CS536 – Concurrent Separation Logic and Wrap-up

November 29, 2023

Example (last time) – “race condition”

$$\begin{tikzcd}[column sep=small]
& \langle [\, x := x + \overline{1} \parallel x := x * \overline{2} \,], (\sigma, h) \rangle \arrow[dl] \arrow[dr] & \\
\langle [\, \mathsf{skip} \parallel x := x * \overline{2} \,], (\sigma[x \mapsto n + 1], h) \rangle \arrow[d] & & \langle [\, x := x + \overline{1} \parallel \mathsf{skip} \,], (\sigma[x \mapsto 2n], h) \rangle \arrow[d] \\
\langle [\, \mathsf{skip} \parallel \mathsf{skip} \,], (\sigma[x \mapsto 2(n + 1)], h) \rangle & & \langle [\, \mathsf{skip} \parallel \mathsf{skip} \,], (\sigma[x \mapsto 2n + 1], h) \rangle
\end{tikzcd}$$

Big-step semantics of parallel programs

- $M\big(S, (\sigma, h)\big) = \ \{(\sigma', h') | \langle s, (\sigma, h)\rangle \rightarrow^* \langle skip, (\sigma', h')\rangle\}$
  ($\cup$ $\{\perp\}$ if S can raise a runtime error)
- $M\big(S, (\sigma, h)\big) = \{\}$ if *all execution paths* diverge

Example (last time)

$$
\begin{tikzcd}
& \langle [\, x := x + \overline{1} \parallel x := x * \overline{2} \,], (\sigma, h) \rangle \arrow[dl] \arrow[dr] & \\
\langle [\, \mathsf{skip} \parallel x := x * \overline{2} \,], (\sigma[x \mapsto n + 1], h) \rangle \arrow[d] & & \langle [\, x := x + \overline{1} \parallel \mathsf{skip} \,], (\sigma[x \mapsto 2n], h) \rangle \arrow[d] \\
\langle [\, \mathsf{skip} \parallel \mathsf{skip} \,], (\sigma[x \mapsto 2(n + 1)], h) \rangle & & \langle [\, \mathsf{skip} \parallel \mathsf{skip} \,], (\sigma[x \mapsto 2n + 1], h) \rangle
\end{tikzcd}
$$

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

Example 5

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

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

$$\langle \mathit{while}\ x = 0\ \mathit{do}\ [\, x \coloneqq 0 \parallel x \coloneqq 1], (\{x = 0\}, h) \rangle$$

$$\langle \mathit{while}\ x = 0\ \mathit{do}\ [\, x \coloneqq 0 \parallel x \coloneqq 1], (\{x = 0\}, h) \rangle \qquad \langle \mathit{while}\ x = 0\ \mathit{do}\ [\, x \coloneqq 0 \parallel x \coloneqq 1], (\{x = 1\}, h) \rangle$$

$$\langle \mathit{while}\ x = 0\ \mathit{do}\ [\, x \coloneqq 0 \parallel x \coloneqq 1], (\{x = 0\}, h) \rangle \qquad \langle \mathit{skip}, (\{x = 1\}, h) \rangle$$

$$\langle \mathit{while}\ x = 0\ \mathit{do}\ [\, x \coloneqq 0 \parallel x \coloneqq 1], (\{x = 0\}, h) \rangle$$

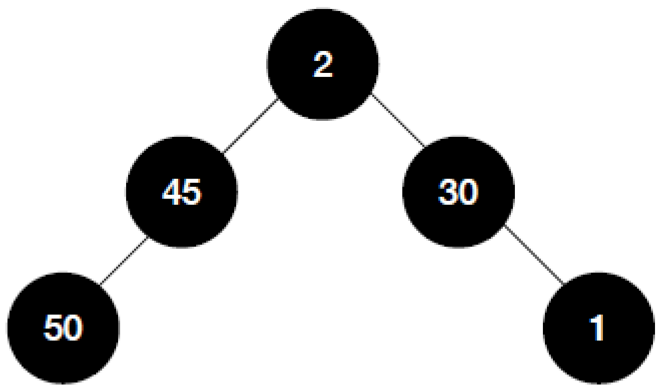
…

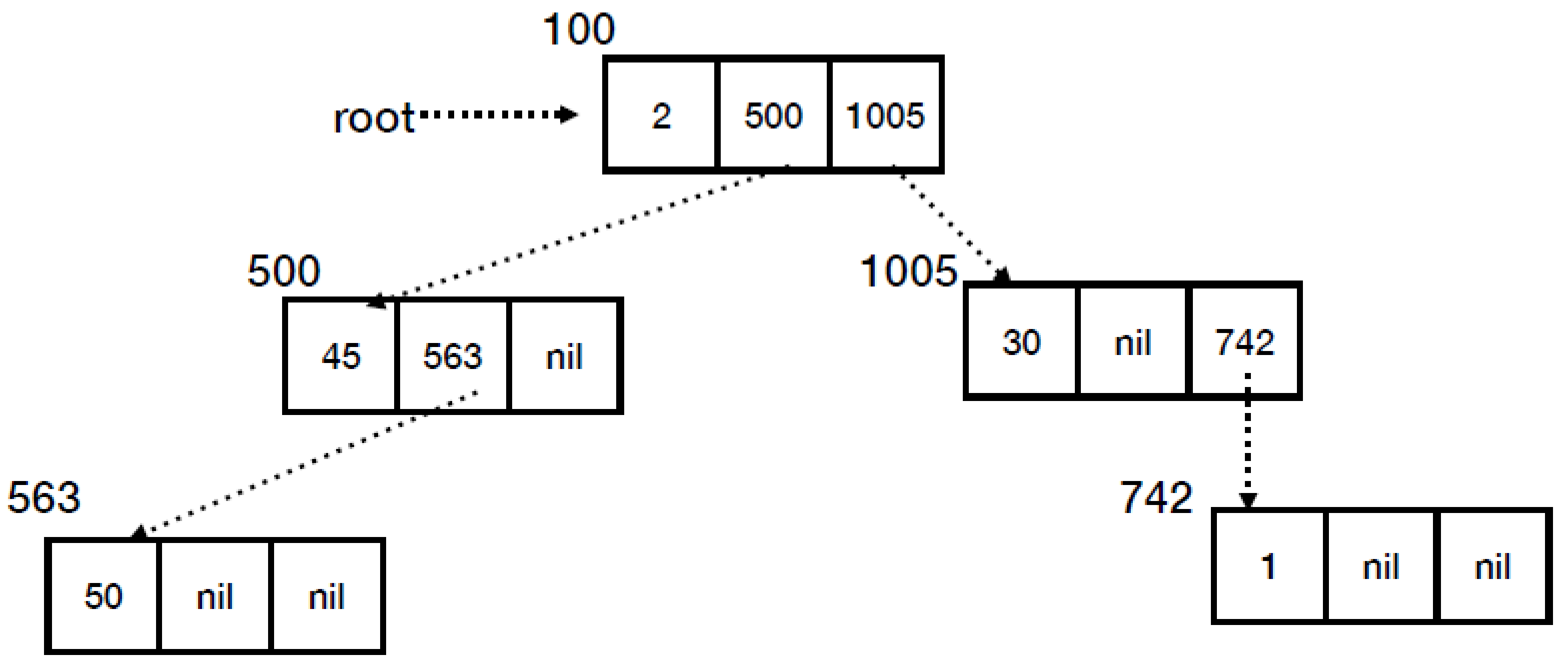
Example 4

No race condition! What happened?

$$\langle [\, x := v \parallel y := v + \overline{2} \parallel z := v * \overline{2} \,], (\sigma, h) \rangle$$

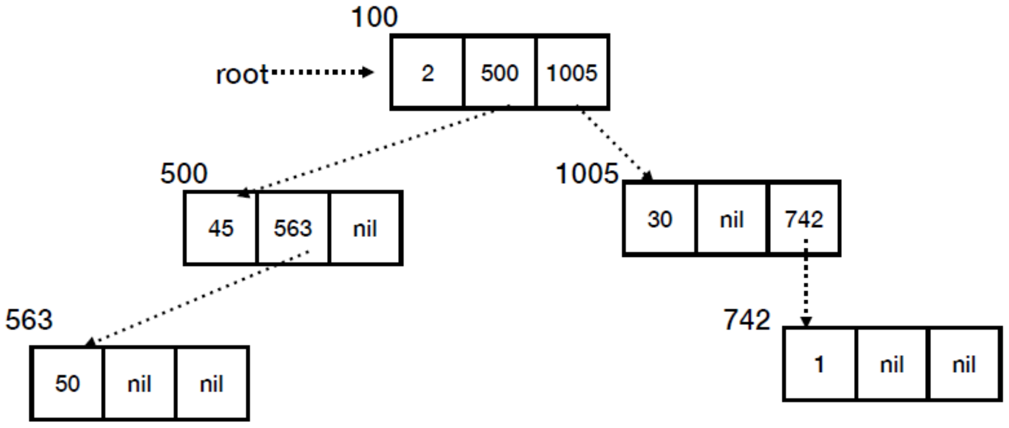
Binary trees





Delete a binary tree (in parallel)

$$
\begin{array}{lll}
\multicolumn{3}{l}{\texttt{deleteTree} \triangleq} \\
[& x_0 := !(root+1); x_1 := !(x_0+1); & \mathrm{dispose}(x_1); \\
 & & \mathrm{dispose}(x_1+1); \\
 & & \mathrm{dispose}(x_1+2); \\
 & & \mathrm{dispose}(x_0); \\
 & & \mathrm{dispose}(x_0+1); \\
 & & \mathrm{dispose}(x_0+2) \\
 & & \mathrm{dispose}(root+1) \\
\| & y_0 := !(root+2); y_1 := !(y_0+2); & \mathrm{dispose}(y_1); \\
 & & \mathrm{dispose}(y_1+1); \\
 & & \mathrm{dispose}(y_1+2); \\
 & & \mathrm{dispose}(y_0); \\
 & & \mathrm{dispose}(y_0+1); \\
 & & \mathrm{dispose}(y_0+2) \\
 & & \mathrm{dispose}(root+2) \quad]; \\
\multicolumn{3}{l}{\mathrm{dispose}(root)}
\end{array}
$$



Parallel rule

- For 2 threads, if threads are "disjoint" (p1, s1, and q1 aren't modified by s2 and vice versa)

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

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

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

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

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

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

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

Making disjointness formal

- Threads i and j are disjoint if

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

Variables si might read or write

Free variables of the condition

Making disjointness formal: reads

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

Making disjointness formal: writes

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

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

$$
\begin{array}{lcl}
\{root + 1 \mapsto j_\ell * tree(T_\ell, j_\ell)\} & & \{root + 2 \mapsto j_r * tree(T_r, j_r)\} \\
x_0 :=!(root + 1); & & y_0 :=!(root + 2); \\
x_1 :=!(x_0 + 1); & & y_1 :=!(y_0 + 2); \\
\mathbf{dispose}(x_1); & & \mathbf{dispose}(y_1); \\
\mathbf{dispose}(x_1 + 1); & & \mathbf{dispose}(y_1 + 1); \\
\mathbf{dispose}(x_1 + 2); & \| & \mathbf{dispose}(y_1 + 2); \\
\mathbf{dispose}(x_0); & & \mathbf{dispose}(y_0); \\
\mathbf{dispose}(x_0 + 1); & & \mathbf{dispose}(y_0 + 1); \\
\mathbf{dispose}(x_0 + 2); & & \mathbf{dispose}(y_0 + 2); \\
\mathbf{dispose}(root + 1) & & \mathbf{dispose}(root + 2) \\
\{\mathbf{emp}\} & & \{\mathbf{emp}\}
\end{array}
$$

$$
\begin{array}{c}
\{\mathbf{emp} * \mathbf{emp}\} \\
\{root \mapsto a * \mathbf{emp} * \mathbf{emp}\} \\
\mathbf{dispose}(root) \\
\{\mathbf{emp} * \mathbf{emp} * \mathbf{emp}\} \\
\{\mathbf{emp}\}
\end{array}
\qquad (\textbf{FRAME})
$$

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**

- gives the right answer
- doesn't take too long
- has the right *effects*
- has the right security properties

this course (mostly)

Usually: that it meets a *specification*

What we’ve seen

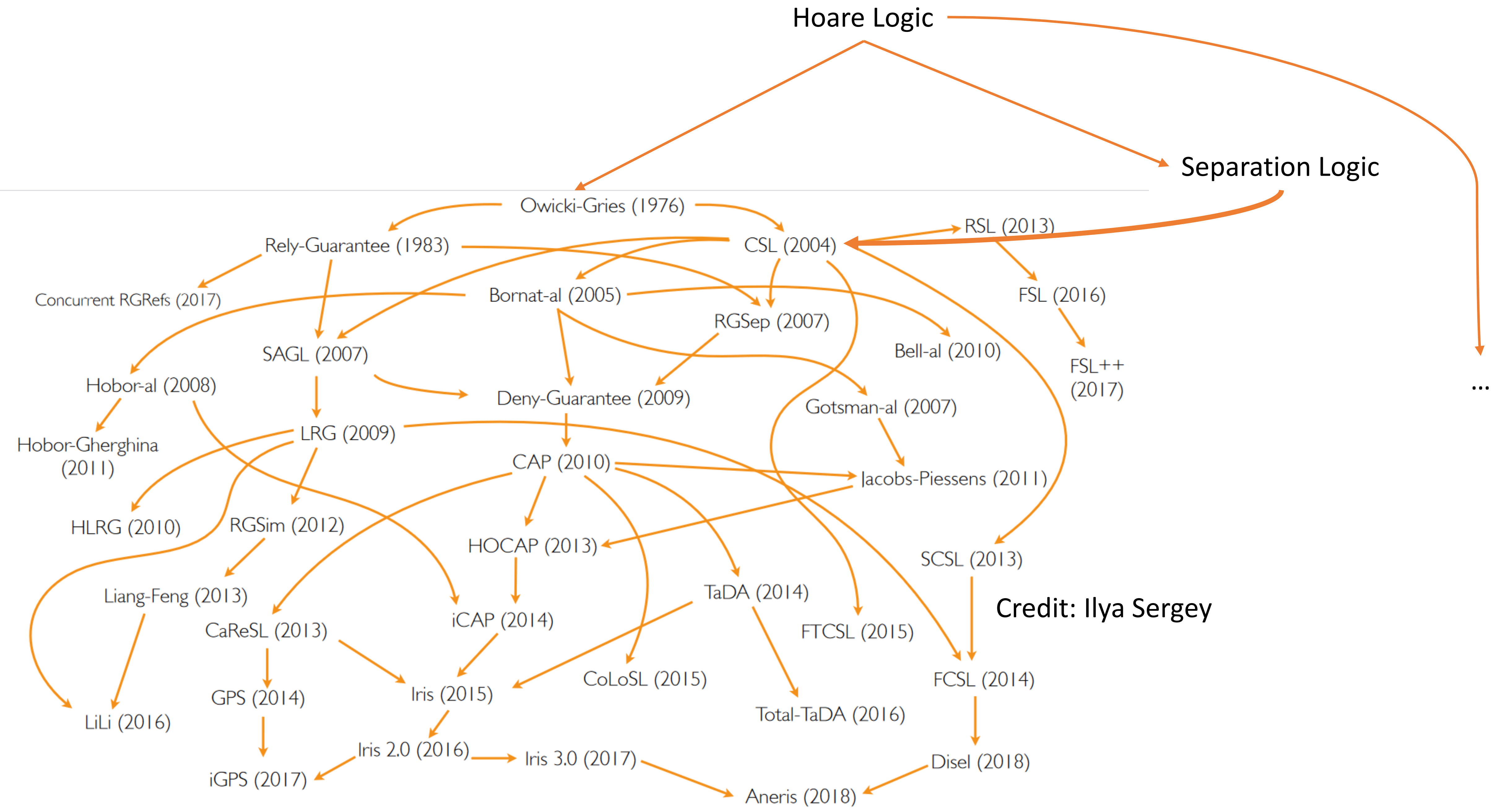
	Partial Correctness	Total Correctness
Loop-free, det., seq.	Lec. 7-13	
Loops	Lec. 14-15	Lec. 17
Pointers	Lec. 19	
Nondeterminism	Lec. 21	
Parallelism	Lec. 22-23	

Where to go from here

Quantitative properties, security, etc.

<table>
<tr><th></th><th>Partial Correctness</th><th>Total Correctness</th></tr>
<tr><td>Loop-free, det., seq.</td><td colspan="2">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 colspan="2">Lec. 21</td></tr>
<tr><td>Parallelism</td><td colspan="2">Lec. 22-23</td></tr>
</table>

Concurrency, I/O, …



Credit: Ilya Sergey

Some things are important enough to *fully* verify

- CompCert – formally verified C compiler

seL4
Security. Performance. Proof.

AIRBUS

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 `sort : int list -> int list`
  - Takes an integer list and returns an integer list.
  - Valid: `sort([8;2;1;6;3]) = [8;2;1;6;3]`
  - Valid: `sort([8;2;1;6;3]) = [10;11;12]`

- Coq
```
sort : forall (l1 : list int), exists (l2: int list),
                     Sorted l2 /\ Permutation l1 l2
```
  - Takes an integer list and returns a sorted permutation of it.
  - Valid: `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