Disjoint Programs

CS 536: Science of Programming, Spring 2022

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.
- The sequentialization rule (though imperfect) gives us a way to prove the correctness of disjoint parallel programs.

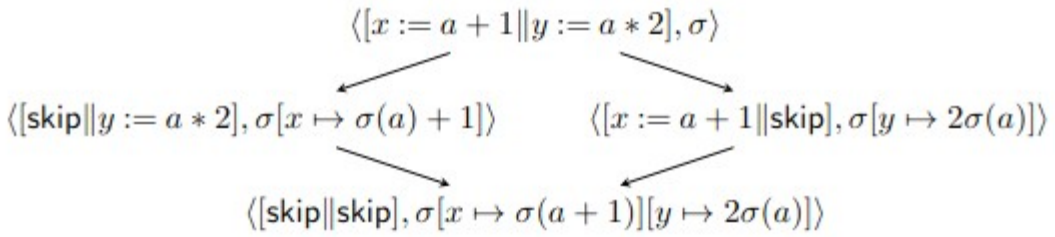
B. Objectives

After this class, you should know

- What distinguishes disjoint parallel programs
- The sequentialization rule for disjoint parallel programs

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.
- This has a property we mentioned last time, that it is free of *race conditions*.
- *Example 1*: Here is the the evaluation graph for $\langle [x := a+1 \| y := a*2], \sigma \rangle$. The final state is $\sigma[x \mapsto \sigma(a)+1][y \mapsto 2\sigma(a)]$ if we take the left-hand path and $\sigma[y \mapsto 2\sigma(a)][x \mapsto \sigma(a)+1]$ if we take the right-hand path, but since $x \not\equiv y \not\equiv$ a, these two states are exactly the same, so we show two arrows going to the final state configuration.



- 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 ⊇ the sets of variables actually read or written at runtime, eg., If $s \equiv$ *if e then* { $x := 1$ } *else* { $y := 1$ } then *change*(s) = {x, y}.
- *Definition:* Thread s_1 *interferes with* s_2 if s_1 changes the variables used by s_2, i.e. if *change*(s_1) ∩ *vars*(s_2) ≠ ∅.
- *Definition*: Threads s_1 and s_2 are *disjoint* if they do not interfere with each other, i.e., *change*(s_1) ∩ *vars*(s_2) = ∅ = *change*(s_2) ∩ *vars*(s_1)
- *Definition*: The threads s_1, s_2, …, s_n are *pairwise disjoint* if no two threads interfere: i.e., *change*(s_i) ∩ *vars*(s_j) = ∅ for all $1 \le i \ne j \le n$.
- *Example 2*: $s_1 \equiv a := a+x$ 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_n] is their *disjoint parallel composition*. We also say [s_1 ‖ … ‖ s_n] is a *disjoint parallel program* (DPP).
- *Example 3:*
  - $a := a+x$ 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.

- An easy way to calculate whether or not programs are pairwise disjoint is to use a table listing the *change*(s_j) and *vars*(s_k) sets for each pair of pair of threads.

- *Example 4*: Here is a table for $a := a+x$ and $y := y+x$, showing that they are pairwise disjoint:

j	*k*	*Change j*	*Vars k*	*j interferes with with k*
1	2	*a*	*x y*	no
2	1	*y*	*a x*	no

Conclusion: The two programs are pairwise disjoint.

- *Example 5:* Here's a table for and *a := x* and *x := c* showing that while the first doesn't interfere with the second, the second does interfere with the first, which makes the pair not disjoint.

j	*k*	*Change j*	*Vars k*	*j interferes with k*
1	2	*a*	*c x*	no
2	1	*x*	*a x*	yes

Conclusion: The two programs are not pairwise disjoint.

- *Example 6:* Here's a table showing the interference relationships for the three threads
  - *a := v; v := c+b*
  - *if b > 0 then { b := c*b } else* { c *:= c*2 }*
  - *while d ≥ 0 { d := d ÷ 2 – c }*

j	*k*	*Change j*	*Vars k*	*j interferes with k*
1	2	*a v*	*b c*	no
1	*3*	*a v*	*c d*	no
2	1	*b c*	*a b c v*	yes
2	*3*	*b c*	*c d*	yes
3	*1*	*d*	*a b c v*	no
3	*2*	*d*	*b c*	no

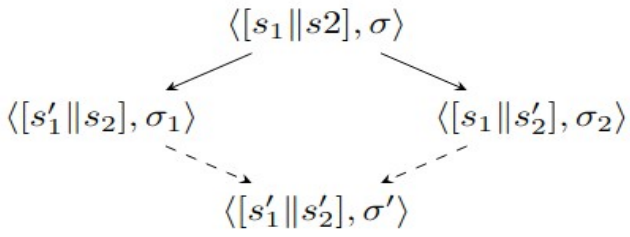
Conclusion: Thread 2 interferes with threads 1 and 3; the other combinations are disjoint

- *Disjointedness Test Can Overestimate Amount of Interference:* The disjointedness test is a static (compile-time) that aims for safety over accuracy when it comes to looking for interference. Not all the variables in *Change(...)* and *Vars(...)* are necessarily used at runtime.
- As a result, in the above, we should really say “apparently interferes with” instead of “interferes with”, since, e.g.
  if true then { x := a } else { y := a } and *y := b*
  will not actually interfere with each other at runtime, but they interfere by our definition.
- For convenience and flexibility, we'll often omit the “apparently” in “apparently interferes with”. But it’s perhaps clearer to just stick with saying “isn’t disjoint from”.

- Passing a disjointedness test of thread *j* against thread *k* guarantees that interference cannot happen, no matter what the starting state is, and no matter what execution path gets taken.
- Failing a disjointedness test simply says we can't guarantee that thread *j* interferes with thread *k*. Without knowing more about the threads and the starting state, we can't say anything about whether interference in fact doesn't occur, or occurs only with some start states, or only along some execution paths. Failing a test certainly does not guarantee that interference is inevitable at runtime.

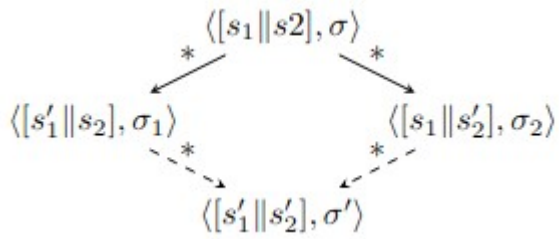
D. *The Diamond Property; Confluence*

- 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. In other words, there are no race conditions.
- In general, with $[s_1 \| s_2]$, we can execute s_1 or s_2 for one step. In an evaluation graph, the current evaluation path splits into two paths. With parallel programs in general, there might be no way for those two paths to eventually merge back together into one path, but DPPs are different.
- Let $[s_1 \| s_2]$ be a DPP. If $\langle s_1, \sigma\rangle \to \langle s_1', \sigma_1\rangle$ and $\langle S_2, \sigma\rangle \to \langle s_2', \sigma_2\rangle$ then there is a state σ' such that $\langle [s_1' \| s_2], \sigma_1\rangle$ and $\langle [s_1 \| s_2'], \sigma_2\rangle$ both $\to \langle [s_1' \| s_2'], \sigma'\rangle$. (Note: the same σ'.)
- 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.

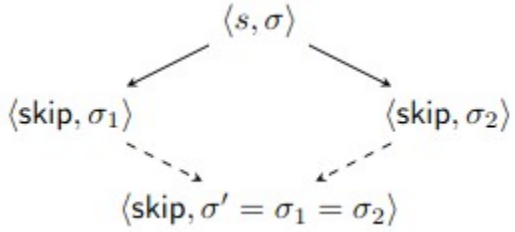


- The diamond property holds because the threads are disjoint so that it doesn't matter which thread you execute first: Any change in state caused by S_1 will be the same whether or not you execute part of S_2 (and vice-versa).
- Note that the diamond property actually says more than just "the paths can *eventually* merge back together": it says they do it in one step! The weaker property, that the paths will eventually merge, is 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 (→

becomes →*). The diamond property is stronger because it implies 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, σ) = {σ'}*, *{⊥d}*, or *{⊥e}*.
- *Proof*: If ⟨*s*, *σ*⟩ →* ⟨skip, *σ1*⟩ and ⟨*s*, *σ*⟩ →* ⟨skip, *σ2*⟩, then by confluence, there exists some common ⟨*s'*, *σ'*⟩ that both ⟨skip, *σ1*⟩ and ⟨skip, *σ2*⟩ can →* to. Since no semantics rule takes⟨*skip*, ... ⟩ → anything, the →* relations must both involve zero steps, so *s'* is *skip* and *σ'* = *σ1* = *σ2* = *τ1* = *τ2*.



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 *[s1‖ ...‖ sn]* is the sequence s1; ...; *sn*. The *sequentialized execution* of the parallel statement is the execution of its sequentialization: We evaluate s1 completely, then s2 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 s1, ..., *sn* are pairwise disjoint, then

$$\frac{\{p\}s_1;\ldots;s_n\{q\}\,s_1,\ldots,s_n\ pairwise\ disjoint}{\{p\}[s_1 \parallel \ldots \parallel s_n]\{q\}}$$

- *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 ∧ 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}*
- Since the order of evaluation the threads doesn't matter for a DPP, we can actually shuffle the order of the statements in the sequentialized program. For example, we can use Example 4's
  - *{T} a := x+1 {a = x+1}; b := x+2 {a = x+1 ∧ b = x+2} {a+1 = b}*

  or swap the two threads:
  - *{T} b := x+2 {b = x+2}; a := x+1 {a = x+1 ∧ b = x+2} {a+1 = b}*