Basics of Parallel Programs

Stefan Muller, based on material by Jim Sasaki

CS 536: Science of Programming, Spring 2022
Lecture 20

1 Definitions

Syntax:

$$s ::= \ldots \mid [\ s \parallel \cdots \parallel s\ ]$$

We say $[\ s_1 \parallel \cdots \parallel s_n\ ]$ 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[1]. (You can embed parallel programs within larger programs, such as in the body of a loop.)

Examples.

- Valid: $[\ x := x + \overline{1} \parallel x := x * \overline{2} \parallel y := x^{\overline{2}}\ ]$ is a parallel program with 3 threads.
- Invalid: $[\ x := x + \overline{1} \parallel [\ x := x * \overline{2} \parallel y := x^{\overline{2}}\ ]\ ]$ tries to nest parallel programs.
- Valid: $[\ x := x + \overline{1} \parallel x := x * \overline{2}\ ]; [\ x := x + \overline{1} \parallel y := x^{\overline{2}}\ ]$ has two threads that then join back together and split into 2 threads again.

1.1 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 $[\ x := x + \overline{1} \parallel x := x * \overline{2}\ ]$ is done by interleaving the operational semantics steps of the two threads, we can either evaluate $x := x + \overline{1}$ and then $x := x * \overline{2}$ or vice versa[2]
- The difference between $[\ x := x + \overline{1} \parallel x := x * \overline{2}\ ]$ and $\mathsf{branch}\ \{T \to x := x + \overline{1} \,\square\, T \to x := x * \overline{2}\}$ is that the nondeterministic branch executes only one of the two assignments whereas the parallel composition executes both assignments but in an unpredictable order. The sequential nondeterministic branch that simulates the parallel assignments is $\mathsf{branch}\ \{T \to x := x + \overline{1}; x := x * \overline{2} \,\square\, T \to x := x * \overline{2}; x := x + \overline{1}\}$. It nondeterministically chooses between the two possible traces of execution for the program[3]
- Because of the nondeterminism, re-executions of a parallel program can use different orders. For example, two executions of $\mathsf{while}\ e\ \{[\ x := x + \overline{1} \parallel x := x * \overline{2}\ ]\}$ can have the same sequence or different sequences of updates to x.

[1] Some languages, including several that Stefan works with, allow you to do this; we might look at that later, but for now we'll say we can't nest.

[2] If you're familiar with writing concurrent programs, you might also know that, if we were using, e.g., C, it would be possible to calculate both $x + 1$ and $x * 2$ before we store either, and have one overwrite the other, ending up with either $x + 1$ or $2x$ rather than $2x + 1$ or $2(x + 1)$. In this class, we'll assume that we interleave at the level of operational steps, so calculating the expression and doing the assignment happens atomically (all at once).

[3] This is a good use of sequential nondeterminism to understand the nondeterminism that results from parallel interleaving; unfortunately, it doesn't scale well to large programs.

1.2 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:

$$\vDash \{x = 5\}\ x := x + \overline{1}\ \{x = 6\} \text{and} \vDash \{x = 5\}\ x := x * \overline{2}\ \{x = 10\}$$

but

$$\vDash \{x = 5\}\ [\ x := x + \overline{1} \parallel x := x * \overline{2}\ ]\ \{x = 11 \vee x = 12\}$$

- 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.

2 Semantics of Parallel Programs

2.1 Small-step Semantics

To execute the sequential composition $s_1; \ldots; s_n$ for one step, we execute s_1 for one step (unless it is skip). To execute the parallel composition $[\ s_1 \parallel \cdots \parallel s_n\ ]$ for one step, we take one of the threads and evaluate it for one step.

$$\frac{\langle s_k, \sigma\rangle \to \langle s_k', \sigma_k\rangle}{\langle[\ s_1 \parallel \cdots \parallel s_n\ ], \sigma\rangle \to \langle[\ s_1 \parallel \cdots \parallel s_{k-1} \parallel s_k' \parallel s_{k+1} \parallel \cdots \parallel s_n\ ], \sigma_k\rangle}$$

A completely-executed parallel program looks like $[\ \mathsf{skip} \| \cdots \| \mathsf{skip} \| \mathsf{skip}\ ]$. We'll treat $[\ \mathsf{skip} \| \cdots \| \mathsf{skip} \| \mathsf{skip}\ ] \equiv \mathsf{skip}$.

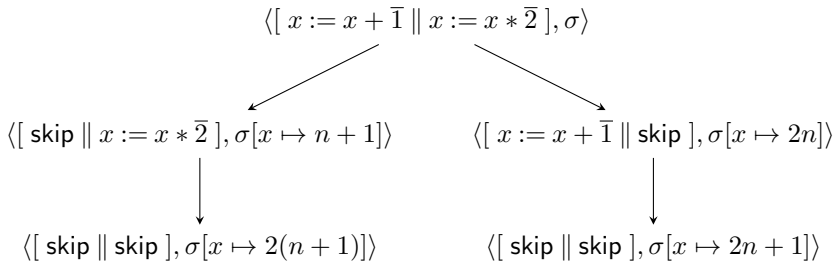
- Notation: The $\to^*$ notation has the same meaning whether the configurations involved have parallel programs or not: $\to^*$ means $\to^n$ for some $n \geq 0$, and $c_0 \to^n c_n$ means that there is actually a sequence of n+1 configurations, $c_0 \to c_1 \to \ldots \to c_n$ where we've omitted writing the intermediate configurations.
- Common Mistake: Writing $\langle[\ \mathsf{skip} \parallel \mathsf{skip}\ ], \sigma\rangle \to \langle \mathsf{skip}, \sigma\rangle$. The finished parallel program doesn't take a step. With parallel programs, since $[\ \mathsf{skip} \parallel \mathsf{skip}\ ] \equiv \mathsf{skip}$, we can write $\langle[\ \mathsf{skip} \parallel \mathsf{skip}\ ]\sigma, \to\rangle^0\langle \mathsf{skip}, \sigma\rangle$.

2.2 Evaluation Graph and Big-step Semantics

- The evaluation graph for $\langle s, \sigma\rangle$ is the directed graph of configurations and evaluation arrows leading from $\langle s, \sigma\rangle$.
  - The nodes are configurations
  - An edge from c_0 to c_1 means that $c_0 \to c_1$.
  - 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.
- A path through the graph corresponds to an 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 $\perp_e$ and $\perp_d$). These are the states found at the *sinks* (the nodes with no outgoing edges). (We'll modify this definition when we get to deadlocked programs.)

$$
\begin{array}{rcll}
M(s,\sigma) & = & \{\sigma' \mid \langle s,\sigma\rangle \to^* \langle \mathsf{skip},\sigma'\rangle\} & \\
& \cup & \{\bot_d\} & \text{if } s \text{ can diverge} \\
& \cup & \{\bot_e\} & \text{if } s \text{ can produce a runtime error}
\end{array}
$$

Example. The evaluation graph below is for our running example program, starting with a state σ where $\sigma(x) = n$. The graph has two sinks (possible final states), so $M([\ x := x + \overline{1} \parallel x := x * \overline{2}\ ], \sigma) = \{\sigma[x \mapsto 2n+2], \sigma[x \mapsto 2n+1]\}$.



Example The evaluation graph in Figure 1 is for $\langle[\ x := v \parallel y := v + \overline{2} \parallel z := v * \overline{2}\ ], \sigma\rangle$ where $\sigma(v) = n$. We have $M([\ x := v \parallel y := v + \overline{2} \parallel z := v * \overline{2}\ ], \sigma) = \sigma[x \mapsto n][y \mapsto n+2][z \mapsto 2n]$. Note that even though the program is nondeterministic, it produces the same result regardless of the execution path. Just like with sequential nondeterministic programs, if s is parallel, then $M(s,\sigma)$ *can* have more than one element, but may not.

The reason we end up with one state is that all of the state updates *commute*, i.e., $\sigma[x \mapsto n][y \mapsto n+2] = \sigma[y \mapsto n+2][x \mapsto n]$, and so on. This isn't an accident; it's because the threads are updating different variables and they read variables that no other thread writes. This guarantees all of the updates commute, and the program is deterministic, and is therefore a really nice property for parallel programs to have!

Example Let $W = x := \overline{0}; \mathsf{while}\ x = \overline{0}\ \{[\ x := \overline{0} \parallel x := \overline{1}\ ]\}$. Then $M(W,\sigma) = \{\sigma[x \mapsto 1], \bot_d\}$. The problem here is possible divergence, but it only happens if we always choose thread 1 when we have to make the nondeterministic choice. 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.

If the behavior of a program depends on the relative speed of the processors involved, like the termination of this example does, we call this a *race condition*. Generally, this happens when two or more threads access the same variable and at least one of the accesses is a write (like the two threads writing to x in this example). In the previous example, there was no race condition: one thread each writes to x, y and z; all threads read v, but that's fine. People disagree on the exact definition of the term *race condition*; we may or may not go into more detail on this.

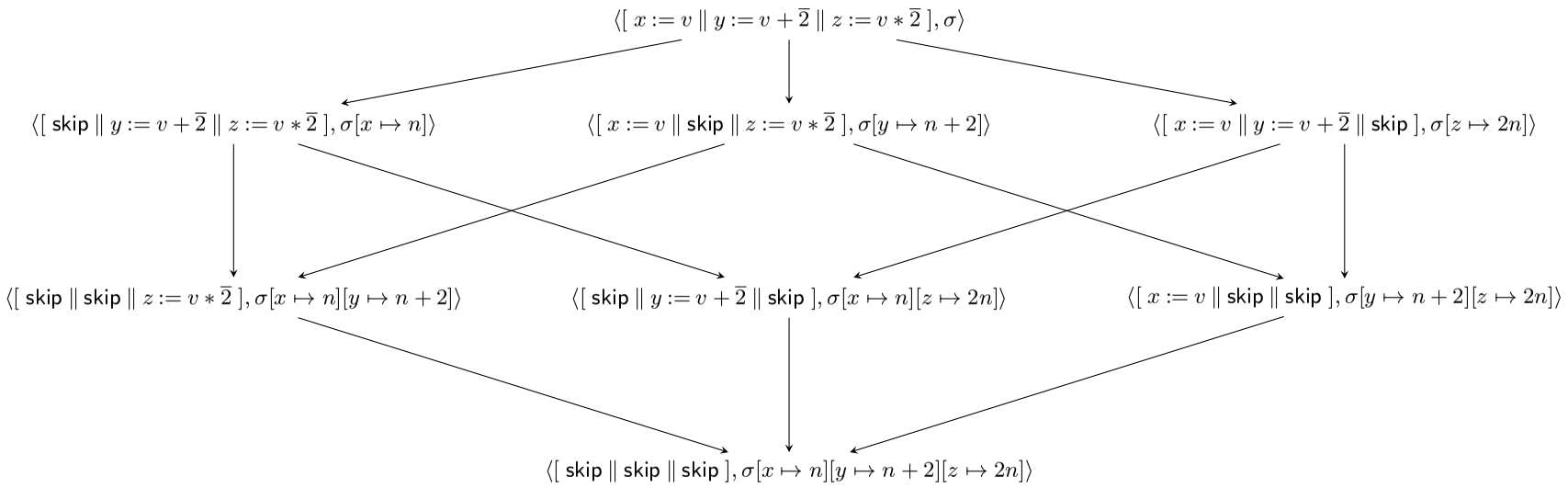


Figure 1: Evaluation graph for example.