Basics of Parallel Programs

CS 536: Science of Programming, Spring 2023

2023-04-06: pp 3–6

A. Why?

- Parallel programs are more flexible than sequential programs but their execution is more complicated.
- Parallel programs are harder to reason about because parts of a parallel program can interfere with other parts.
- Evaluation graphs can be used to show all possible execution paths for a parallel program.

B. Objectives

After this class, you should know

- The syntax and operational & denotational semantics of parallel programs.

C. Basic Definitions for Parallel Programs

- ***Syntax*** for parallel statements: $S := [\, S \,||\, S \,||\, \ldots \,||\, S \,]$. We say $[\, S_1 \,||\, S_2 \,||\, \ldots \,||\, S_n \,]$ is the ***parallel composition*** of the ***threads*** $S_1, S_2, \ldots, S_n$.
  - The threads must be sequential: You can't nest parallel programs. (But you can embed parallel programs within otherwise-sequential programs, such as in the body of a loop.)
- ***Example 1***: $[\, x := x + 1 \,||\, x := x * 2 \,||\, y := x^2 \,]$ is a parallel program with three threads. Since it tries to nest parallel programs, $[\, x := x + 1 \,||\, [\, x := x * 2 \,||\, y := x^2 \,] \,]$ is illegal.

Interleaving Execution of Parallel Programs

- We run sequential threads in parallel by ***interleaving*** their execution. I.e., we interleave the operational semantics steps for the individual threads.
- We execute one thread for some number of operational steps, then execute another thread, etc.
- Depending on the program and the sequence of interleaving, a program can have more than one final state (or cause an error sometimes but not other times).
- As an example, since evaluation of $[\, x := x + 1 \,||\, x := x * 2 \,]$ is done by interleaving the operational semantics steps of the two threads, we can either evaluate $x := x + 1$ and then $x := x * 2$ or evaluate $x := x * 2$ and then $x := x + 1$.
- The difference between $[\, x := x + 1 \,||\, x := x * 2 \,]$ and ***if*** $T \rightarrow x := x + 1 \;\square\; T \rightarrow x := x * 2$ ***fi*** is that the nondeterministic ***if-fi*** executes only one of the two assignments whereas the parallel composition executes both assignments but in an unpredictable order. The sequential nondeterministic ***if-fi*** that simulates the parallel assignments is ***if*** $T \rightarrow x := x + 1 \,;\, x := x * 2 \;\square\; T \rightarrow x := x * 2 \,;\, x := x + 1$

fi. It nondeterministically chooses between the two possible traces of execution for the program.[1]

- Because of the nondeterminism, re-executions of a parallel program can use different orders. For example, two executions of ***while*** B ***do*** $[\, x := x + 1 \;||\; x := x * 2 \,]$ ***od*** can have the same sequence or different sequences of updates to x.

Difficult to Predict Parallel Program Behavior

- The main problem with parallel programs is that their properties can be very different from the behaviors of the individual threads.
- ***Example 2***:
  - $\vDash \{ x = 5 \}\, x := x + 1 \,\{ x = 6 \}$ and $\vDash \{ x = 5 \}\, x := x * 2 \,\{ x = 10 \}$
  - But $\vDash \{ x = 5 \}\, [\, x := x + 1 \;||\; x := x * 2 \,]\, \{ x = 11 \vee x = 12 \}$
- The problem with reasoning about parallel programs is that different threads can ***interfere*** with each other: They can change the state in ways that don't maintain the assumptions used by other threads.
- Full interference is tricky, so we're going to work our way up to it. First we'll look at simple, limited parallel programs that don't interact at all (much less interfere).
- But before that, we need to look at the semantics of parallel programs more closely.

D. Semantics of Parallel Programs

- To execute the sequential composition $S_1 ; \ldots ; S_n$ for one step, we execute S_1 for one step.
- To execute the parallel composition $[\, S_1 \;||\; \ldots \;||\; S_n \,]$ for one step, we take one of the threads and evaluate it for one step.

Operational and Denotational Semantics of Parallel Programs

- ***Definition***: Given $[\, S_1 \;||\; \ldots \;||\; S_n \,]$, for each $k = 1, 2, \ldots, n$, if $\langle S_k, \sigma \rangle \to \langle T_k, \tau_k \rangle$, then $\langle [\, S_1 \;||\; \ldots \;||\; S_n \,], \sigma \rangle \to \langle [\, S_1 \;||\; \ldots \;||\; S_{k-1} \mid T_k \;||\; S_{k+1} \;||\; \ldots \;||\; S_n \,], \tau_k \rangle$
- We write E for sequential thread that has finished execution, so a parallel program that has finished execution is written $[\, E \;||\; \ldots \;||\; E \;||\; E \,]$. We'll treat E and $[\, E \;||\; \ldots \;||\; E \;||\; E \,]$ as being syntactically equal, i.e., $E \equiv [\, E \;||\; \ldots \;||\; E \;||\; E \,]$.

The $\to$ Notation*

- ***Notation***: The $\to$* notation has the same meaning whether the configurations involved have parallel programs or not: $\to$* means $\to^n$ for some $n \geq 0$, and $C_0 \to^n C_n$ means we've omitted writing the out intermediate configurations in the sequence $C_0 \to C_1 \to \ldots \to C_{n-1} \to C_n$ (for some collection of C.)

---

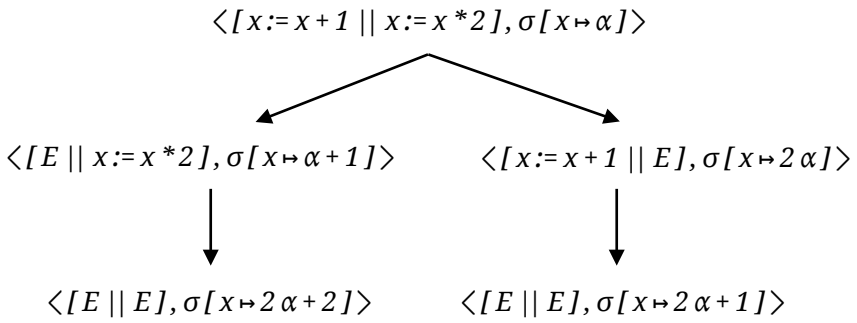
[1] This trick doesn't scale up well to larger programs, but it helps with initially understanding parallel execution.

- ***Common Mistake***: Writing $\langle [\, E \,||\, E \,], \tau \rangle \to \langle E, \tau \rangle$ is a common mistake. Since $[\, E \,||\, E \,] \equiv E$, going from $\langle [\, E \,||\, E \,], \tau \rangle$ to $\langle E, \tau \rangle$ doesn't involve an execution step. But $\langle [\, E \,||\, E \,], \tau \rangle \to^0 \langle E, \tau \rangle$ is ok because it says that in zero steps, we go from one empty configuration to itself.

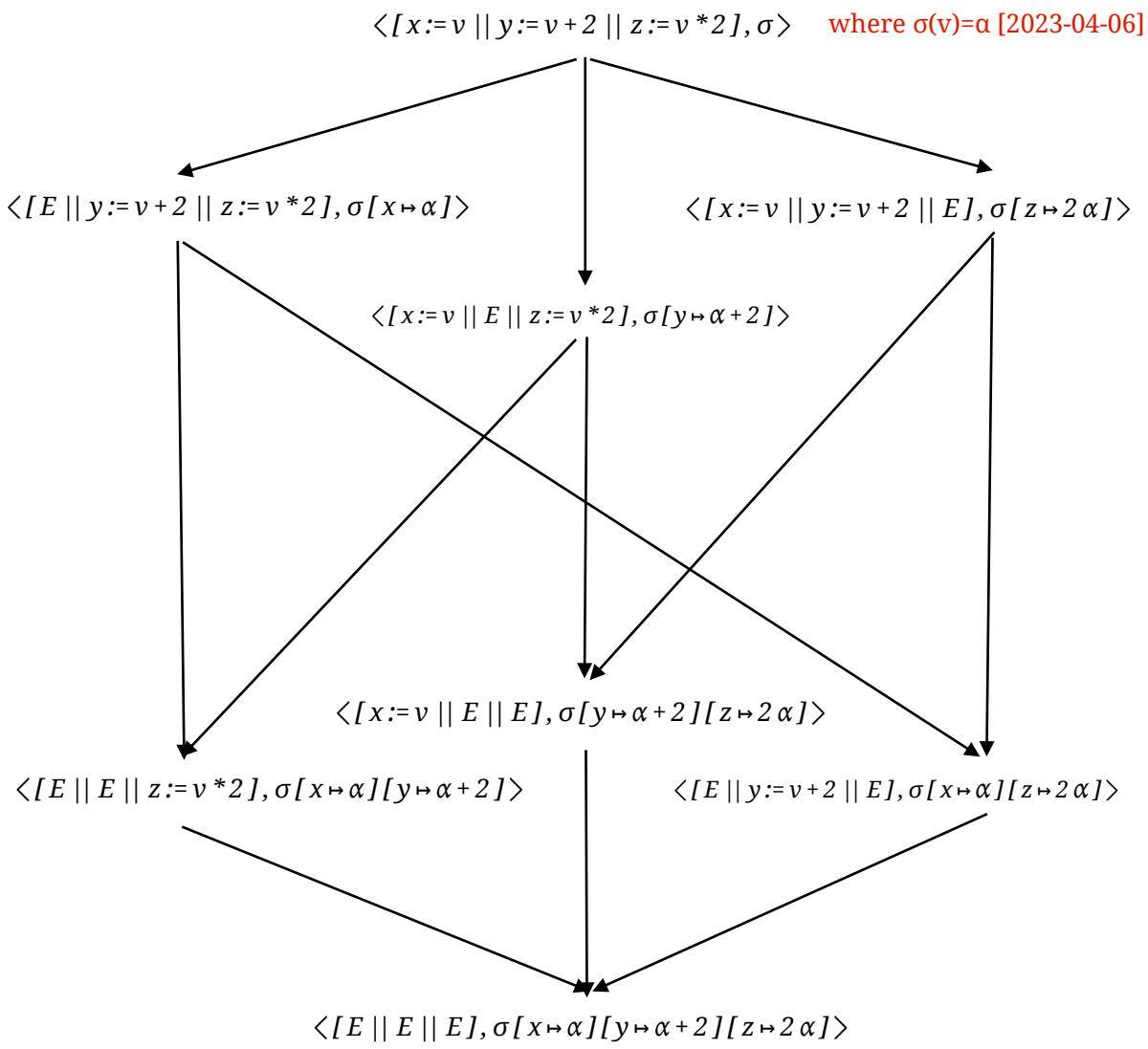
Evaluation Graph and Denotational Semantics

- Recall that the ***evaluation graph*** for $\langle S, \sigma \rangle$ is the directed graph of configurations and evaluation arrows leading from $\langle S, \sigma \rangle$.
- When drawing evaluation graphs, the configuration nodes need to be different.
  - (I.e., if the same configuration appears more than once, show multiple arrows into it — don't repeat the same node.)
- An evaluation graph shows all possible executions.
  - A program with n threads will have n out-arrows from its configuration.
  - (Exception: Evaluation graphs are not multigraphs: If two arrows go to exactly the same configuration, we write the configuration just once and write exactly one arrow to it.)
- A path through the graph corresponds to one possible evaluation of the program.
- The ***denotational semantics*** of a program in a state is the set of all possible terminating states (plus possibly the pseudostates $\perp_d$ and $\perp_e$). I.e., the states found in the sinks (i.e., at the leaves) of an evaluation graph. (We'll modify this definition when we get to deadlocked programs.)
  - $M(S, \sigma) = \{ \tau \in \sigma \mid \langle S, \sigma \rangle \to^* \langle E, \tau \rangle \}$
    $\cup \{ \perp_d \}$ if S can diverge; i.e., if $\langle S, \sigma \rangle \to^* \langle E, \perp_d \rangle$ is possible [2023-04-06]
    $\cup \{ \perp_e \}$ if S can produce a runtime error; i.e., $\langle S, \sigma \rangle \to^* \langle E, \perp_e \rangle$ is possible. [2023-04-06]

- ***Example 3***: The evaluation graph below is for the same program as in Example 2, but starting with an arbitrary state σ where $\sigma(x) = \alpha$. The graph has two sinks for the two possible final states, so $M([\, x := x+1 \,||\, x := x*2 \,], \sigma) = \{ \sigma[x \mapsto 2\alpha+2], \sigma[x \mapsto 2\alpha+1] \}$.



Example 3

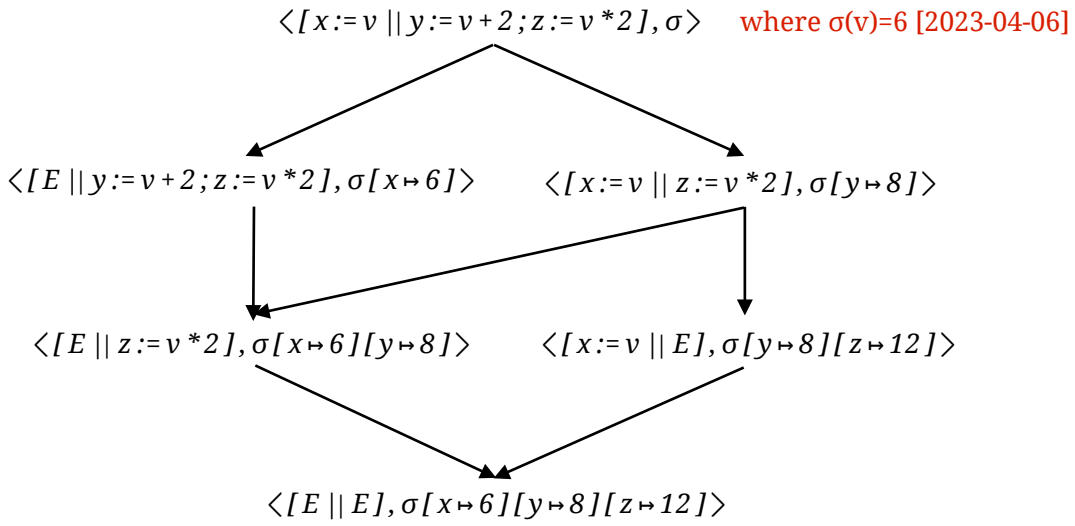


Example 4

- ***Example 4***: For this example, the evaluation graph is for $\langle [\, x := v \,||\, y := v + 2 \,||\, z := v * 2 \,], \sigma \rangle$, where $\sigma(v) = \alpha$. $M([\, x := v \,||\, y := v + 2 \,||\, z := v * 2 \,], \sigma) = \{\sigma[\, x \mapsto \alpha \,][\, y \mapsto \alpha + 2 \,][\, z \mapsto 2\,\alpha \,]\}$. Note even though the program is nondeterministic, it produces the same result no matter what execution path it uses.

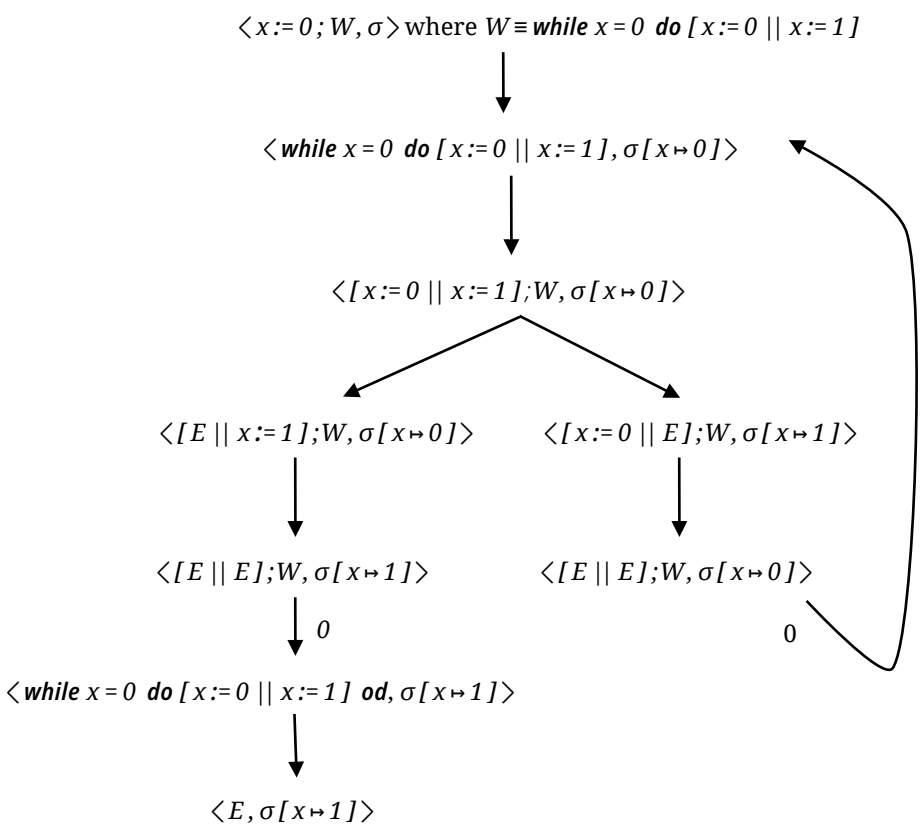
  (More generally, if S is parallel, then $M(S, \sigma)$ can have more than 1 member, but the converse is not true: Having $M(S, \sigma)$ of size 1 does not imply that S is nondeterministic.)

- ***Example 5***: If we take the program from Example 4 and combine the last two threads sequentially, then the evaluation graph for the resulting program is a subgraph of the graph from Example 4. Below, $\sigma(v) = 6$, and $M([x := v \,||\, y := v + 2 \,||\, z := v * 2], \sigma) = \{\sigma[x \mapsto 6][y \mapsto 8][z \mapsto 12]\}$.



Example 5

- ***Example 6***: Let $W \equiv x := 0;$ ***while*** $x = 0$ ***do*** $[x := 0 \,||\, x := 1]$ ***od***. Then $M(W, \sigma) = \{\sigma[x \mapsto 1], \perp_d\}$. as shown in the evaluation graph. Note the transitions $\langle [E \,||\, E]; W, \sigma[x \mapsto \ldots] \rangle \rightarrow^0 \langle W, \sigma[x \mapsto \ldots] \rangle$ take 0 steps because $[E \,||\, E]; W \equiv E; W \equiv W$; that is, they're all the same program, textually.
- The problem in this example is that there is possible divergence.
  - On the other hand, it only happens if we ***always*** choose thread 1 when we have to make the nondeterministic choice of $[x := 0 \,||\, x := 1]$.
  - This is definitely unfair behavior, but it's allowed because of the unpredictability of our nondeterministic choices. In real life, we would want a fairness mechanism to ensure that all threads get to evaluate once in a while.
- If each thread is on a separate processor, then the nondeterministic choice corresponds to which processor is fastest, so the possible divergence of the program is a ***race condition***, where the correct behavior of a program depends on the relative speed of the processors involved. Here, divergence occurs if the processor for x:=1 is always faster than the processor for x:=0. [2023-04-06]
- Note that it's not necessarily a race condition to have a parallel program producing different results when run multiple times. As long as all results satisfy the specification, there's no race condition.



Example 6

- ***Example 7***: The correctness triple *{ T } [x := 0 || x := 1] { x ≥ 0 }* does not have a race condition, but *{ T } [x := 0 || x := 1] { x > 0 }* does. [2023-04-06] The program terminates with *x = 0* or *1*. With postcondition *x ≥ 0* , both states are correct even though they're different. But with post-condition *x > 0* , the relative speed of the threads means we may or may not produce a correct result.