Disjoint Programs and Conditions

A. Why?
- 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
- 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. Disjoint Parallel Programs
- The following example shows a program with an innocuous kind of parallelism: no matter what order we execute the threads in, we end up in the same final state.
- **Example 1:** Below is the the evaluation graph for \( \langle [x := a + 1 \mid | y := a \cdot 2], \sigma \rangle \) where \( \alpha = \sigma(a) \). The final state is \( \sigma[x \mapsto \alpha + 1][y \mapsto 2\alpha] \) if we take the left-hand path and \( \sigma[y \mapsto 2\alpha][x \mapsto \alpha + 1] \) if we take the right-hand path, but since \( x \not\equiv y \), these two states are exactly the same, so we show two arrows going to the final state configuration.
Example 3: $a := x + x$ and $y := y + x$ are disjoint, so $[a := a + x \mid y := y + x]$ is a DPP.

• The parallelism in DPPs is innocuous because different threads don't interfere with each other's execution: If one thread modifies a variable, that modification can't be overwritten by any other thread. Also, since the modified variable can't even be inspected by other threads, we know the modification won't affect how the other threads execute. This “disjointedness” causes all the evaluation paths to end in the same configuration.
D. Diamond Property Of Disjoint Parallel Programs

• Let \([S_1 | S_2]\) be a DPP. If \(\langle S_1, \sigma \rangle \rightarrow \langle T_1, \sigma_1 \rangle\) and \(\langle S_2, \sigma \rangle \rightarrow \langle T_2, \sigma_2 \rangle\) then there is a state \(\tau\) such that \(\langle T_1 | S_2, \sigma_1 \rangle\) and \(\langle S_1 | T_2, \sigma_2 \rangle\) both \(\rightarrow \langle T_1 | T_2, \tau \rangle\). (Note: the same \(\tau\).)

• This is called the **diamond property** because people often draw it as in the diagram shown below. The claim is that if the solid arrows exist then the dashed arrows will exist.

\[
\begin{array}{c}
\langle S_1 \parallel S_2, \sigma \rangle \\
\langle [T_1 \parallel S_2], \sigma_1 \rangle \quad \langle [S_1 \parallel T_2], \sigma_2 \rangle \\
\langle [T_1 \parallel T_2], \tau \rangle
\end{array}
\]

• The diamond property holds because the threads are disjoint so that it doesn't matter which thread you execute first: Any change in state caused by \(S_1\) will be the same whether or not you execute part of \(S_2\) (and vice-versa).

• The diamond property is a stronger version of a property called **confluence** (or **Church-Rosser**, after two investigators of the lambda calculus), where the one-step arrows are replaced by zero-or-more-step arrows (\(\rightarrow\) becomes \(\rightarrow^*\)). The diamond property is stronger because if a computation system has the diamond property, then it also has confluence, but the converse is not true.

\[
\begin{array}{c}
\langle S, \sigma \rangle \\
\langle T_1, \sigma_1 \rangle \quad \langle T_2, \sigma_2 \rangle \\
\langle T, \tau \rangle
\end{array}
\]

• Basically, a computation system in general (not just parallel programs) is confluent if execution doesn't have side effects. Everyday arithmetic expressions are confluent; C expressions with assignment operators are not.

• Because execution of disjoint parallel programs is confluent, if execution terminates, it terminates in a unique state.

• **Theorem (Unique Result of Disjoint Parallel Program):** If \(S\) is a disjoint parallel program then either \(M(S, \sigma) = \{\tau\}\) (for some \(\tau \in \Sigma\)), \(\{\bot_d\}\), or \(\{\bot_e\}\).

• **Proof:** If \(\langle S, \sigma \rangle \rightarrow^* \langle E, \tau_1 \rangle\) and \(\langle S, \sigma \rangle \rightarrow^* \langle E, \tau_2 \rangle\), then by confluence, there exists some common \(\langle S', \tau \rangle\) that both \(\langle E, \tau_1 \rangle\) and \(\langle E, \tau_2 \rangle\) can \(\rightarrow^*\) to. Since no semantics rule take \(\langle E, ... \rangle \rightarrow\) anything, the \(\rightarrow^*\) relations must both involve zero steps, so \(S'\) is \(E\) and \(\tau = \tau_1 = \tau_2\).
E. Sequentialization Proof Rule for Disjoint Parallel Programs

- We'll have three rules for proving disjoint parallel programs correct: a sequential rule and two parallel rules. The sequential rule is powerful but burdensome.

- **Definition**: The **sequentialization** of the parallel statement \([S_1 | ... | S_n]\) is the sequence \(S_1; ...; S_n\). The **sequentialized** execution of \([S_1 | ... | S_n]\) is the execution of its sequentialization: We evaluate \(S_1\) completely, then \(S_2\) completely, and so on.

- Since it doesn’t matter how we interleave evaluation of pairwise disjoint parallel threads, their total effect will be the same as if we had evaluated them sequentially.

**Sequentialization Rule**

- If the sequential threads \(S_1, ... , S_n\) are pairwise disjoint, then
  1. \(\{p\} S_1; ...; S_n \{q\}\)
  2. \(\{p\} [S_1 | ... | S_n] \{q\}\)  

**Example 4**: First, prove \(\{T\} a := x+1; b := x+2 \{a+1 = b\}\):

- \(\{T\} a := x+1 \{a = x+1\}; b := x+2 \{a = x+1 \land b = x+2\} \{a+1 = b\}\)

- From the sequentialization rule for disjoint parallel programs, it follows that
  \(\{T\} [a := x+1 | b := x+2] \{a+1 = b\}\)

**Example 5**: From \(\{x = y\} \{x+1 = y+1\} x := x+1; \{x = y+1\} y := y+1 \{x = y\}\)

- We can prove \(\{x = y\} x := x+1; y := y+1 \{x = y\}\)

- So by the sequentialization rule for disjoint parallel programs,
  \(\{x = y\} [x := x+1 | y := y+1] \{x = y\}\)

F. Disjoint Parallelism Rule for DPPs?

- The sequentialization proof rule for DPPs lets us reason about DPPs, which is nice, but it to use it, we have to develop many intermediate conditions: To prove \(\{p\} [S_1 | ... | S_n] \{q\}\), we need to prove \(\{p\} S_1; ...; S_n \{q\}\), which (if we use \(wp\)) means finding a sequence of preconditions \(q_1, ... , q_n\) and proving

\(\{p\} \{q_n\} S_1; \{q_{n-1}\} S_2; \{q_{n-2}\} ...; \{q_1\} S_n \{q\}\)

The proofs of \(q_1, q_2, ... q_n\) can get increasingly complicated because each \(q_i\) can depend on all the threads and conditions to its right.
Ideally, we'd like to prove correctness of the individual threads and then combine them to get correctness of the parallel program. I.e., we'd like something that lets us take sequential thread triples, combine their preconditions, run them in parallel, and conclude the conjunction of their postconditions.

**Example 6:** As a proof rule application, we'd like something like
\[
\{x \geq 0\} z := x \{z \geq 0\}
\]
\[
\{y \leq 0\} w := -y \{w \geq 0\}
\]
\[
\{x \geq 0 \land y \leq 0\} [z := x \mid \mid w := -y] \{z \geq 0 \land w \geq 0\}
\]
by ???

As a full proof outline, we would have
\[
\{x \geq 0 \land y \leq 0\}
\]
\[
[\{x \geq 0\} z := x \{z \geq 0\}]
\]
\[
\mid \mid \{y \leq 0\} w := -y \{w \geq 0\}
\]
\[
] \{z \geq 0 \land w \geq 0\}
\]

But we must be careful — this combination doesn't always work.

**Example 7:** In the (invalid) proof outline below, we can't combine the \(x = 1\) and \(x = y\) postconditions because the first thread invalidates the \(x = 0\) precondition that the second thread relies on.

\[
\{x = 0\}
\]
\[
[\{x = 0\} x := 1 \{x = 1\}]
\]
\[
\mid \mid \{x = 0\} y := 0 \{x = y\}
\]
\[
] \{x = 1 \land x = y\} \quad \text{**Bad! Can't combine the postconditions!**}
\]
\[
\{x = y = 1\}
\]

Even though the threads of the DPP can't affect each others runtime states, they can affect variables that appear in the conditions other threads. We need an additional restriction on our programs.

**Definition:** \(\{p_1\} S_1 \{q_1\}\) and \(\{p_2\} S_2 \{q_2\}\) have **disjoint conditions** if neither program can affect the other's conditions: \(\text{Change}(S_1) \cap \text{Free}(p_2, q_2) = \emptyset\) and \(\text{Change}(S_2) \cap \text{Free}(p_1, q_1) = \emptyset\).

**Example 8:** Some disjoint conditions:
- \(\{x \geq 0\} z := x \{z \geq 0\}\) and \(\{y \leq 0\} w := -y \{w \geq 0\}\), since \(\{z\} \cap \{w, y\} = \emptyset\) and \(\{w\} \cap \{x, z\} = \emptyset\).
- \(\{z = 0\} x := z + 1 \{x \leq z\}\) and \(\{z = 0\} y := z \{z = y\}\), since \(\{x\} \cap \{y, z\} = \{y\} \cap \{x, z\} = \emptyset\).

**Example 9:** Some nondisjoint conditions:
- \(\{x = 0\} x := 1 \{x = 1\}\) and \(\{x = 0\} y := 0 \{x = y\}\), since \(\{x\} \cap \{x, y\} = \{x\}\) (the first thread interferes with the conditions of the second thread). Note that thread 2 doesn't interfere with thread 1, since \(\{y\} \cap \{x\} = \emptyset\).
- \(\{x \geq y\} x := x + 1 \{x > y\}\) and \(\{y = z\} y := y \times 2; z := z \times 2 \{y = z\}\), since \(\{y, z\} \cap \{x, y\} = \{y\}\) (the second thread interferes with the conditions of the first thread). Note thread 1 doesn't interfere with thread 2.
- If we have a variable \(y_0\) that holds the value of \(y\) before running the second thread, then we can get disjoint conditions:
• \( \{ x \geq y_0 \} \ x := x + 1 \ \{ x > y_0 \} \)
• \( \{ y = y_0 \land y = z \} \ y := y \ast 2 \ ; \ z := z \ast 2 \ \{ y = z \land y = y_0 + 1 \} \).

If two triples have disjoint programs and conditions, then neither can modify information used by the programs or conditions of the other. E.g., take the threads
• \( \{ x = z \} \ x := x + 2 \ ; \ x := x \ast 3 \ \{ x = 3 \ast z + 6 \} \)
• \( \{ y \ast 2 > z \geq 1 \} \ y := y \ast 2 \ \{ y > z \geq 1 \} \)

The first thread changes \( x \), uses \( x \) in its program and uses \( x \) and \( z \) in its conditions. The second thread changes \( y \), uses \( y \) in its program, and uses \( y \) and \( z \) in its conditions. Therefore the threads have disjoint programs and disjoint conditions. No matter how we interleave execution, the first thread's changes to \( x \) will not affect \( y \) or \( z \), and the second thread's changes to \( y \) will not affect \( x \) or \( z \).

**Disjoint Parallelism Rule (the parallelism rule for disjoint programs with disjoint conditions)**

1. \( \{ p_1 \} \ S_1 \{ q_1 \} \)
2. \( \{ p_2 \} \ S_2 \{ q_2 \} \)
...  
n. \( \{ p_n \} \ S_n \{ q_n \} \)
n+1 \( \{ p_1 \land p_2 \land ... \land p_n \} \)
\[ [ S_1 \ | \ ... \ | \ S_n ] \]
\( \{ q_1 \land q_2 \land ... \land q_n \} \)

Disjoint Parallelism, 1, 2, ..., \( n \)

where the \( \{ p_i \} \ S_i \{ q_i \} \) are pairwise disjoint programs with disjoint conditions.

**Example 4 revisited**: The program from Example 4 can use disjoint parallelism, since the threads are disjoint parallel with disjoint conditions.

\( \{ T \} \ \\
\[ \{ T \} \ a := x + 1 \ \{ a = x + 1 \} \]
\[ \ | \ \{ T \} \ b := x + 2 \ \{ b = x + 2 \} \]
\( \{ a = x + 1 \land b = x + 2 \} \)
\( \{ a + 1 = b \} \)

**Example 6 revisited**: The program in Example 6 can also use disjoint parallelism.

\( \{ x \geq 0 \land y \leq 0 \} \ \\
\[ \{ x \geq 0 \} \ z := x \ \{ z \geq 0 \} \]
\[ \ | \ \{ y \leq 0 \} \ w := -y \ \{ w \geq 0 \} \]
\( \} \ { z \geq 0 \land w \geq 0 } \)

**Example 7, revisited**: The program in Example 7 has disjoint parallel threads but not disjoint conditions (thread 1 modifies \( x \), which appears in the conditions of thread 2).

\( \{ x = 0 \} \ \\
\[ \{ x = 0 \} \ x := 1 \ \{ x = 1 \} \]
\[ \ | \ \{ x = 0 \} \ y := 0 \ \{ x = y = 0 \} \]
\( \} \ { x = 1 \land x = y = 0 } \)  // Conditions are not disjoint from thread 1
\( \{ x = y = 1 = 0 \} \)  // Can't use disjoint parallelism
• To fix this situation we could change thread 1 so that it doesn't interfere with the conditions of thread 2 or change the conditions of thread 2 so that they aren't interfered with by thread 1 or some combination of these tactics.

\[
\begin{align*}
\{ x = 0 \land x_0 = 0 \} \\
\{ x_0 = 0 \land x = 0 \} x := 1 \{ x = 1 \}
\end{align*}
\]

// use sp of second thread to avoid talking about x in the postcondition?
\[
\begin{align*}
\{ x = 1 \land x_0 = 0 \land y = 0 \} \\
\{ x = 1 \land x_0 = 0 \land y = 0 \}
\end{align*}
\]

• Thread 1 still interferes with the conditions of thread 2 (but only the precondition, at least).

• We'll have to remove information about x from the precondition of thread 2.

\[
\begin{align*}
\{ x = 0 \land x = x_0 \} & \quad \text{// Only need } x_0 \text{ if we use } sp \\
\{ x = 0 \land x = x_0 \} x := 1 \{ x_0 = 0 \land x = 1 \}
\end{align*}
\]

// no info about x in precondition or postcondition
\[
\begin{align*}
\{ x_0 = 0 \land x = 1 \land y = 0 \} & \quad \text{// Can use disjoint parallelism} \\
\{ x = 1 \land y = 0 \}
\end{align*}
\]