Basics of Parallel Programs

Stefan Muller and Farzaneh Derakhshan, based on material by Jim Sasaki

CS 536: Science of Programming, Fall 2023
Lecture 22

1 Introduction

In this lecture, we extend our language to allow parallelism. We add one more construct to our grammar for the statements

$$s ::= \ldots \mid [\ s \parallel \cdots \parallel s\ ]$$

We say $[\ s_1 \parallel \cdots \parallel s_n\ ]$ is the *parallel composition* of the *threads* $s_1, s_2, \ldots, s_n$. We put the limitation that threads $s_1, s_2, \ldots, s_n$ must be sequential: You can't nest parallel programs[1]. (You can embed parallel programs within larger programs, such as in the body of a loop.)

Example 1.

- Valid: $[\ x := x + \overline{1} \parallel x := x * \overline{2} \parallel y := x^{\overline{2}}\ ]$ is a parallel program with 3 threads.
- Invalid: $[\ x := x + \overline{1} \parallel [\ x := x * \overline{2} \parallel y := x^{\overline{2}}\ ]\ ]$ tries to nest parallel programs.
- Valid: $[\ x := x + \overline{1} \parallel x := x * \overline{2}\ ]; [\ x := x + \overline{1} \parallel y := x^{\overline{2}}\ ]$ has two threads that then join back together and split into 2 threads again.

1.1 Interleaving Execution of Parallel Programs

The parallel execution of sequential threads is by interleaving their execution, i.e., we run multiple threads by interleaving the operational semantics steps for the individual threads. This means that we execute one thread for some number of operational steps, then execute another thread, and so on. Depending on the program and the sequence of interleaving, different final states can be reached, or the program can cause an error in one execution but not in another. For instance, suppose we have $[\ x := x + \overline{1} \parallel x := x * \overline{2}\ ]$ to evaluate. In this case, we would interleave the operational semantics steps of the two threads. One possible execution could be evaluating $x := x + \overline{1}$ first and then $x := x * \overline{2}$, while another possible execution could be evaluating $x := x * \overline{2}$ first and then $x := x + \overline{1}$. Each of these two possible executions can create different final states [2].

Parallel computation and sequential nondeterminism. The difference between parallel composition of threads as in

$$[\ x := x + \overline{1} \parallel x := x * \overline{2}\ ]$$

and the nondeterministic branch as in

$$\mathsf{if}\ T \to x := x + \overline{1} \,\square\, T \to x := x * \overline{2}\ \mathsf{fi}$$

[1] Some languages, including several that Stefan works with, allow you to do this; we might look at that later, but for now we'll say we can't nest.

[2] If you're familiar with writing concurrent programs, you might also know that if we were using a language like C, it would be possible to calculate both $x + 1$ and $x * 2$ before we store either and have one overwrite the other, ending up with either $x + 1$ or $2x$ rather than $2x + 1$ or $2(x + 1)$. In this class, we'll assume that we interleave at the level of operational steps, so calculating the expression and doing the assignment happens atomically (all at once).

is that the nondeterministic branch executes only one of the two assignments, whereas the parallel composition executes both assignments but in an unpredictable order. The sequential nondeterministic branch that simulates the parallel assignments is:

$$\mathsf{if}\ T \to x := x + \overline{1}; x := x * \overline{2} \,\square\, T \to x := x * \overline{2}; x := x + \overline{1}\ \mathsf{fi}.$$

It nondeterministically chooses between the two possible traces of execution for the program. This is a good use of sequential nondeterminism to understand the nondeterminism that results from parallel interleaving. However, it doesn't scale well to large programs. Because of the nondeterminism, re-executions of a parallel program can use different orders. For example, two executions of $\mathsf{while}\ e\ \mathsf{do}\ [\ x := x + \overline{1} \parallel x := x * \overline{2}\ ]\ \mathsf{od}$ can have the same sequence or different sequences of updates to x.

1.2 What is the Challenge?

The main challenge with parallel programs is that their properties can be very different from the behaviors of the individual threads. Each thread can change the state in ways that don't maintain the assumptions used by other threads.

Example 2: Consider the parallel program $[\ x := x + \overline{1} \parallel x := x * \overline{2}\ ]$ with two threads $x := x + \overline{1}$ and $x := x * \overline{2}$. If we consider only the local behavior of each thread we get

$$\vDash \{x = 5\}\ x := x + \overline{1}\ \{x = 6\} \quad \text{and} \quad \vDash \{x = 5\}\ x := x * \overline{2}\ \{x = 10\}$$

but when we consider possible interleaving of both threads we get

$$\vDash \{x = 5\}\ [\ x := x + \overline{1} \parallel x := x * \overline{2}\ ]\ \{x = 11 \vee x = 12\}.$$

The reason for having different behavior between individual threads and their parallel composition is that the interleaving threads can interfere with each other's data. Full interference can be tricky, so we will start with simple, limited parallel programs that don't interact at all. But before we get into that, let's take a closer look at the semantics of parallel programs.

2 Semantics of Parallel Programs

2.1 Small-step Semantics

Recall that to execute the sequential composition $s_1; \ldots; s_n$ for one step, we execute the first statement in the sequence, i.e., s_1, for one step. If s_1 is skip, we dismiss it and continue with the next statement in the sequence, i.e., s_2.

Now, to execute the parallel composition $[\ s_1 \parallel \cdots \parallel s_n\ ]$ for one step, we can choose any one of the statements (nondeterministically) and execute it for one step. The following small-step semantics formalizes this behavior.

$$\frac{\langle s_k, (\sigma, h)\rangle \to \langle s_k', (\sigma_k, h_k)\rangle}{\langle [\ s_1 \parallel \cdots \parallel s_n\ ], (\sigma, h)\rangle \to \langle [\ s_1 \parallel \cdots \parallel s_{k-1} \parallel s_k' \parallel s_{k+1} \parallel \cdots \parallel s_n\ ], (\sigma_k, h_k)\rangle}$$

A completely-executed parallel program, i.e., the one that is terminated, looks like $[\ \mathsf{skip} \parallel \cdots \parallel \mathsf{skip} \parallel \mathsf{skip}\ ]$. We'll treat $[\ \mathsf{skip} \parallel \cdots \parallel \mathsf{skip} \parallel \mathsf{skip}\ ]$ as skip, i.e., we have $[\ \mathsf{skip} \parallel \cdots \parallel \mathsf{skip} \parallel \mathsf{skip}\ ] \equiv \mathsf{skip}$.

Notation. The $\to^*$ notation has the same meaning whether the configurations involved have parallel programs or not: $\to^*$ means $\to^n$ for some $n \geq 0$, and $c_0 \to^n c_n$ means that there is actually a sequence of n+1 configurations, $c_0 \to c_1 \to \ldots \to c_n$ where we've omitted writing the intermediate configurations.

Common Mistake. Writing $\langle [\ \mathsf{skip} \parallel \mathsf{skip}\ ], (\sigma, h)\rangle \to \langle \mathsf{skip}, (\sigma, h)\rangle$. The finished parallel program doesn't take a step. With parallel programs, since $[\ \mathsf{skip} \parallel \mathsf{skip}\ ] \equiv \mathsf{skip}$, we can write

$$\langle [\ \mathsf{skip} \parallel \mathsf{skip}\ ](\sigma, h), \to\rangle^0 \langle \mathsf{skip}, (\sigma, h)\rangle.$$

2.2 Evaluation Graph

As we saw earlier, each configuration can result in multiple executions. We show all possible executions of a configuration by building a graph called the evaluation graph. More formally, the evaluation graph for $\langle s, (\sigma, h)\rangle$ is the directed graph of configurations and evaluation arrows leading from $\langle s, (\sigma, h)\rangle$ such that

- The nodes of the graph are configurations
- An edge from configuration c_0 to configuration c_1 means that c_0 can step to c_1, i.e., $c_0 \to c_1$.
- 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.)

A parallel 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. We call a node *sink* if it does not have any outgoing edges. The sink nodes are terminating configurations, i.e., they cannot take any other steps [3].

2.3 Big-step Semantics

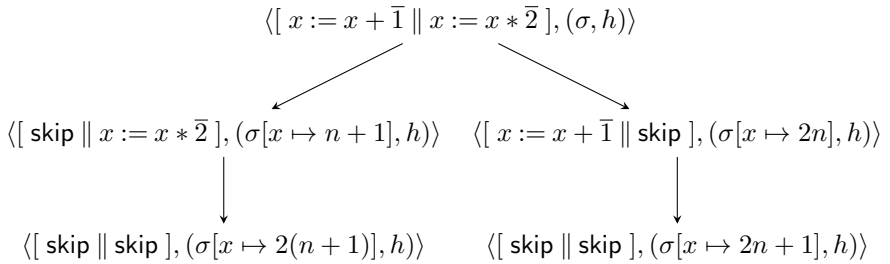
The big-step semantics of a program in a state is the set of all possible terminating states (plus possibly the pseudostate $\perp$) reachable from the initial configuration.

$$\begin{array}{rcll} M(s,(\sigma,h)) & = & \{(\sigma',h') \mid \langle s,(\sigma,h)\rangle \to^* \langle \mathsf{skip},(\sigma',h')\rangle\} & \\ & \cup & \{\perp\} & \text{if } s \text{ can produce a runtime error} \end{array}$$

If all executions of the configuration diverge we get $M(s, (\sigma, h)) = \{\}$.

2.4 Examples

Example 3. The evaluation graph below is for our running example program, starting with a state (σ, h) where $\sigma(x) = n$.



The graph has two sinks (possible final states), so

$$M([\ x := x + \overline{1} \parallel x := x * \overline{2}\ ], (\sigma, h)) = \{(\sigma[x \mapsto 2n + 2], h), (\sigma[x \mapsto 2n + 1], h)\}.$$

Example 4. The evaluation graph in Figure 1 is for $\langle [\ x := v \parallel y := v + \overline{2} \parallel z := v * \overline{2}\ ], (\sigma, h)\rangle$ where $\sigma(v) = n$. We have $M([\ x := v \parallel y := v + \overline{2} \parallel z := v * \overline{2}\ ], (\sigma, h)) = (\sigma[x \mapsto n][y \mapsto n+2][z \mapsto 2n], h)$. Note that even though the program is nondeterministic, it produces the same result regardless of the execution path. Just like with sequential nondeterministic programs, if s is parallel, then $M(s, \sigma)$ *can* have more than one element, but may not.

The reason we end up with one state is that all of the state updates *commute*, i.e., $\sigma[x \mapsto n][y \mapsto n+2] = \sigma[y \mapsto n + 2][x \mapsto n]$, and so on. This isn't an accident; it's because the threads are updating different variables and they read variables that no other thread writes. This guarantees all of the updates commute, and the program is deterministic, and is therefore a really nice property for parallel programs to have!

Example 5. Let the program W be

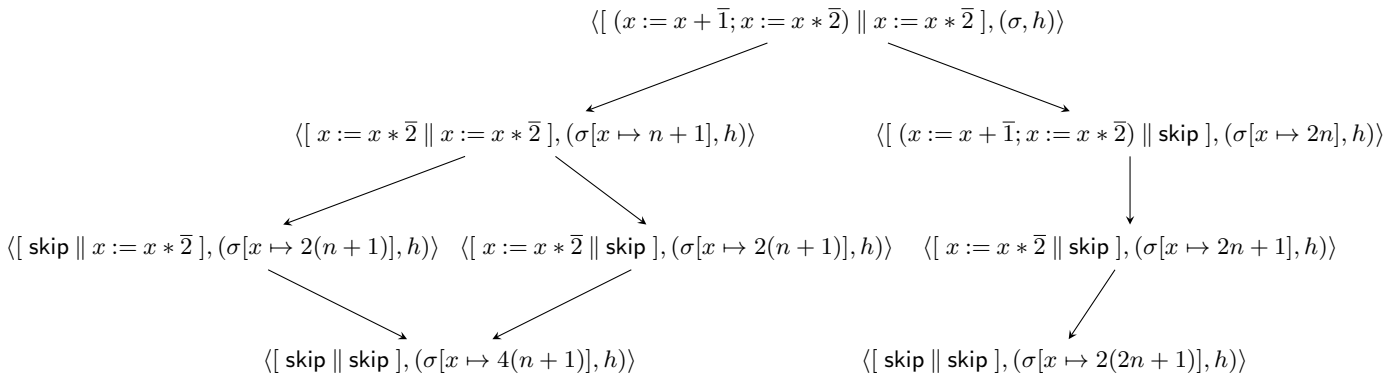
$$W \triangleq x := \overline{0}; \mathsf{while}\ x = \overline{0}\ \mathsf{do}\ [\ x := \overline{0} \parallel x := \overline{1}\ ]\ \mathsf{od}.$$

[3] This observation is not true when we have deadlocks.

Then $M(W, (\sigma, h)) = \{(\sigma[x \mapsto 1], h)\}$. The problem here is possible divergence, but it only happens if we always choose thread 1 when we have to make the nondeterministic choice. This is definitely unfair behavior, but it's allowed because of the unpredictability of our nondeterministic choices. In real life, we would want a fairness mechanism to ensure that all threads get to evaluate once in a while. If each thread is on a separate processor, then the nondeterministic choice corresponds to which processor is fastest.

If the behavior of a program depends on the relative speed of the processors involved, like the termination of this example does, we call this a *race condition*. Generally, this happens when two or more threads access the same variable and at least one of the accesses is a write (like the two threads writing to x in this example). In Example 4, there was no race condition: one thread each writes to x, y and z; all threads read v, but that's fine. People disagree on the exact definition of the term *race condition*; we may or may not go into more detail on this.

Example 6. Let $S = [\,(x := x + \overline{1}; x := x * \overline{2}) \parallel x := x * \overline{2}\,]$. Here we've added another statement to the first thread in Example 2, which increases the number of possible interleavings (though some of the execution paths still result in the same state).



So far, our examples only worked with the state. Next, we see an example of a parallel program that works with the heap.

2.5 Binary trees

A binary tree is a data structure where each element is stored as a separate object in a node. Each node is stored in three consecutive memory locations. The data entry is stored in the first location. The second location stores a reference to the left child of the node, and the third location stores a reference to the right child of the node. A reference, usually called the root, points to the node with the first data entry. If a node does not have a left child, its second location has value *nil*. Similarly, if a node does not have a right child, its third location has value *nil*. A node with neither a left child nor a right child is called a leaf and has the value *nil* in both its second and third locations. See Figure 2.

Example 7. `deleteTree` is a parallel program that works on the tree T from Figure 2 with root value *root* and deallocates every node in the tree.

$$
\begin{array}{lll}
\texttt{deleteTree} \triangleq & & \\
[\quad x_0 := !(root+1); x_1 := !(x_0+1); & \textbf{dispose}(x_1); & \\
 & \textbf{dispose}(x_1+1); & \\
 & \textbf{dispose}(x_1+2); & \\
 & \textbf{dispose}(x_0); & \\
 & \textbf{dispose}(x_0+1); & \\
 & \textbf{dispose}(x_0+2) & \\
 & \textbf{dispose}(root+1) & \\
\| \quad y_0 := !(root+2); y_1 := !(y_0+2); & \textbf{dispose}(y_1); & \\
 & \textbf{dispose}(y_1+1); & \\
 & \textbf{dispose}(y_1+2); & \\
 & \textbf{dispose}(y_0); & \\
 & \textbf{dispose}(y_0+1); & \\
 & \textbf{dispose}(y_0+2) & \\
 & \textbf{dispose}(root+2) \quad]; & \\
\textbf{dispose}(root) & &
\end{array}
$$

The program `deleteTree` has two parallel threads. The first thread in this program deallocates elements of the left subtree of the first node, and the second thread deallocates the elements of the right subtree of the first node. After both threads are done, the program deallocates the first element of the tree. We could also write a sequential program that deallocates all elements of the tree by simply running one of the threads first and the other one next. However, in a concurrent setting[4], where we have multiple cores to run a program, this parallel implementation of `deleteTree` is more efficient than the sequential one.

3 Concurrent separation logic (Limited)

We introduce an extension of Separation logic for partial correctness. The logic allows proofs of parallel programs based on the idea of "ownership ."The idea is that each thread owns part of the state (heap and store), e.g., having the exclusive right to access, update, or deallocate a pointer. Moreover, the ownership can be transferred dynamically between threads. Initially, we start with a limited version of Concurrent Separation Logic (CSL) that deals with programs where threads maintain exclusive ownership throughout their execution without transferring it to other threads. An example of such a program is the deleteTree from Example 7, which deallocates a tree. Here, the first thread deals with the left subtree of T and the second thread with the right subtree of T. This means that the first thread has exclusive ownership of the left subtree, the second thread has exclusive ownership of the right subtree, and they maintain this exclusive ownership.

We add a rule to separation logic to handle parallel statements. We first introduce the rule for a statement with only two threads and then generalize it to n threads:

$$
\frac{\{p_1\}\ s_1\ \{q_1\} \qquad \{p_2\}\ s_2\ \{q_2\}}{\{p_1 * p_2\}\ [\ s_1 \parallel s_2\ ]\ \{q_1 * q_2\}}\ \textsc{par}(2\ \textsc{threads})
$$

with the condition that the free variables of p_1, s_1, and q_1 are not modified by the statement s_2, and vice-versa. This condition, which we call the *disjointness condition*, ensures that s_1 and s_2 work on separate *store variables*. The rule also separates the *heap* into parts that can only be mutated by a single process. By (i) separating the heap and (ii) the disjointness condition on the store variables, the rule states that we can combine the postconditions for each process separately to prove a parallel composition. This approach allows for completely independent reasoning about processes, making it easier to prove processes that do not share storage access.

[4]Concurrency and parallelism often are used interchangeably (by mistake), but they are different, and their difference is subtle. Please read the blog post titled **Parallelism Is Not Concurrency** by Bob Harper.

A more general version (but still limited by restricting ownership transfer) of the concurrency rule which allows for reasoning about multiple threads is:

$$\frac{\forall 1 \leq i \leq n \qquad \{p_i\}\ s_i\ \{q_i\}}{\{p_1 * \cdots * p_n\}\ [\ s_1 \parallel \cdots \parallel s_n\ ]\ \{q_1 * \cdots * q_n\}}\ \textsc{Par}(n \text{ threads})$$

with the disjointness condition that states for every $1 \leq k \leq n$, the free variables of p_k, s_k, and q_k are not modified by any of other threads s_j where $j \neq k$.

3.1 Disjointness, Confluence, and Diamond Property

We mentioned that threads in the `deleteTree` program are completely separate, i.e., we can run the two threads sequentially rather than in parallel and get the exact same final state. In fact, the parallelism in `deleteTree` is innocuous because the two threads do not interfere with each other's execution: (1) Each thread works on a separate part of the heap, i.e., left and right subtrees. (2) If one thread modifies a store variable, that modification cannot be overwritten by the other thread. For example, consider variable x_0, which is modified by the first thread. Since the second thread will never read from x_0, we know the modification won't affect how the second threads execute. This is, in fact, true for all variables that are modified by the first thread: they do not occur in the second thread. Similarly, none of the variables that are modified by the second thread occur in the first thread. The *separation* of the heap and *disjointness* of the store causes all the evaluation paths to end in the same configuration.

Confluence is a property that arises from separation and disjointness of threads. It states that it does not matter how the thread executions have been interleaved; in the end, all executions will converge into a single final state. For example, the execution graph of Example 4 in Figure 1 shows this property as all execution paths converge into a single final state. The program in Example 4 does not work with the heap, i.e., none of the threads touch the heap, thus the separation is obvious. It also has the disjointness property, since the variables modified in one thread do not occur free in other threads: Thread 1 writes on variable x, but x does not occur free in threads 2 and 3. Thread 2 writes on y, but y does not occur free in threads 1 and 2. Thread 3 writes on z, but z does not occur free in threads 1 and 2.

Having a closer look at Figure 1, we can observe that the graph not only has the confluence property but also a stronger property that for every node $\langle s, (\sigma, h)\rangle$ on the graph all pairs of nodes diverging out of the node can be closed into a diamond, i.e., converge in a single step. The property that "the paths can merge back together in one step" is called the *diamond property*. (The execution graph in Example 4 looks like a diamond, hence the name diamond property.)

Note that the inverse of the statement "if a program is disjoint, it is confluent and has the diamond property" is not true. Part of the execution graph (corresponding to the program $[\ x := x * \overline{2} \parallel x := x * \overline{2}\ ]$ in Example 6 has the diamond property, even though these threads are not disjoint.

More formally, we can define the confluence and diamond properties as follows.

Diamond Property: An execution graph has the diamond property iff for any node $\langle s, (\sigma, h)\rangle$ on the graph

$$\begin{array}{c}\textbf{if}\ \langle s, (\sigma, h)\rangle \to \langle s_1, (\sigma_1, h_1)\rangle\ \textbf{and}\ \langle s, (\sigma, h)\rangle \to \langle s_2, (\sigma_2, h_2)\rangle,\ \textbf{then} \\ \textbf{there is a state}\ (\sigma', h')\ \textbf{and a statement}\ s'\ \textbf{such that} \\ \langle s_1', (\sigma_1, h_1)\rangle \to \langle s', (\sigma', h')\rangle\ \textbf{and}\ \langle s_2, (\sigma_2, h_2)\rangle \to \langle s', (\sigma', h')\rangle\end{array}$$

Note the same s' and (σ', h') in both final states.

Confluence Property: An execution graph has the confluence property iff for any node $\langle s, (\sigma, h)\rangle$ on the graph

$$\begin{array}{c}\textbf{if}\ \langle s, (\sigma, h)\rangle \to^* \langle s_1, (\sigma_1, h_1)\rangle\ \textbf{and}\ \langle s, (\sigma, h)\rangle \to^* \langle s_2, (\sigma_2, h_2)\rangle,\ \textbf{then} \\ \textbf{there is a state}\ (\sigma', h')\ \textbf{and a statement}\ s'\ \textbf{such that} \\ \langle s_1', (\sigma_1, h_1)\rangle \to^* \langle s', (\sigma', h')\rangle\ \textbf{and}\ \langle s_2, (\sigma_2, h_2)\rangle \to^* \langle s', (\sigma', h')\rangle\end{array}$$

We can observe that the two properties are very similar, the only difference is that confluence states any two paths will converge *eventually*, while the diamond property ensures that the two paths converge in *exactly one step*. The diamond property is stronger because it implies confluence, but the converse is not true.

We can prove that the execution graphs of disjoint parallel programs are always confluent, i.e., if an execution terminates, it always terminates in a unique state.

3.2 Disjointness test

Now that we realized the significance of the disjointness condition, we provide an algorithm to determine whether two threads are disjoint or not. In particular, for every two threads k and j, the algorithm needs to check whether the free variables of p_k, s_k, and q_k are modified by s_j or not. To do so, we define the set of free variables of p_k, s_k, and q_k sing the function `free`(), i.e., `free`(p_k) `free`(q_k) and `free`(s_k), and the set of variables modifies by s_j using the function `writes`(), i.e., `writes`(s_j). Once we have this algorithm, it will be straightforward to verify the disjointness condition of the parallel rule.

Our algorithm works statically, i.e., at compile-time, since it is used in the condition of a CSL rule. Therefore, it overapproximates whether two threads can interfere with each other. In particular, `writes`(s) only gives an overapproximation of the variables that will be written into by s; not all variables in this set will necessarily be used at runtime. Similarly, not all variables in `free`(s) will affect the final result of the statement s. Failing a disjointedness test simply says we cannot guarantee that thread j interferes with thread k. Without knowing more about the threads and the starting state, we cannot say anything about whether interference in fact, does not 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.

To define the sets of variables that occur free in an expression e and logic formula p, we assume the standard structurally inductive definitions of the sets `free`(e) and `free`(p). We define `reads`(s) as the set of identifiers having a free read occurrence in s, and `writes`(s) as the set of identifiers having a free write occurrence in s. **The set of free variables of a statement, i.e., `free`(s), is defined as `reads`$(s) \cup$ `writes`(s).**

Definition 1. We define `reads`(s) inductively as

$$
\begin{array}{lcl}
\texttt{reads}(x := e) & = & \texttt{free}(e) \\
\texttt{reads}(x :=\!!e) & = & \texttt{free}(e) \\
\texttt{reads}(i := \mathbf{cons}(e_1, \cdots, e_n)) & = & \texttt{free}(e_1) \cup \cdots \cup \texttt{free}(e_n) \\
\texttt{reads}(!e_1 := e_2) & = & \texttt{free}(e_1) \cup \texttt{free}(e_2) \\
\texttt{reads}(\mathbf{dispose}(e)) & = & \texttt{free}(e) \\
\texttt{reads}([\ s_1 \parallel s_2\ ]) & = & \texttt{reads}(s_1) \cup \texttt{reads}(s_2) \\
\texttt{reads}(s_1; s_2) & = & \texttt{reads}(s_1) \cup \texttt{reads}(s_2) \\
\texttt{reads}(\mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\ \mathsf{fi}) & = & \texttt{free}(e) \cup \texttt{reads}(s_1) \cup \texttt{reads}(s_2) \\
\texttt{reads}(\mathsf{while}\ e\ \mathsf{do}\ s\ \mathsf{od}) & = & \texttt{free}(e) \cup \texttt{reads}(s).
\end{array}
$$

In other words, `reads`(s) is the set of variables that appear on the right-hand side of statements in s.

Definition 2. We define `writes`(s) inductively as

$$
\begin{array}{lcl}
\texttt{writes}(x := e) & = & \{x\} \\
\texttt{writes}(x :=\!!e) & = & \{x\} \\
\texttt{writes}(i := \mathbf{cons}(e_1, \cdots, e_n)) & = & \{x\} \\
\texttt{writes}(!e_1 := e_2) & = & \{\} \\
\texttt{writes}(\mathbf{dispose}(e)) & = & \{\} \\
\texttt{writes}([\ s_1 \parallel s_2\ ]) & = & \texttt{writes}(s_1) \cup \texttt{writes}(s_2) \\
\texttt{writes}(s_1; s_2) & = & \texttt{writes}(s_1) \cup \texttt{writes}(s_2) \\
\texttt{writes}(\mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\ \mathsf{fi}) & = & \texttt{writes}(s_1) \cup \texttt{writes}(s_2) \\
\texttt{writes}(\mathsf{while}\ e\ \mathsf{do}\ s\ \mathsf{od}) & = & \texttt{writes}(s)
\end{array}
$$

In other words, `writes`(s) is the set of variables that appear on the left-hand side of assignments in s. In particular, note that the set is empty for mutation and deallocation of the heap, since these two statements do not write to a store variable.

Exercise 1. Calculare the write set of the first thread and the free set of the second thread in `deleteTree`, i.e.,

$$
\begin{array}{c}
\texttt{writes}(x_0 :=\!!(\mathit{root} + 1); x_1 :=\!!(x_0 + 1); \mathbf{dispose}(x_1); \mathbf{dispose}(x_1 + 1); \mathbf{dispose}(x_1 + 2); \\
\mathbf{dispose}(x_0); \mathbf{dispose}(x_0 + 1); \mathbf{dispose}(x_0 + 2)\mathbf{dispose}(\mathit{root} + 1))
\end{array}
$$

$$
\begin{array}{c}
\texttt{free}(y_0 :=\!!(\mathit{root} + 2); y_1 :=\!!(y_0 + 2); \mathbf{dispose}(y_1); \mathbf{dispose}(y_1 + 1); \mathbf{dispose}(y_1 + 2); \\
\mathbf{dispose}(y_0); \mathbf{dispose}(y_0 + 1); \mathbf{dispose}(y_0 + 2)).
\end{array}
$$

What is the intersection of these two sets?

The concurrency rule - revisited Using the above definitions we can rewrite the concurrency rule as

$$\frac{\forall 1 \le i \le n \qquad \{p_i\}\ s_i\ \{q_i\}}{\{p_1 * \cdots * p_n\}\ [\ s_1 \parallel \cdots \parallel s_n\ ]\ \{q_1 * \cdots * q_n\}}\ \textsc{Par}(n \text{ threads})$$

with the disjointness condition that states for every $1 \le k, j \le n$ with $j \neq k$, we have

$$(\texttt{free}(p_k) \cup \texttt{free}(s_k) \cup \texttt{free}(q_k)) \cap \texttt{writes}(s_j) = \emptyset.$$

3.3 Proof outline

When writing the proof outlines, it is common to illustrate the premises of the concurrency rule, i.e., the first and second threads annotated with pre and post conditions, in left and right, respectively. We put the overall precondition $(p_1 * p_2)$ at the beginning, followed by the postcondition at the end. Figure 3.3 shows a partial proof outline for Exapmle 6. Note the application of the frame rule on $root \mapsto a$. The tree predicate $tree(T, j)$ is defined similar to the list predicate from Lecture 20. It states that tree T and its root at address j. We can define it by (structural) induction as:

$$\texttt{tree}(\epsilon, i) \triangleq i = nil \quad \textit{and} \quad \texttt{tree}(a \bullet T_\ell \bullet T_r, i) \triangleq i \mapsto a, j_\ell, j_r\ *\ \texttt{tree}(T_\ell, j_\ell) * \texttt{tree}(T_r, j_r)$$

$a \bullet T_\ell \bullet T_2$ describes a tree with root value a, left subtree T_1, and right subtree T_2. For example the tree in Figure 2(b) can be written as $2 \bullet (45 \bullet (50 \bullet \epsilon)) \bullet (30 \bullet (\epsilon \bullet 1))$.

3.4 Next?

While the disjointness and separation conditions make reasoning simple, if CSL had only been able to reason about disjoint concurrency, where there is no interthread interaction, then it would have rightly been considered rather restrictive. We can overcome this limitation by utilizing ownership of resources and the resource invariant. If you are interested in learning more about this, come talk to us!

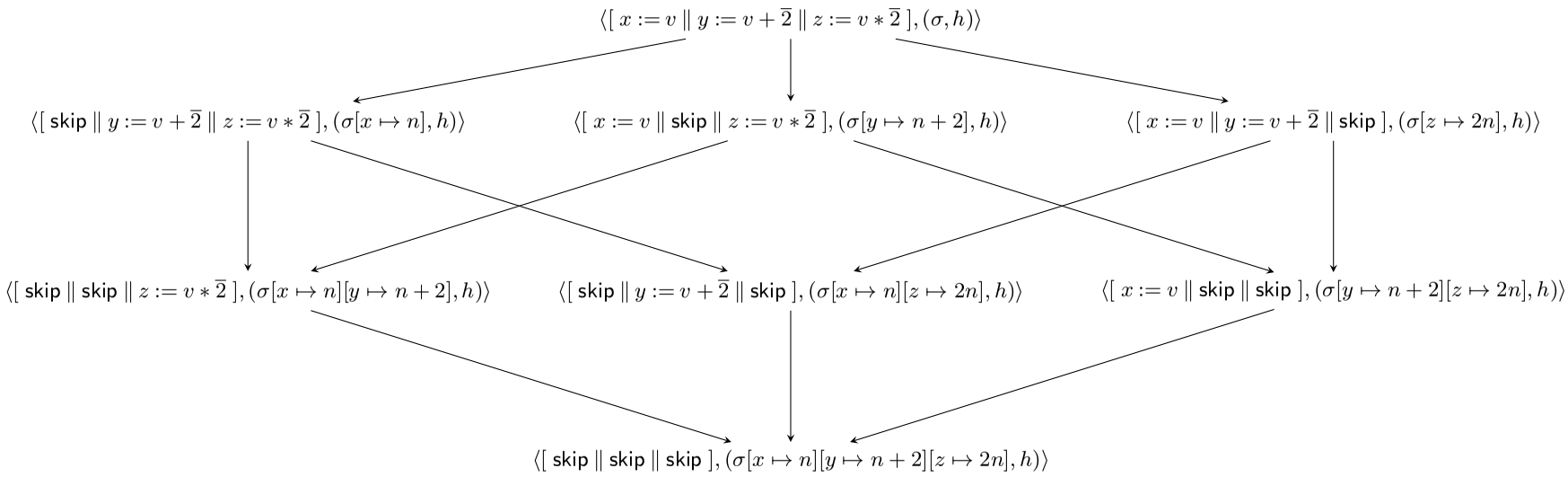


Figure 1: Evaluation graph for Example 4.

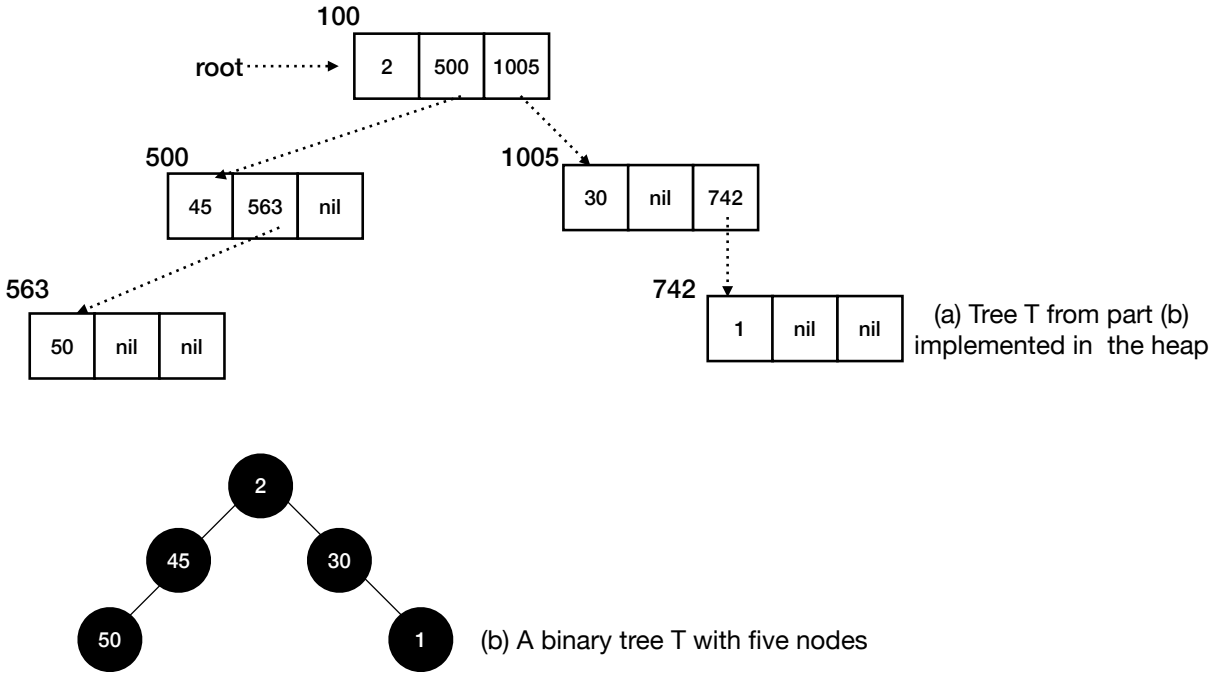


Figure 2: Binary tree structure

$$\{root \mapsto a, j_\ell, j_r * tree(T_\ell, j_\ell) * tree(T_r, j_r)\}$$
$$\{root \mapsto a * root + 1 \mapsto j_\ell * root + 2 \mapsto j_r * tree(T_\ell, j_\ell) * tree(T_r, j_r)\}$$
$$\{root + 1 \mapsto j_\ell * root + 2 \mapsto j_r * tree(T_\ell, j_\ell) * tree(T_r, j_r)\} \qquad (\textbf{FRAME})$$

$\{root + 1 \mapsto j_\ell * tree(T_\ell, j_\ell)\}$		$\{root + 2 \mapsto j_r * tree(T_r, j_r)\}$
$x_0 :=!(root + 1);$		$y_0 :=!(root + 2);$
$x_1 :=!(x_0 + 1);$		$y_1 :=!(y_0 + 2);$
$\textbf{dispose}(x_1);$		$\textbf{dispose}(y_1);$
$\textbf{dispose}(x_1 + 1);$		$\textbf{dispose}(y_1 + 1);$
$\textbf{dispose}(x_1 + 2);$	$\parallel$	$\textbf{dispose}(y_1 + 2);$
$\textbf{dispose}(x_0);$		$\textbf{dispose}(y_0);$
$\textbf{dispose}(x_0 + 1);$		$\textbf{dispose}(y_0 + 1);$
$\textbf{dispose}(x_0 + 2);$		$\textbf{dispose}(y_0 + 2);$
$\textbf{dispose}(root + 1)$		$\textbf{dispose}(root + 2)$
$\{\textbf{emp}\}$		$\{\textbf{emp}\}$

$$\{\textbf{emp} * \textbf{emp}\}$$
$$\{root \mapsto a * \textbf{emp} * \textbf{emp}\} \qquad (\textbf{FRAME})$$
$$\textbf{dispose}(root)$$
$$\{\textbf{emp} * \textbf{emp} * \textbf{emp}\}$$
$$\{\textbf{emp}\}$$

Figure 3: partial proof outline for `deleteTree`