Basics of Parallel Programs
CS 536: Science of Programming, Fall 2020

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.
- Reducing the amount of interference between threads lets us reason about parallel programs by combining the proofs of the individual threads.
- Disjoint parallel programs ensure that no thread can interfere with the execution of another thread.
- Disjoint conditions ensure that no thread can interfere with the conditions of a triple.
- Disjoint parallel programs with disjoint conditions can be proved correct by combining the proofs of their individual threads.

B. Objectives

After this class, you should know

- The syntax and operational & denotational semantics of parallel programs
- What the interference problem is.
- What disjoint parallel programs and disjoint conditions are.
- What the disjoint parallelism rule for disjoint parallel programs with disjoint conditions allows.

C. Basic Definitions for Parallel Programs

- Syntax for parallel statements: \( S := [S_1 || S_2 || \ldots || S_n] \). 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. (You can embed parallel programs within larger programs, such as in the body of a loop.)
  - Example 1: \([x := x+1 || x := x*2]\) is a parallel program with two threads.

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 \([x := x + 1 \mid x := x \times 2]\) is done by interleaving the operational semantics steps of the two threads, we can either evaluate \(x := x + 1\) and then \(x := x \times 2\) or evaluate \(x := x \times 2\) and then \(x := x + 1\).

- The choice of which thread to execute is nondeterministic, so re-execution of a parallel program doesn’t have to use the same order. For example, if we run

  \[
  \text{while } B \text{ do } [x := x + 1 \mid x := x \times 2] \text{ od}
  \]

  multiple times, there’s no guarantee that any two executions will have the same sequence of updates to \(x\). (For that matter, there’s no guarantee that any two executions will have different orders of updates.)

**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:**
  
  - \(\models \{x = 5\} x := x + 1 \{x = 6\}\) and \(\models \{x = 5\} x := x \times 2 \{x = 10\}\)
  
  - But \(\models \{x = 5\} [x := x + 1 \mid x := x \times 2] \{x = 11 \lor x = 12\}\)

- 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 \([S_1 \mid \ldots \mid S_n]\) for one step, we take one of the threads and evaluate it for one step.

**Operational and Denotational Semantics of Parallel Programs**

- **Definition:** For \([S_1 \mid \ldots \mid S_n]\), then for each \(k = 1, 2, \ldots, n\), if \(\langle S_k, \sigma \rangle \rightarrow \langle T_k, \tau_k \rangle\), then

\[
\langle [S_1 \mid \ldots \mid S_n], \sigma \rangle \rightarrow \langle [S_1 \mid \ldots \mid S_{k-1} \mid T_k \mid S_{k+1} \mid \ldots \mid S_n], \tau_k \rangle \quad [\text{for each } k]
\]

- We’ll write \(E\) for the completely-executed program, so a completely-executed parallel program looks like \([E \mid \ldots \mid E \mid E]\). For consistency’s sake we’ll treat \([E \mid \ldots \mid E \mid E]\) as \(=\) to \(E\).

- We can’t say \(\langle E, \tau \rangle \rightarrow \) anything in a positive number of steps. All we can have is \(\langle E, \tau \rangle \rightarrow^0 \langle E, \tau \rangle\).

(In zero steps, \(\langle E, \tau \rangle\) leads to \(\langle E, \tau \rangle\).)
In particular, \( \langle [E | | E], \tau \rangle = \langle [E | | E], \tau \rangle \) so \( \langle [E | | E], \tau \rangle \rightarrow^0 \langle E, \tau \rangle \) is incorrect. You can write \( \langle [E | | E], \tau \rangle \rightarrow^0 \langle E, \tau \rangle \), however.

- 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.
- 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 \( \bot_e \) and \( \bot_d \)).
  - \( M(S, \sigma) = \{ \tau \in \Sigma \mid \langle S, \sigma \rangle \rightarrow^* \langle E, \tau \rangle \} \)
  - \( \cup \{ \bot_d \} \) if \( S \) can diverge; i.e., if \( \langle S, \sigma \rangle \rightarrow^* \langle \bot_d, \tau \rangle \) is possible
  - \( \cup \{ \bot_e \} \) if \( S \) can produce a runtime error; i.e., \( \langle S, \sigma \rangle \rightarrow^* \langle \bot_e, \tau \rangle \) is possible

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

```plaintext
let \( \alpha = \sigma(x) \)
```
**Example 4:** Here is the evaluation graph for \( \langle [x := v \mid y := v+2 \mid z := v+2], \sigma \rangle \), where \( \sigma(v) = \alpha \).

\[
M([x := v \mid y := v+2 \mid z := v+2], \sigma) = \{\sigma[x \mapsto \alpha][y \mapsto \alpha+2][z \mapsto 2\alpha]\}\.
\]

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.
• **Example 5**: If we keep the program from Example 4 but start evaluation in a different state, then the evaluation graph can be different. Here, $\sigma(v) = 6$, and the evaluation graph is a subgraph of the general graph in Example 4. This time, $M([x := v \mid \mid y := v+2 \mid \mid z := v*2], \sigma) = \{\sigma[x \mapsto 6][y \mapsto 8][z \mapsto 12]\}$.

\[\langle \ldots \rangle\]

• **Example 6**: If $W = x := 0; \textbf{while } x = 0 \textbf{ do } [x := 0 \mid \mid x := 1] \textbf{ od}$, then $M(W, \sigma) = \{\sigma[x \mapsto 1], \bot\}$

\[\langle \ldots \rangle\]