Semantics of Parallel Programs

- **Operational semantic** of parallel statements: given $S \equiv [S_1 \parallel S_2 \parallel \cdots \parallel S_n]$, for each $k = 1, 2, \ldots, n$, if $\langle S_k, \sigma\rangle \to \langle T_k, \tau_k\rangle$, then $\langle [S_1 \parallel \cdots \parallel S_{k-1} \parallel S_k \parallel S_{k+1} \parallel \cdots \parallel S_n],\ \sigma\rangle \to \langle [S_1 \parallel \cdots \parallel S_{k-1} \parallel T_k \parallel S_{k+1} \parallel \cdots \parallel\ S_n]\ , \tau_k\rangle$. If we don't have any runtime error or divergence, the execution of S will end with configuration $\langle E \equiv [E \parallel E \parallel \cdots \parallel E]\ , \tau\rangle$.
  - Note that, from each configuration, we can go to at most n configurations in the next step.
  - The notations $\to^*$ and $\to^k$ that we used in a non-parallel program still work here.
  - Note that, $E \equiv [E \parallel E \parallel \cdots \parallel E]$. It is a common mistake to write $\langle [E \parallel E \parallel \cdots \parallel E]\ , \tau\rangle \to \langle E, \tau\rangle$; we can write $\langle [E \parallel E \parallel \cdots \parallel E]\ , \tau\rangle\ \to^0\ \langle E, \tau\rangle$ since $E \equiv [E \parallel E \parallel \cdots \parallel E]$.

- Remember that **denotational semantics** of a statement in a state is the collection of all possible terminating states (plus possibly the pseudo states $\perp_d$ and $\perp_e$). It is the same for parallel statements.

1. Show the operational sematic for $\langle S, \sigma\rangle$ till the end where $S \equiv [x \coloneqq x + 1 \parallel x \coloneqq x * 2]$ and $\sigma = \{x = 5\}$. What is $M(S, \sigma)$?

$$\begin{tikzcd}
& \langle [x \coloneqq x + 1 \parallel x \coloneqq x * 2\ ], \{x = 5\}\rangle \arrow[dl] \arrow[dr] & \\
\langle [E \parallel x \coloneqq x * 2], \{x = 6\}\rangle \arrow[d] & & \langle [x \coloneqq x + 1 \parallel E], \{x = 10\}\rangle \arrow[d] \\
\langle [E \parallel E], \{x = 12\}\rangle & & \langle [E \parallel E], \{x = 11\}\rangle
\end{tikzcd}$$

   - $M(S, \sigma) = \big\{\{x = 12\}, \{x = 11\}\big\}$. Since parallel statements can be considered as a simulation of nondeterminism, it is still true that "If $M(S, \sigma)$ contains more than one states, then S is nondeterministic."

- In a parallel program, at each configuration there is usually more than one configuration that can be the next step, so the analysis of operational semantics usually gives directed graph instead of a list, we call this graph an **evaluation graph**.
  - While drawing an evaluation graph, we need to make sure that:
    1) Each vertex in the graph is a configuration and each configuration is *unique*.
    2) Each directed edge shows one step (or n steps if $\to^n$, or any number steps if $\to^*$) in the evaluation, and we don't allow multi-edge in the graph. In other words, if we go from one configuration to another twice, we only draw this edge once in the graph.
    3) All the possible executions need to be shown in the graph: if a thread composition has n threads, then there can be at most n outcoming edges from that node.

Let's look at an example with a loop.

2. Let $W \equiv \textbf{while}\ x = 0\ \textbf{do}\ [x \coloneqq 0 \parallel x \coloneqq 1]\ \textbf{od}$. Draw evaluation graph for $\langle x \coloneqq 0; W, \sigma\rangle$, and calculate $M(x \coloneqq 0; W, \sigma)$.

$$
\begin{tikzcd}
 & \langle x \coloneqq 0;\ \mathbf{while}\ x = 0\ \mathbf{do}\ [x \coloneqq 0 \parallel x \coloneqq 1]\ \mathbf{od}, \sigma\rangle \arrow[d] & \\
 & \langle \mathbf{while}\ x = 0\ \mathbf{do}\ [x \coloneqq 0 \parallel x \coloneqq 1]\ \mathbf{od}, \sigma[x \mapsto 0]\rangle \arrow[d] & \\
 & \langle\, [x \coloneqq 0 \parallel x \coloneqq 1]; W, \sigma[x \mapsto 0]\rangle \arrow[dl] \arrow[dr] & \\
\langle\, [E \parallel x \coloneqq 1]; W, \sigma[x \mapsto 0]\rangle \arrow[d] & & \langle [x \coloneqq 0 \parallel E]; W, \sigma[x \mapsto 1]\rangle \arrow[d] \arrow[uul, bend right] \\
\langle\, [E \parallel E]; W, \sigma[x \mapsto 1]\rangle \arrow[d] \arrow[dd, bend right] & & \langle [E \parallel E]; W, \sigma[x \mapsto 0]\rangle \\
\langle W, \sigma[x \mapsto 1]\rangle \arrow[d] & & \\
\langle E, \sigma[x \mapsto 1]\rangle & &
\end{tikzcd}
$$

- From the evaluation graph, we can see that $M(S, \sigma) = \{\bot_d, \sigma[x \mapsto 1]\}$

- We can see that W might diverge on σ: in each iteration if we execute $x \coloneqq 1$ first, then we will end up with $x = 0$ and we will go to another iteration.
- Let program S contains parallel statement $[S_1 \parallel S_2 \parallel \cdots \parallel S_n]$ and $\not\models_{tot} \{p\}\, S\, \{q\}$; if this invalidity is caused by the *execution order* (*or relative speed*) of threads in $[S_1 \parallel S_2 \parallel \cdots \parallel S_n]$, then we say S has **race conditions**.

3. Does each of the following triples have race conditions?
   a) $\{T\}[\, x \coloneqq 0 \parallel x \coloneqq 1]\, \{x > 0\}$
   Yes. We can get $x = 0$ if we execute the first thread after the second thread, it doesn't satisfy the postcondition.

   b) $\{T\}[\, x \coloneqq 0 \parallel x \coloneqq 1\,]\, \{x \geq 0\}$
   No. The program can end up with $x = 0$ or 1, they both satisfy the postcondition.

   c) $\{T\}[\, x \coloneqq 0 \parallel x \coloneqq 1\,];\ z \coloneqq 0 \div x\, \{z = 0\}$
   Yes. If we execute the second thread first then we will end up with a runtime error.

   d) $\{T\}[\, x \coloneqq 0 \parallel x \coloneqq 1\,]\, \{x > 2\}$
   No. The execution order is not the reason for the invalidity.

Disjoint Parallel Programs

4. Draw the evaluation graph for configuration $\langle [x \coloneqq v \parallel y \coloneqq v * 2 \parallel z \coloneqq v + 1],\ \sigma \rangle$ where $\sigma(v) = \alpha$, and v, x, y, z are different variables.

$$\langle [x \coloneqq v \parallel y \coloneqq v * 2 \parallel z \coloneqq v + 1], \sigma \rangle$$

$$\langle [E \parallel y \coloneqq v * 2 \parallel z \coloneqq v + 1],\ \sigma[x \mapsto \alpha] \rangle \qquad \langle [x \coloneqq v \parallel\ y \coloneqq v * 2 \parallel E], \sigma[z \mapsto \alpha + 1] \rangle$$

$$\langle [x \coloneqq v \parallel E \parallel z \coloneqq v + 1], \sigma[y \mapsto 2\alpha] \rangle$$

$$\langle [E \parallel E \parallel z \coloneqq v + 1],\ \sigma[x \mapsto \alpha][y \mapsto 2\alpha] \rangle \qquad \langle [E \parallel y \coloneqq v * 2 \parallel E], \sigma[x \mapsto \alpha][z \mapsto \alpha + 1] \rangle$$

$$\langle [x \coloneqq v \parallel E \parallel E], \sigma[y \mapsto 2\alpha][z \mapsto \alpha + 1] \rangle$$

$$\langle [E \parallel E \parallel E], \sigma[x \mapsto \alpha][y \mapsto 2\alpha][z \mapsto \alpha + 1] \rangle$$

- o In this example, although the program is parallel, it generates the same state no matter what execution paths it uses.

- The program in the above example is a **disjoint parallel program**. Disjoint parallel programs model the situation that multiple threads share readable memory but not writable memory. For every variable x in a disjoint parallel program, there are two situations:
  a) All threads can read x and no threads can write x. In example 4, every thread reads v and no thread writes v.
  b) At most 1 thread can read and write x, and other threads can neither read nor write x. In example 4, the first thread writes x so other threads can neither read nor write x.

- In general, with $[S_1 \parallel S_2]$ we can execute either S_1 or S_2 for the next step. In an evaluation graph, the current evaluation path splits into two paths. In general, there might be no way for those two paths to eventually merge back together into one path, but disjoint parallel programs are different.

- **[Diamond Property]** Let $[S_1 \parallel S_2]$ be a disjoint parallel program. If $\langle S_1, \sigma \rangle \rightarrow \langle T_1, \tau_1 \rangle$ and $\langle S_2, \sigma \rangle \rightarrow \langle T_2, \tau_2 \rangle$ then there is a state τ such that $\langle [T_1 \parallel S_2], \tau_1 \rangle$ and $\langle [S_1 \parallel T_2], \tau_2 \rangle$ both $\rightarrow \langle [T_1 \parallel T_2], \tau \rangle$.
  - o It is easy to see the property is true since the order of execution of S_1 and S_2 does not matter: any change in state caused by the S_1 will be the same despite whether we execute S_2, since S_1 and S_2 write into different variables and they don't read the variables the other thread writes (and vice-versa).
  - o It is also easy to see how the property is named if we draw the evaluation graph. Here I use different colors on directed edges: it means if red edges exist, then the green edges exist.

$$
\begin{tikzcd}
& \langle [S_1 \parallel S_2], \sigma \rangle \arrow[dl] \arrow[dr] & \\
\langle [T_1 \parallel S_2], \tau_1 \rangle \arrow[dr] & & \langle [S_1 \parallel T_2], \tau_2 \rangle \arrow[dl] \\
& \langle [T_1 \parallel T_2], \tau \rangle &
\end{tikzcd}
$$

- Diamond property can be generalized into a more general property called **confluence**: where the one-step arrows are replaced by any-step arrows ($\rightarrow$ becomes $\rightarrow^*$). Similarly, if red edges exist, then green edges exist.

$$
\begin{tikzcd}
& \langle [\underline{S_1} \parallel S_2], \sigma \rangle \arrow[dl, "*"'] \arrow[dr, "*"] & \\
\langle [\underset{E}{T_1} \parallel S_2], \tau_1 \rangle \arrow[dr, "*"'] & & \langle [S_1 \parallel \underset{E}{T_2}], \tau_2 \rangle \arrow[dl, "*"] \\
& \langle [\underset{E}{T_1} \parallel \underset{E}{T_2}], \tau \rangle &
\end{tikzcd}
$$

- **[Confluence]** Let $[S_1 \parallel S_2]$ be a disjoint parallel program If $\langle S_1, \sigma \rangle \rightarrow^* \langle T_1, \tau_1 \rangle$ and $\langle S_2, \sigma \rangle \rightarrow^* \langle T_2, \tau_2 \rangle$ then there is a state τ such that $\langle [T_1 \parallel S_2], \tau_1 \rangle$ and $\langle [S_1 \parallel T_2], \tau_2 \rangle$ both $\rightarrow^* \langle [T_1 \parallel T_2], \tau \rangle$.
  - It is easy to see that the diamond property implies confluence (confluence is a generalized diamond property).

- Because execution of disjoint parallel programs is confluent, if execution terminates, it terminates in a unique state. This gives us the following theorem.
- **[Theorem (Unique Result of Disjoint Parallel Program)]:** If S is a disjoint parallel program then $M(S, \sigma) = \{\tau\}$, where $\tau \in \Sigma \cup \{\bot_d, \bot_e\}$.
  - It is easy to prove this theorem using confluence: If $\langle S_1, \sigma \rangle \rightarrow^* \langle E, \tau_1 \rangle$ and $\langle S_2, \sigma \rangle \rightarrow^* \langle E, \tau_2 \rangle$ then there is a state τ such that $\langle [E \parallel S_2], \tau_1 \rangle$ and $\langle [S_1 \parallel E], \tau_2 \rangle$ both $\rightarrow^* \langle [E \parallel E], \tau \rangle$.

The above theorem shows a disjoint parallel program does not have race conditions. How to recognize a disjoint parallel program?

- For statement S, we define that:
  - $vars(S) =$ the set of variables in S. (We either read or write these variables in S)
  - $change(S) =$ the set of variables appears on the left-hand side of assignments in S. (We write these variables in S)

  We say thread S_i **interferes** with S_j if $change(S_i) \cap vars(S_j) \neq \emptyset$.
  We say threads S_i and S_j are **disjoint**, if $change(S_i) \cap vars(S_j) = change(S_j) \cap vars(S_i) = \emptyset$.

- If for $0 < i \neq j \leq n$, S_i and S_j are disjoint, then we say threads $S_1, S_2, \ldots, S_n$ are **pairwise disjoint**, and we say $[S_1 \parallel S_2 \parallel \cdots \parallel S_n]$ is a **disjoint parallel composition**. We also say $[S_1 \parallel S_2 \parallel \cdots \parallel S_n]$ is a **disjoint parallel program**.

5. Determine whether the following threads are disjoint.
   a) $S_1 \equiv a := a + x$ and $S_2 \equiv y := y + x$
      Yes. $vars(S_1) = \{a, x\}$ and $change(S_2) = \{y\}$, then $vars(S_1) \cap change(S_2) = \emptyset$; $changes(S_1) = \{a\}$ and $vars(S_2) = \{y, x\}$, then $vars(S_2) \cap change(S_1) = \emptyset$.

   b) $S_1 \equiv a := x$ and $S_2 \equiv x := c$

No, $vars(S_1) = \{a, x\}$ and $change(S_2) = \{x\}$, then $change(S_2) \cap vars(S_1) \neq \emptyset$; thus S_2 interferes with S_1.

c) $S_1 \equiv x := a + 1$ and $S_2 \equiv x := b + 1$
No, both S_1 and S_2 write x, so they interfere with each other.

6. Is the program $S \equiv [S_1 \parallel S_2 \parallel S_3]$ a disjoint parallel program? Here, we have
$S_1 \equiv a := v;\ v := v + c$
$S_2 \equiv \textbf{if } b > 0 \textbf{ then } b := b * 2 \textbf{ else } c := c * 2 \textbf{ fi}$
$S_3 \equiv \textbf{while } d \geq 0 \textbf{ do } d := d \div 2 - c \textbf{ od}$

   - We can come up with the following table:

i	j	$vars(S_i)$	$changes(S_j)$	S_j interferes with S_i?
1	2	a, v, c	b, c	Yes
2	1	b, c	a, v	No
1	3	a, v, c	d	No
3	1	d, c	a, v	No
2	3	b, c	d	No
3	2	d, c	b, c	Yes

To sum up, S_2 interferes with both S_1 and S_3, thus the program S is not disjoint parallel.

- In the above example, we can see that S_2 only interferes with S_1, S_3 if we execute the false branch: If $b > 0$ at the runtime, these threads will not interfere with each other.
- In fact, this "disjointedness test" we use here is static and can be too strict, it can over-estimate the interference:
   - If our test shows that each pair of threads are disjoint, then the program is definitely a disjoint parallel program.
   - If our test shows that each thread can interfere with other threads, then it only means that this thread **might** interfere with other threads while executing.

Inference Rules for Disjoint Parallel Programs

Due to the Unique Result of Disjoint Parallel Program Theorem, we can come up with the following rule:

- **Sequentialization Rule**
If $S_1, S_2, \dots, S_n$ are pairwise disjoint:
   1. $\{p\}\ S_1; S_2; \dots; S_n\ \{q\}$
   2. $\{p\}\ [S_1 \parallel S_2 \parallel \cdots \parallel S_n]\ \{q\}$ sequentialization 1

   - We call the **sequentialization** of the parallel statement $[S_1 \parallel S_2 \parallel \cdots \parallel S_n]$ (not necessarily disjoint parallel) is the sequence statement $S_1; S_2; \dots; S_n$. The **sequentialized execution** of the parallel statement is the execution of its sequentialization: We evaluate S_1 completely, then S_2 completely, and so on.
   - In disjoint parallel programs, the execution order of threads do not matter. Thus, while using the sequentialization rule in proof, we can write in the following way as well:
   1. $\{p\}\ S_2; S_1\ \{q\}$
   2. $\{p\}\ [S_1 \parallel S_2]\ \{q\}$ sequentialization 1

7. Give a formal proof of $\{T\}\ [a \coloneqq x + 1 \parallel b \coloneqq x + 2]\ \{a + 1 = b\}$.
   - We can show threads are pairwise disjoint to apply the sequentialization rule. We can give the following Hilbert style proof.

1. $\{a + 1 = x + 2\}\ b \coloneqq x + 2\ \{a + 1 = b\}$	backward assignment
2. $\{x + 1 + 1 = x + 2\}\ a \coloneqq x + 1\ \{a + 1 = x + 2\}$	backward assignment
3. $\{x + 1 + 1 = x + 2\}\ a \coloneqq x + 1; b \coloneqq x + 2\ \{a + 1 = b\}$	sequence 2,1
4. $T \to x + 1 + 1 = x + 2$	predicate logic
5. $\{T\}\ a \coloneqq x + 1; b \coloneqq x + 2\ \{a + 1 = b\}$	strengthen precondition 4,3

#

i	j	$vars(S_i)$	$changes(S_j)$	S_j interferes with S_i?
1	2	a, x	b	No
2	1	b, x	a	No

6. $\{T\}[a \coloneqq x + 1 \parallel b \coloneqq x + 2]\{a + 1 = b\}$	sequentialization 5