Example (last time) – “race condition”
Big-step semantics of parallel programs

\[ M(S, (σ, h)) = \{(σ', h')| ⟨s, (σ, h)⟩ \rightarrow^* ⟨skip, (σ', h')⟩\} \]
(\(∪\{⊥\} if S can raise a runtime error)\)

\[ M(S, (σ, h)) = {} if all execution paths diverge \]
Example (last time)

\[
M([ x := x + \overline{1} || x := x \cdot \overline{2} ], (\sigma, h)) = \{(\sigma[x \mapsto 2n + 2], h), (\sigma[x \mapsto 2n + 1], h)\}.
\]
Example 5

\[ W \triangleq x := 0; \text{while } x = 0 \text{ do } \begin{cases} x := 0, & \text{if } x = 0 \\ x := 1, & \text{if } x = 1 \end{cases} \text{ od} \]

\[ M(W, (\sigma, h)) = \{x = 1, h\} \]

\[
\langle \text{while } x = 0 \text{ do } [ x := 0 \parallel x := 1], \{x = 0\}, h \rangle
\]

\[
\langle \text{while } x = 0 \text{ do } [ x := 0 \parallel x := 1], \{x = 0\}, h \rangle
\]

\[
\langle \text{while } x = 0 \text{ do } [ x := 0 \parallel x := 1], \{x = 0\}, h \rangle
\]

\[
\langle \text{skip, } \{x = 1\}, h \rangle
\]

\[
\langle \text{while } x = 0 \text{ do } [ x := 0 \parallel x := 1], \{x = 0\}, h \rangle
\]

\[
\langle \text{while } x = 0 \text{ do } [ x := 0 \parallel x := 1], \{x = 0\}, h \rangle
\]

\[
\langle \text{while } x = 0 \text{ do } [ x := 0 \parallel x := 1], \{x = 0\}, h \rangle
\]

\[
\ldots
\]
Example 4

No race condition! What happened?

\[ (\{ x := v \parallel y := v + 2 \parallel z := v \cdot 2 \}, (\sigma, h)) \]
Binary trees
Delete a binary tree (in parallel)

\[
\text{deleteTree} \triangleq \\
[\quad x_0 := !(\text{root} + 1); x_1 := !(x_0 + 1) ; \\
\quad \| \quad y_0 := !(\text{root} + 2); y_1 := !(y_0 + 2) ; \\
\quad \| \quad \text{dispose}(\text{root}) \\
\quad \text{dispose}(x_1) ; \\
\quad \text{dispose}(x_1 + 1) ; \\
\quad \text{dispose}(x_1 + 2) ; \\
\quad \text{dispose}(x_0) ; \\
\quad \text{dispose}(x_0 + 1) ; \\
\quad \text{dispose}(x_0 + 2) ; \\
\quad \text{dispose}(\text{root} + 1) ; \\
\quad \text{dispose}(y_1) ; \\
\quad \text{dispose}(y_1 + 1) ; \\
\quad \text{dispose}(y_1 + 2) ; \\
\quad \text{dispose}(y_0) ; \\
\quad \text{dispose}(y_0 + 1) ; \\
\quad \text{dispose}(y_0 + 2) ; \\
\quad \text{dispose}(\text{root} + 2) ] ;
\]
Parallel rule

• For 2 threads, if threads are “disjoint” (p1, s1, and q1 aren’t modified by s2 and vice versa)

\[
\frac{\{p_1\} \ s_1 \ \{q_1\} \quad \{p_2\} \ s_2 \ \{q_2\}}{\{p_1 \cdot p_2\} \ [ \ s_1 \ || \ s_2 \ ] \ \{q_1 \cdot q_2\}} \quad \text{PAR(2 THREADS)}
\]

• For n threads (assuming all n are disjoint with all others)

\[
\forall 1 \leq i \leq n \quad \frac{\{p_i\} \ s_i \ \{q_i\}}{\{p_1 \cdot \cdots \cdot p_n\} \ [ \ s_1 \ || \ \cdots \ || \ s_n \ ] \ \{q_1 \cdot \cdots \cdot q_n\}} \quad \text{PAR(n THREADS)}
\]
Confluence and the diamond property

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

\[
\text{if} \langle s, (\sigma, h) \rangle \rightarrow \langle s_1, (\sigma_1, h_1) \rangle \text{ and } \langle s, (\sigma, h) \rangle \rightarrow \langle s_2, (\sigma_2, h_2) \rangle, \text{ then}
\]

\[
\text{there is a state } (\sigma', h') \text{ and a statement } s' \text{ such that}
\]

\[
\langle s_1, (\sigma_1, h_1) \rangle \rightarrow \langle s', (\sigma', h') \rangle \text{ and } \langle s_2, (\sigma_2, h_2) \rangle \rightarrow \langle s', (\sigma', h') \rangle
\]

Note the same \( s' \) and \( (\sigma', h') \) 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

\[
\text{if} \langle s, (\sigma, h) \rangle \rightarrow^* \langle s_1, (\sigma_1, h_1) \rangle \text{ and } \langle s, (\sigma, h) \rangle \rightarrow^* \langle s_2, (\sigma_2, h_2) \rangle, \text{ then}
\]

\[
\text{there is a state } (\sigma', h') \text{ and a statement } s' \text{ such that}
\]

\[
\langle s_1, (\sigma_1, h_1) \rangle \rightarrow^* \langle s', (\sigma', h') \rangle \text{ and } \langle s_2, (\sigma_2, h_2) \rangle \rightarrow^* \langle s', (\sigma', h') \rangle
\]
Making disjointness formal

• Threads $i$ and $j$ are disjoint if

$$\left( \text{free}(s_i) \cup \text{free}(p_i) \cup \text{free}(q_i) \right) \cap \text{writes}(s_j) = \{\}$$

Variables $s_i$ might read or write

Free variables of the condition
Making disjointness formal: reads

\[
\begin{align*}
\text{reads}(x := e) &= \text{free}(e) \\
\text{reads}(x := !e) &= \text{free}(e) \\
\text{reads}(i := \text{cons}(e_1, \cdots, e_n)) &= \text{free}(e_1) \cup \cdots \cup \text{free}(e_n) \\
\text{reads}(!e_1 := e_2) &= \text{free}(e_1) \cup \text{free}(e_2) \\
\text{reads}(&\text{dispose}(e)) = \text{free}(e) \\
\text{reads}([s_1 \parallel s_2]) &= \text{reads}(s_1) \cup \text{reads}(s_2) \\
\text{reads}(s_1; s_2) &= \text{reads}(s_1) \cup \text{reads}(s_2) \\
\text{reads}(\text{if } e \text{ then } s_1 \text{ else } s_2 \text{ fi}) &= \text{free}(e) \cup \text{reads}(s_1) \cup \text{reads}(s_2) \\
\text{reads}(\text{while } e \text{ do } s \text{ od}) &= \text{free}(e) \cup \text{reads}(s).
\end{align*}
\]
Making disjointness formal: writes

\[
\begin{align*}
\text{writes}(x := e) &= \{x\} \\
\text{writes}(x := !e) &= \{x\} \\
\text{writes}(i := \text{cons}(e_1, \cdots, e_n)) &= \{x\} \\
\text{writes}(!e_1 := e_2) &= \{\} \\
\text{writes}((\text{dispose}(e))) &= \{\} \\
\text{writes}([s_1 \| s_2]) &= \text{writes}(s_1) \cup \text{writes}(s_2) \\
\text{writes}(s_1 ; s_2) &= \text{writes}(s_1) \cup \text{writes}(s_2) \\
\text{writes}(\text{if } e \text{ then } s_1 \text{ else } s_2 \text{ fi}) &= \text{writes}(s_1) \cup \text{writes}(s_2) \\
\text{writeoc}(\text{while } e \text{ do } s \text{ od}) &= \text{writeoc}(s)
\end{align*}
\]
\{ root \mapsto a, j_\ell, j_r \ast \text{tree}(T_\ell, j_\ell) \ast \text{tree}(T_r, j_r) \} \\
\{ root \mapsto a \ast root + 1 \mapsto j_\ell \ast root + 2 \mapsto j_r \ast \text{tree}(T_\ell, j_\ell) \ast \text{tree}(T_r, j_r) \} \\
\{ root + 1 \mapsto j_\ell \ast root + 2 \mapsto j_r \ast \text{tree}(T_\ell, j_\ell) \ast \text{tree}(T_r, j_r) \} \\
\{ root + 2 \mapsto j_r \ast \text{tree}(T_r, j_r) \} \\
\}\\n
x_0 := !(root + 1); \\
x_1 := !(x_0 + 1); \\
dispose(x_1); \\
dispose(x_1 + 1); \\
dispose(x_1 + 2); \\
dispose(x_0); \\
dispose(x_0 + 1); \\
dispose(x_0 + 2); \\
dispose(root + 1) \\
\{ \text{emp} \} \\
\{ \text{root} \mapsto a \ast \text{emp} \ast \text{emp} \} \\
dispose(root) \\
\{ \text{emp} \ast \text{emp} \ast \text{emp} \} \\
\{ \text{emp} \} \\
\}
Important Dates

• Thursday, 11/30 11:59pm: HW7 Due
• Thursday, 11/30 11:59pm: Extra credit (HW/midterm redos) due
  • NO LATE DAYS!
• Saturday, 12/2 11:59pm: HW7 Due (w/ 2 late days)
  • No extensions, because...
• Sunday, 12/3: HW7 Solutions posted
• TBA (soon): Review session(s)
• Tuesday, 12/5 2-4pm: Final exam
Final: 12/5 2-4pm

• Rooms:
  • Section 1 (in-person students): WH 113
  • Sections 2-3 (PhD and online): PH 131
  • Important: Make sure you go to the right room

• Seats will be assigned. Come early to find your seat!
  • Seat assignments will be posted on Blackboard, like for the midterm

• Section 03: Let me know by Friday if you’re not taking the exam in person and haven’t already.
Content

• All lectures (including this week)
• All HWs
• Roughly 1/3 material from before the midterm, 2/3 material since the midterm
Format

• 5-10 short answer

• 2 programs w/ loops to do **full** proof (Hilbert or full proof outline) + termination – marked Proof A and Proof B
  • You supply loop invariant, bound, full proof outline
  • Do **one** (your choice)
  • If you do both, we will choose one to grade nondeterministically

• ~4 other longer answer (possibly multi-part) questions

• Total: 100 points (good rule of thumb: 1 point = 1 minute)
Provided resources

Everything from midterm, plus:

• Additional IMP semantics:
  • Small- and big-step semantics for nondeterminism
  • Small-step semantics for parallelism

• Rules for simplifying “if e then e else e” expressions

• Algorithm for expanding proof outlines

• Resource (heap) logic laws

• Separation logic inference rules
Allowed:
• **Four (4)** (double-sided) 8.5x11” sheets of notes
  • Content: anything you want
• Blue or black pen *or pencil*

Not allowed:
• More notes, books, laptops, phones, ...
• Green, purple, red, etc., pen
• Anything else (unless approved through disability accommodations)
Practice/Review

• Practice exam posted on Blackboard today/tomorrow
• Same rough format as exam (no guarantees on topic coverage, timing, difficulty, etc.)

• Additional practice questions posted over the weekend
  • Made possible by viewers like you

• Review session(s) TBA (probably Friday + Monday)
Program Verification

Formally checking that a program is **correct**

Usually: that it meets a *specification*

- gives the right answer
- doesn’t take too long
- has the right *effects*
- has the right security properties
## What we’ve seen

<table>
<thead>
<tr>
<th>Partial Correctness</th>
<th>Total Correctness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop-free, det., seq.</td>
<td>Lec. 7-13</td>
</tr>
<tr>
<td>Loops</td>
<td>Lec. 14-15</td>
</tr>
<tr>
<td>Pointers</td>
<td>Lec. 19</td>
</tr>
<tr>
<td>Nondeterminism</td>
<td>Lec. 21</td>
</tr>
<tr>
<td>Parallelism</td>
<td>Lec. 22-23</td>
</tr>
</tbody>
</table>
### Where to go from here

<table>
<thead>
<tr>
<th></th>
<th>Partial Correctness</th>
<th>Total Correctness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop-free, det., seq.</td>
<td></td>
<td>Lec. 7-13</td>
</tr>
<tr>
<td>Loops</td>
<td>Lec. 14-15</td>
<td>Lec. 17</td>
</tr>
<tr>
<td>Pointers</td>
<td>Lec. 19</td>
<td></td>
</tr>
<tr>
<td>Nondeterminism</td>
<td></td>
<td>Lec. 21</td>
</tr>
<tr>
<td>Parallelism</td>
<td></td>
<td>Lec. 22-23</td>
</tr>
</tbody>
</table>

Quantitative properties, security, etc.
Some things are important enough to fully verify

• CompCert – formally verified C compiler
Or, if you don’t fully verify your whole codebase...

- Program to a specification
- Use assertions (kinda like a proof outline if you squint!)
- Think about loop invariants and bounds
- Informally verify important pieces in your head
But there are other ways of verifying programs too...

Static types can be seen as a form of verification

- **OCaml**
  \[
  \text{sort} : \text{int list} \rightarrow \text{int list}
  \]
  - Takes an integer list and returns an integer list.
  - Valid: \( \text{sort}([8;2;1;6;3]) = [8;2;1;6;3] \)
  - Valid: \( \text{sort}([8;2;1;6;3]) = [10;11;12] \)

- **Coq**
  \[
  \text{sort} : \forall (l1 : \text{list int}), \exists (l2 : \text{int list}),\n  \quad \text{Sorted } l2 \land \text{Permutation } l1 \ l2
  \]
  - Takes an integer list and returns a sorted permutation of it.
  - Valid: \( \text{sort}([8;2;1;6;3]) = [1;2;3;6;8] \)
  - ... and nothing else

Static types can be seen as a form of verification

... but that’s a whole other class
What next?

• CS534: Types and Programming Languages
  • First offering: Spring 2024!
  • Also meets MS theory requirement
  • Prerequisite: CS430

• CS443: Compiler Construction
  • Fall 2024, maybe?

• CS440: Programming Languages and Translators
  • Semantics, types, interpreters
If you really like this stuff...

• Spring 2024 Programming Languages reading group
  • Details to come