  then busy0 := 1;
...
busy0 := 0;
busy1 := 0;
```

```
k := -1;  
(newvar i := 0 in while k = -1 do  
  if f(i) ≥ 0 then k := i else i := i + 2  
|| newvar i := 1 in while k = -1 do  
  if f(i) ≥ 0 then k := i else i := i + 2)
```

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

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

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:

```
newvar y := 0 in (x := 0; ((while y = 0 do x := x + 1) || y := 1))
```

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

```
newvar y := 0 in  
  (x := 0;  
   ((while y = 0 do x := 1 - x) || (await x = 1 then y := 1))  
  )
```

Trace Semantics

Can we give a denotational semantics to concurrent programs?
The domain-based approach is complex. Here we use *transition traces* to model the execution of programs.

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

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

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:

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

We omit the commands to get a transition trace:

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

or

$$(\sigma_0, \sigma'_0), (\sigma_1, \sigma'_1), \dots$$

Interference-Free Traces

A trace $(\sigma_0, \sigma'_0), (\sigma_1, \sigma'_1), \dots, (\sigma_{n-1}, \sigma'_{n-1})$ (or $(\sigma_0, \sigma'_0), (\sigma_1, \sigma'_1), \dots)$ 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.

ϵ empty trace

$\tau_1 \mathbin{++} \tau_2 \stackrel{\text{def}}{=} \text{concatenation of } \tau_1 \text{ and } \tau_2$
 τ_1 if τ_1 is infinite.

$\mathcal{T}_1 ; \mathcal{T}_2 \stackrel{\text{def}}{=} \{\tau_1 \mathbin{++} \tau_2 \mid \tau_1 \in \mathcal{T}_1 \text{ and } \tau_2 \in \mathcal{T}_2\}$

$\mathcal{T}^0 \stackrel{\text{def}}{=} \{\epsilon\}$

$\mathcal{T}^{n+1} \stackrel{\text{def}}{=} \mathcal{T} ; \mathcal{T}^n$

$\mathcal{T}^* \stackrel{\text{def}}{=} \bigcup_{n=0}^{\infty} \mathcal{T}^n$

$\mathcal{T}^\omega \stackrel{\text{def}}{=} \{\tau_0 \mathbin{++} \tau_1 \mathbin{++} \dots \mid \tau_i \in \mathcal{T}\}$

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

Trace Semantics — First Try

$$\mathcal{T}[\![x := e]\!] = \{(\sigma, \sigma') \mid \sigma' = \sigma\{x \rightsquigarrow \llbracket e \rrbracket_{intexp} \sigma}\}$$

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

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

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

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

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

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 $\tau = (\sigma_0, \sigma'_0), (\sigma_1, \sigma'_1), \dots$ and $\hat{\tau} = (\hat{\sigma}_0, \hat{\sigma}'_0), (\hat{\sigma}_1, \hat{\sigma}'_1), \dots$):

- 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 , $\hat{\sigma}_i x = \hat{\sigma}'_i x$;
- $\sigma_0 x = \llbracket e \rrbracket_{\text{intexp}} \hat{\sigma}_0$.

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

Fair Interleaving

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

Definition: $\text{fair-merge}(\tau_1, \tau_2, \tau)$ iff there exist functions $f \in \text{dom}(\tau_1) \rightarrow \text{dom}(\tau)$ and $g \in \text{dom}(\tau_2) \rightarrow \text{dom}(\tau)$ such that the following are true:

- f and g are *monotone injections*:

$$i < j \implies (f i < f j) \wedge (g i < g j)$$

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

Then $\mathcal{T}_{\text{fair}}[\![c_1 \parallel c_2]\!] =$

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

Definition: $unfair\text{-merge}(\tau_1, \tau_2, \tau)$ if one of the following are true:

- $fair\text{-merge}(\tau_1, \tau_2, \tau)$
- τ_1 is infinite and there exist τ'_2 and τ''_2 such that $\tau_2 = \tau'_2 ++ \tau''_2$ and $fair\text{-merge}(\tau_1, \tau'_2, \tau)$
- τ_2 is infinite, and there exist τ'_1 and τ''_1 such that $\tau_1 = \tau'_1 ++ \tau''_1$ and $fair\text{-merge}(\tau'_1, \tau_2, \tau)$

$$\begin{aligned} \mathcal{T}_{unfair} \llbracket c_1 \parallel c_2 \rrbracket \\ = \{ \tau \mid \exists \tau_1 \in \mathcal{T}_{unfair} \llbracket c_1 \rrbracket, \tau_2 \in \mathcal{T}_{unfair} \llbracket c_2 \rrbracket. unfair\text{-merge}(\tau_1, \tau_2, \tau) \} \end{aligned}$$

$$\begin{aligned} \mathcal{T}[\mathbf{await} \ b \ \mathbf{then} \ c] = & \\ & (\mathcal{B}[\neg b] ; \mathcal{T}[\mathbf{Skip}])^* ; \\ & \{(\sigma, \sigma') \mid \llbracket b \rrbracket_{\text{bool exp}} \sigma = \mathbf{true} \\ & \quad \text{and there exist } \sigma'_0, \sigma_1, \sigma'_1, \dots, \sigma_n \text{ such that} \\ & \quad (\sigma, \sigma'_0), (\sigma_1, \sigma'_1), \dots, (\sigma_n, \sigma') \in \mathcal{T}[c] \\ & \quad \text{and it is Interference-Free.}\} \\ & \cup (\mathcal{B}[\neg b] ; \mathcal{T}[\mathbf{Skip}])^\omega \end{aligned}$$

Trace Semantics (cont'd)

The semantics is equivalent to the following:

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

Problem with This Semantics

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

$x := x + 1$

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

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

Also consider the following two programs:

$x := x + 1 ; x := x + 1$

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

Stuttering and Mumbling

$$\overline{\tau < \tau} \qquad \overline{\tau < (\sigma, \sigma), \tau} \qquad \overline{(\sigma, \sigma'), (\sigma', \sigma''), \tau < (\sigma, \sigma''), \tau}$$

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

$$\mathcal{T}^\dagger \stackrel{\text{def}}{=} \{\tau \mid \tau \in \mathcal{T} \text{ or } \exists \tau' \in \mathcal{T}. \tau' < \tau\}$$

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

Stuttering and Mumbling (cont'd)

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

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