Parallel Programs; Disjoint Programs and Conditions

CS 536: Science of Programming, Fall 2018

11/13

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 these lectures, 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 `:=` $[S \parallel S \parallel \ldots \parallel S]$. We say $[S_1 \parallel S_2 \parallel \ldots \parallel S_n]$ 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. (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 || 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 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

  while B **do** [x := x+1 || x := x*2] **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**:
  - $\vDash \{x = 5\}\ x := x+1\ \{x = 6\}$ and $\vDash \{x = 5\}\ x := x*2\ \{x = 10\}$
  - But $\vDash \{x = 5\}\ [x := x+1 \parallel x := x*2]\ \{x = 11 \vee x = 12\}$
- 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

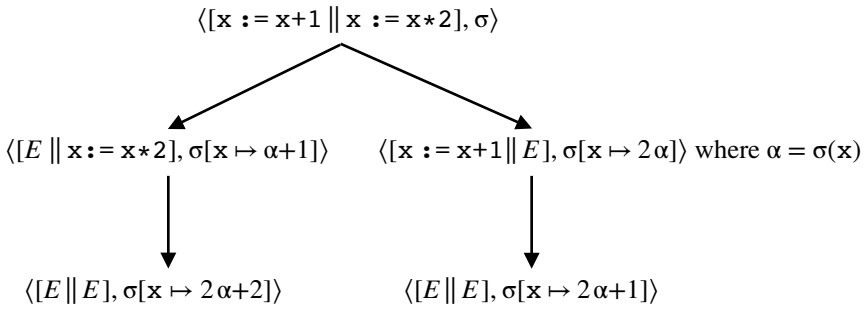
- To execute the sequential composition $S_1; \ldots; S_n$ for one step, we execute S_1 for one step.
- To execute the parallel composition $[S_1 \parallel \ldots \parallel S_n]$ for one step, we take one of the threads and evaluate it for one step.

Operational Semantics of Parallel Programs

- **Definition**: For $[S_1 \parallel \ldots \parallel S_n]$, then for each $k = 1, 2, \ldots, n$, if $\langle S_k, \sigma\rangle \to \langle T_k, \tau_k\rangle$, then

  $$\langle [S_1 \parallel \ldots \parallel S_n], \sigma\rangle \to \langle [S_1 \parallel \ldots \parallel S_{k-1} \parallel T_k \parallel S_{k+1} \parallel \ldots \parallel S_n], \tau_k\rangle \text{ [for each } k]$$

- A completely-executed parallel program looks like $[E \parallel \ldots \parallel E \parallel E]$. Since we've been writing E for the completely-executed program, for consistency's sake we'll treat $[E \parallel \ldots \parallel E \parallel E]$ as $\equiv$ to E.
  - In particular, $\langle [E \parallel E], \tau\rangle = \langle E, \tau\rangle$ so $\langle [E \parallel E], \tau\rangle \to \langle E, \tau\rangle$ is incorrect. (Since the configurations are equal, we do have $\langle [E \parallel E], \tau\rangle \to^0 \langle E, \tau\rangle$, however.)
- Recall that the **evaluation graph** for $\langle S, \sigma\rangle$ is the directed graph of configurations and evaluation arrows leading from $\langle S, \sigma\rangle$. Also, when drawing evaluation graphs, the configuration nodes need to be different: 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. An actual execution is a path through the graph.
- **Example 3**: The evaluation graph below is for the program 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.

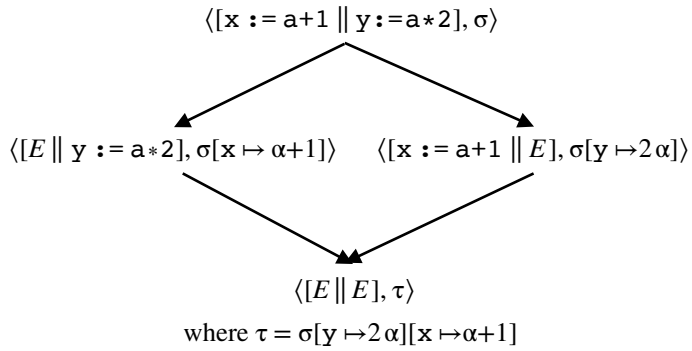


Denotational Execution of Parallel Programs

- As before, the denotational meaning of a program in a state is the set of all possible terminating states (plus possibly the pseudostates $\perp_e$ and $\perp_e$).
  - $M(S, \sigma) = \{\tau \in \Sigma \mid \langle S, \sigma\rangle \rightarrow^* \langle E, \tau\rangle\} \cup \{\perp_d\}$ (if S can diverge at σ) $\cup \{\perp_e\}$ (if $\langle S, \sigma\rangle \rightarrow^* \langle \perp_e, \tau\rangle$).
- **Example 4**: $M([\mathtt{x := x+1} \parallel \mathtt{x := x*2}], \sigma) = \{\sigma[\mathtt{x} \mapsto 2\alpha+1], \sigma[\mathtt{x} \mapsto 2\alpha+2]\}$, where $\alpha = \sigma(\mathtt{x})$.
- **Example 5**: If $W \equiv \mathtt{x := 0;\ while\ x = 0\ do\ [x := 0 \parallel x := 1]\ od}$, then $M(W, \sigma) = \{\sigma[\mathtt{x} \mapsto 1], \perp_d\}$

E. 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 6**: Below is the the evaluation graph for $\langle[\mathtt{x := a+1} \parallel \mathtt{y := a*2}], \sigma\rangle$ where $\alpha = \sigma(\mathtt{a})$. The final state is $\sigma[\mathtt{x} \mapsto \alpha+1][\mathtt{y} \mapsto 2\alpha]$ if we take the left-hand path and $\sigma[\mathtt{y} \mapsto 2\alpha][\mathtt{x} \mapsto \alpha+1]$ if we take the right-hand path, but since $\mathtt{x} \not\equiv \mathtt{y}$, these two states are exactly the same, so we show two arrows going to the final state configuration.

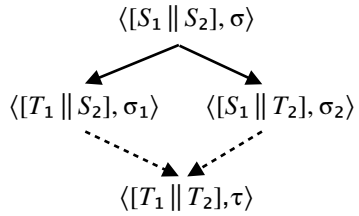


- **Disjoint Parallel Programs (DPPs)** model computations with n processors that share readable memory but not writable memory. In a disjoint parallel program, for every variable `x` that appears in the program, either
  - One or more threads read `x` (i.e., look up its value) and no thread writes to `x` (i.e., assigns it a value).
  - Exactly one thread writes to `x` and that thread can read `x`; no other thread can read or write `x`.
- **Definition**: *vars*(S) = the set of variables that appear in S and *change*(S) = the set of variables that appear on the left-hand side of assignments in S. Since these sets are statically calculable, they are $\supseteq$ the sets of variables actually read or written at runtime. Another way to say this is that execution order isn't taken into account. E.g., If $S \equiv$ **if** B **then** `x := 1` **else** `y := 1` **fi** then *change*(S) = {`x`, `y`}.

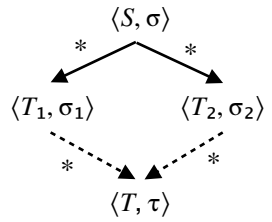
- **Definition**: The threads S_1, S_2, …, S_n are **pairwise disjoint** if no thread can change the variables used by any other: I.e., $change(S_i) \cap vars(S_j) = \varnothing$ for all $1 \le i \ne j \le n$.
  - **Example 7**: $S_1 \equiv$ `a := a+x` and $S_2 \equiv$ `y := y+x` are disjoint: $change(S_1) = \{$`a`$\}$ and $vars(S_2) = \{$`x`, `y`$\}$ and these sets don't intersect. Similarly, $change(S_2) = \{$`y`$\}$ and $vars(S_1) = \{$`a`, `x`$\}$ and those sets don't intersect.
- **Definition**: For $n > 1$, if S_1, S_2, …, S_n are pairwise disjoint, then $[S_1 \,\|\, S_2 \,\|\, \ldots \,\|\, S_n]$ is their **disjoint parallel composition**. We also say $[S_1 \,\|\, S_2 \,\|\, \ldots \,\|\, S_n]$ is a **disjoint parallel program** (DPP).
- **Example 8:**
  - `a := a+x` and `y := y+x` are disjoint, so [`a := a+x` $\|$ `y := y+x`] is a DPP.
  - `a := x+1` and `y := x+2` are disjoint, so [`a := x+1` $\|$ `y := x+2`] is a DPP.
  - `a := x` and `x := c` are not disjoint so [`a := x` $\|$ `x := c`] isn't a DPP.
  - `a := x` and `x := x+1` are not disjoint so [`a := x` $\|$ `x := x+1`] isn't a DPP.
  - `x := a+1` and `x := b*2` are not disjoint so [`x := a+1` $\|$ `x := b*2`] isn't 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.

F. Diamond Property Of Disjoint Parallel Programs

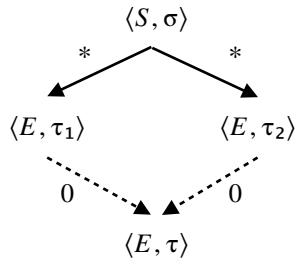
- Let $[S_1 \,\|\, S_2]$ be a DPP. If $\langle S_1, \sigma\rangle \to \langle T_1, \sigma_1\rangle$ and $\langle S_2, \sigma\rangle \to \langle T_2, \sigma_2\rangle$ then there is a state τ such that $\langle [T_1 \,\|\, S_2], \sigma_1\rangle$ and $\langle [S_1 \,\|\, T_2], \sigma_2\rangle$ both $\to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



- 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 ($\to$ becomes $\to$*). 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.



- 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 $\tau \in \Sigma$), $\{\perp_d\}$, or $\{\perp_e\}$.
- **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 $\langle E, \ldots\rangle \rightarrow$ anything, the $\rightarrow^*$ relations must both involve zero steps, so S' is E and $\tau = \tau_1 = \tau_2$.



where $\tau = \tau_1 = \tau_2$

G. 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 $[S_1 \| \ldots \| S_n]$ is the sequence S_1`;` $\ldots$`;` S_n. The **sequentialized** execution of $[S_1 \| \ldots \| S_n]$, 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 $S_1, \ldots, S_n$ are pairwise disjoint, then

1. $\{p\}\ S_1$`;` $\ldots$`;` $S_n\ \{q\}$
2. $\{p\}\ [S_1 \| \ldots \| S_n]\ \{q\}$ Sequentialization, 1

- **Example 9:** First, prove {`T`} `a := x+1; b := x+2` {`a+1 = b`}:
  - {`T`} {`x+1+1 = x+2`} `a := x+1;` {`a+1 = x+2`} `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 10**: 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}.

H. 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 $\{p\}\ [S_1 \| \ldots \| S_n]\ \{q\}$, we need to prove $\{p\}\ S_1; \ldots ; S_n\ \{q\}$, which (if we use *wp*) means finding a sequence of preconditions $q_1, \ldots, q_n$ and proving

  $$\{p\}\ \{q_n\}\ S_1; \{q_{n-1}\}\ S_2 ; \{q_{n-2}\} \ldots \{q_1\}\ S_n\ \{q\}$$

  The proofs of $q_1, q_2, \ldots q_n$ 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 11**: As a proof rule application, we'd like something like

  {x ≥ 0} z := x {z ≥ 0}
  {y ≤ 0} w := −y {w ≥ 0}
  {x ≥ 0 ∧ y ≤ 0} [z := x ‖ w := −y] {z ≥ 0 ∧ w ≥ 0} by ???

  As a full proof outline, we would have

  {x ≥ 0 ∧ y ≤ 0}
  [{x ≥ 0} z := x {z ≥ 0}
  ‖ {y ≤ 0} w := −y {w ≥ 0}
  ] {z ≥ 0 ∧ w ≥ 0}

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

- **Example 12**: 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}
  ‖ {x = 0} y := 0 {x = y}
  ] {x = 1 ∧ x = y} ⇐ **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: $Change(S_1) \cap Free(p_2, q_2) = \varnothing$ and $Change(S_2) \cap Free(p_1, q_1) = \varnothing$.

- **Example 13:** Some disjoint conditions:
  - {x ≥ 0} z := x {z ≥ 0} and {y ≤ 0} w := −y {w ≥ 0}, since {z} ∩ {w, y} = ∅ and {w} ∩ {x, z} = ∅.
  - {z = 0} x := z+1 {x ≤ z} and {z = 0} y := z {z = y}, since {x} ∩ {y, z} = {y} ∩ {x, z} = ∅.
- **Example 14:** Some nondisjoint conditions:
  - {x = 0} x := 1 {x = 1} and {x = 0} y := 0 {x = y}, since {x} ∩ {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} ∩ {x} = ∅.
  - {x ≥ y} x := x+1 {x > y} and {y = z} y := y*2; z := z*2 {y = z}, since {y, z} ∩ {x, y} = {y} (the second thread interferes with the conditions of the first thread). Note thread 1 doesn't interfere with thread 2.
- 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∗3 {x = 3*z+6}
  - {y*2 > z ≥ 1} y := y*2 {y > z ≥ 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

1. $\{p_1\}\ S_1\ \{q_1\}$
2. $\{p_2\}\ S_2\ \{q_2\}$

…

n. $\{p_n\}\ S_n\ \{q_n\}$

n+1 $\{p_1 \wedge p_2 \wedge \ldots \wedge p_n\}\ [S_1 \parallel \ldots \parallel S_n]\ \{q_1 \wedge q_2 \wedge \ldots \wedge q_n\}$ Disjoint Parallelism, 1, 2, …, n

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

- **Example 9 revisited**: The program from Example 9 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 ∧ b = x+2}
{a+1 = b}
```

- **Example 11 revisited**: The program in Example 11 can also use disjoint parallelism.

```
{x ≥ 0 ∧ y ≤ 0}
[ {x ≥ 0} z := x {z ≥ 0}
|| {y ≤ 0} w := −y {w ≥ 0}
] {z ≥ 0 ∧ w ≥ 0}
```


- **Example 12, revisited**: The program in Example 12 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}        // Conditions are not disjoint from thread 1
] {x = 1 ∧ x = y}                // Can't use disjoint parallelism
{x = y = 1}
```


Parallel Programs; Disjoint Programs and Conditions

CS 536: Science of Programming

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

At the end of these activities you should be able to

- Draw evaluation graphs for parallel programs.
- Recognize disjoint parallel programs and correctness triples with disjoint conditions
- Use the rules for sequentialization and disjoint parallelism

C. Questions

Basics of Parallel Programs

Draw evaluation graphs for the following configurations

1. $\langle$[x := v || y := v+2 || z := v*2], σ$\rangle$, where σ(v) = α.
2. $\langle$[x := 1 || x := −1]; y := y+x, σ$\rangle$
3. $\langle$[x := v || y := v+2 ; z := v*2], σ$\rangle$, where σ(v) = 6. Note that in the second thread, the assignment to y must be done before the assignment to z.
4. $\langle$[v := 8 || v := v+2 ; v := v*2], σ$\rangle$, where σ(v) = 4. Note that in the second thread, the assignment v := v+2 must be done before the assignment v := v*2.
5. $\langle$**while** x ≤ n **do** [x := x+1 || y := y*2] **od**, σ$\rangle$, where σ of x, y, and z are 0, 1, and 2 respectively.
6. In $[S_1 \parallel S_2 \parallel \ldots \parallel S_n]$ can any of the threads S_1, S_2, …, S_n contain parallel statements? Can parallel statements be embedded within loops or conditionals?
7. In general, if we have $\{ p_1 \}\ S_1\ \{ q_1 \}$ and $\{ p_2 \}\ S_2\ \{ q_2 \}$, do we know $\{ p_1 \wedge p_2 \}\ [S_1 \parallel S_2]\ \{ q_1 \wedge q_2 \}$? Why or why not? What if we have $\{ p \}\ S_1\ \{ q \}$ and $\{ p \}\ S_2\ \{ q \}$ — Do we know $\{ p \}\ [S_1 \parallel S_2]\ \{ q \}$?

Disjoint Parallel Programs

8. What are *vars*(*S*), *change*(*S*), and how are they used in the definition of "a pair of disjoint programs"?

9. What is a disjoint parallel program? What kind of parallel computation does it model?
10. What are the diamond and confluence/Church-Rosser properties and what do they imply about the evaluation graph for a disjoint parallel program?
11. What is the sequentialization proof rule for disjoint parallel programs?

Disjoint Parallel Programs with Disjoint Conditions

To figure out whether a set of sequential threads is disjoint parallel and/or has disjoint conditions, we have to compare the *changed*, *used*, and *free* variables for various combinations of triples. In the tables below, the first two columns specify the pair of threads *i* and *j* we're discussing, the next three columns specify the variables changed by thread *i*, used by thread *j*, and free in the conditions of thread *j*. The last two columns answer the questions "Does thread *i* interfere with the program of thread *j*?" and "Does thread *i* interfere with the conditions of thread *j*?"

12. Are the following programs parallel disjoint with disjoint conditions?
    - {**T**} `x := 1 ; y := 1` {`x = 1`}
    - {`x = 0`} `z := 0` {`x = z`}

i	*j*	*Change i*	*Vars j*	*Free j*	*Disjoint Program?*	*Disjoint Cond?*
1	2					
2	1					

13. Are the following programs parallel disjoint with disjoint conditions?
    - {**T**} `x := 1 ; y := 0` {`x = 1`}
    - {`z = 0`} `z := z*x` {`z = 0`}

i	*j*	*Change i*	*Vars j*	*Free j*	*Disjoint Program?*	*Disjoint Cond?*
1	2					
2	1					

14. Are the following programs parallel disjoint with disjoint conditions?

- {**T**} **if** x > 0 **then** y := 1; z := 2 **fi** {x ≤ 0 → z = 2}
- {**T**} **if** x ≤ 0 **then** z := 2; y := 3 **fi** {x ≤ 0 → y = 3}

i	*j*	*Change i*	*Vars j*	*Free j*	*Disjoint Program?*	*Disjoint Cond?*

15. Are the following programs parallel disjoint with disjoint conditions?

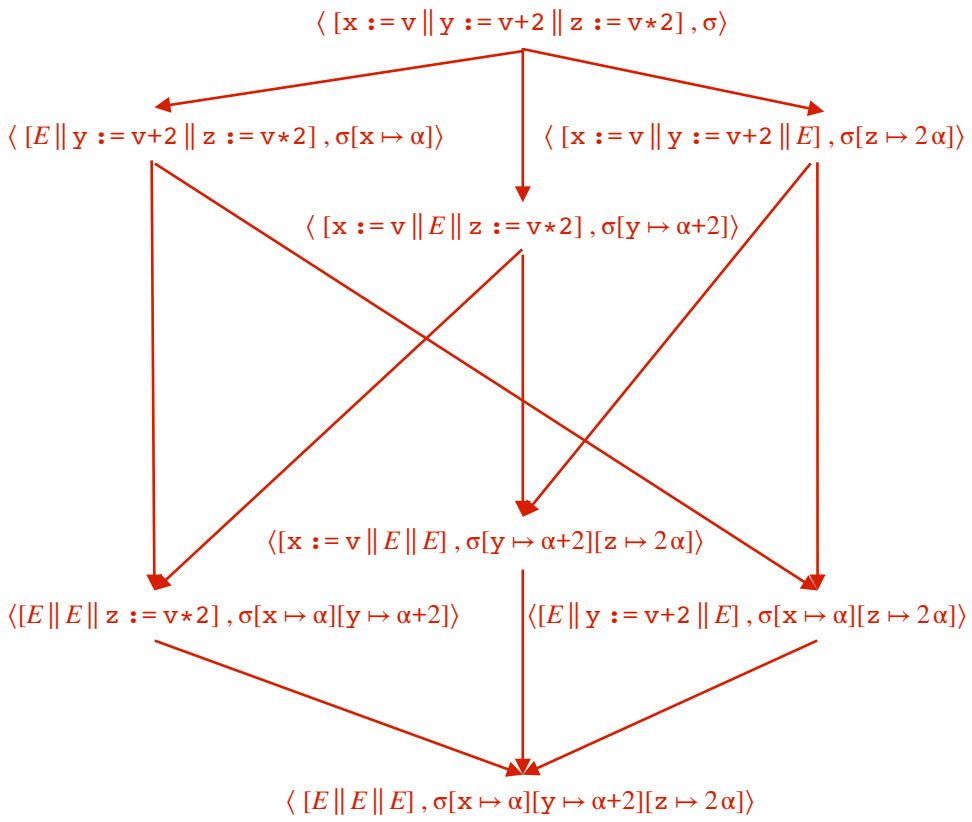
- {**T**} x := u ; y := u {x = y}
- {z > 0} z := z−1 ; v := z {v = z}
- {w ≥ u} w := w+1 {w > u}

i	*j*	*Change i*	*Vars j*	*Free j*	*Disjoint Program?*	*Disjoint Cond?*
1	2					
1	3					
2	1					
2	3					
3	1					
3	2					

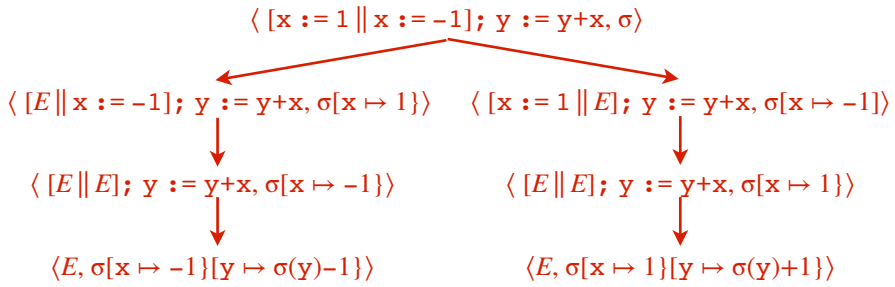
Solution to Activity 21 (Parallel Programs; Disjoint Programs and Conditions)

Basics of Parallel Program

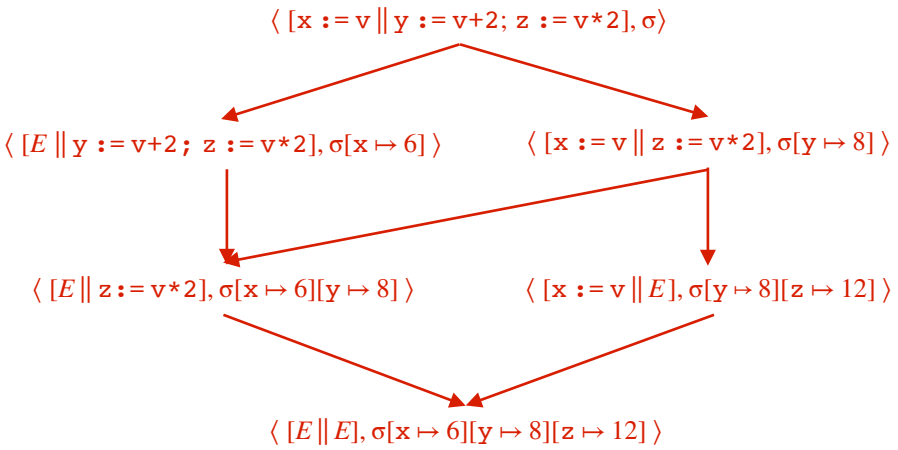
1. Evaluate $\langle\, [\mathtt{x} := \mathtt{v} \parallel \mathtt{y} := \mathtt{v}+2 \parallel \mathtt{z} := \mathtt{v}*2]\,, \sigma\rangle$, where $\sigma(\mathtt{v}) = \alpha$.)



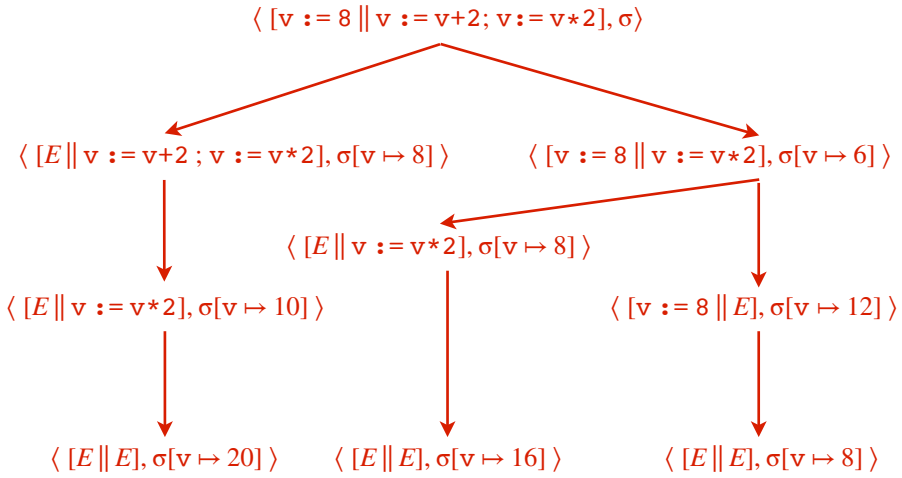
2. Evaluate $\langle\, [\mathtt{x} := 1 \parallel \mathtt{x} := -1];\ \mathtt{y} := \mathtt{y}+\mathtt{x},\ \sigma\rangle$



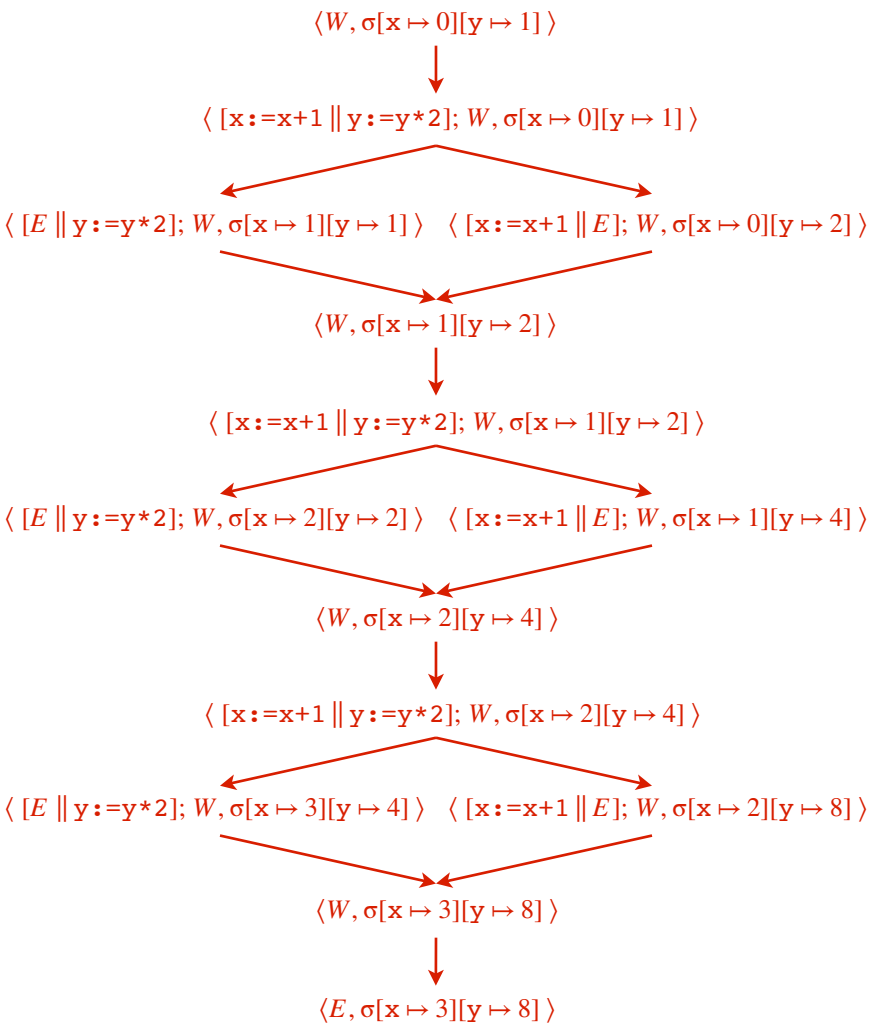
3. $\langle$ [x := v || y := v+2; z := v*2], σ$\rangle$ where σ(v) = 6) [Note: This graph is a subgraph of Question 1.]



4. ([v := 8 || v := v+2 ; v := v*2], if σ(v) = 4)



5. (**`while`** x ≤ n **`do`** [x := x+1 ‖ y := y*2] **`od`**, if σ(x) = 0, σ(y) = 1, and σ(n) = 2)
   Just to be explicit, I wrote σ[x ↦ 0][y ↦ 1] below but just σ is fine.



6. No, in $[S_1 \| S_2 \| \ldots \| S_n]$ the threads cannot contain parallel statements, but yes, parallel statements can be embedded within loops and conditionals.

7. In general, knowing $\{p_1\}\ S_1\ \{q_1\}$ and $\{p_2\}\ S_2\ \{q_2\}$ doesn't mean we know $\{p_1 \wedge p_2\}\ [S_1 \| S_2]\ \{q_1 \wedge q_2\}$, because one thread might rely on the state having certain properties that are invalidated by interleaved execution by the other thread. This is true even if $p_1 \equiv p_2 \equiv p$ and $q_1 \equiv q_2 \equiv q$ because the different threads can establish different sequences of states as they work to establish q.

Disjoint Parallel Programs

8. *vars*(*S*) and *change*(*S*) = are the sets of variables that appear in *S* or in (left-hand sides of) assignments in *S* respectively. Threads *S* and *S*′ are disjoint if neither can change the variables used by the other: $change(S) \cap vars(S') = change(S') \cap vars(S) = \varnothing$.

9. A disjoint parallel program has pairwise disjoint threads. It models computations on n different processors that can share readable memory but not writeable memory.
10. The diamond property says that if $\langle S, \sigma\rangle \rightarrow$ both $\langle T_1, \sigma_1\rangle$ and $\langle T_2, \sigma_2\rangle$, then $\langle T_1, \sigma_1\rangle$ and $\langle T_2, \sigma_2\rangle$ both $\rightarrow$ some $\langle T, \tau\rangle$. Confluence/Church-Rosser replaces $\rightarrow$ by $\rightarrow^*$. Disjoint parallel programs have the diamond property (so are confluent) and have a unique result state; the evaluation graph has a unique sink node.
11. The Sequentialization rule says that if $S_1, \ldots, S_n$ are pairwise disjoint, then knowing $\{p\}\ S_1; \ldots; S_n\ \{q\}$ implies knowing $\{p\}\ [S_1 \parallel \ldots \parallel S_n]\ \{q\}$.

Disjoint Parallel Programs with Disjoint Conditions

(I was lazy and left out the commas in lists like `x, y`; you can put them in or leave them out.)

12. No: Thread 1 interferes with the conditions of thread 2

i	*j*	*Change i*	*Vars j*	*Free j*	*Disjoint Program?*	*Disjoint Cond?*
1	2	`x y`	`z`	`x z`	Yes	No (because of) x
2	1	`z`	`x y`	`x`	Yes	Yes

13. No: Thread 1 interferes with the program of thread 2.

i	*j*	*Change i*	*Vars j*	*Free j*	*Disjoint Program?*	*Disjoint Cond?*
1	2	`x y`	`x z`	`z`	No (because of) x	Yes
2	1	`z`	`x y`	`x`	Yes	Yes

14. No: Each interferes with the other's programs and conditions.

i	*j*	*Change i*	*Vars j*	*Free j*	*Disjoint Program?*	*Disjoint Cond?*
1	2	`y z`	`x y z`	`x y`	No: y, z	No: y
2	1	`y z`	`x y z`	`x z`	No: y, z	No: z

15. Yes, these are parallel disjoint with disjoint conditions

i	*j*	*Change i*	*Vars j*	*Free j*	*Disjoint Program?*	*Disjoint Cond?*
1	2	`x y`	`v z`	`v z`	Yes	Yes
1	3	`x y`	`w`	`u w`	Yes	Yes
2	1	`v z`	`u x y`	`x y`	Yes	Yes
2	3	`v z`	`w`	`u w`	Yes	Yes
3	1	`w`	`u x y`	`x y`	Yes	Yes
3	2	`w`	`v z`	`v z`	Yes	Yes

