Program Logics for Concurrency
[redacted slides]

Adam Chlipala
MIT

Inspired by https://wiki.mpi-sws.org/star/cpl

6.820
October 30, 2013
Related Reading

More from the course linked on the previous slide! [Also, thanks to Viktor Vafeiadis for suggesting good contents for this lecture.]
The Big Idea of Separation Logic

\{A\} \ c \ \{B\}

Initial heap:

\[ \begin{align*}
A \\
F
\end{align*} \]

Final heap:

\[ \begin{align*}
B \\
F
\end{align*} \]

Frame rule:
It is always OK to add extra cells in the initial heap, and the command \( c \) won't touch them!
Recap: Separation Logic

Predicate constructs: \( \text{emp}, p \mapsto v, [\phi], \exists x. P(x), P \times Q, \ldots \)

Keep all the normal Hoare logic rules and add:

\[
\begin{align*}
\text{x} \not\in \text{freeVars}(e) & \quad \{e \mapsto v\} \ x := \ast e \ \{e \mapsto v \times [x = v]\} \quad \{\exists v. e_1 \mapsto v\} \ast e_1 := e_2 \ \{e_1 \mapsto e_2\} \\
\text{emp} & \quad \{\text{emp}\} \ x := \text{alloc} \ \{x \mapsto 0, 0 \times [x \neq 0]\} \quad \{\exists a, b. e \mapsto a, b\} \ \text{dispose}(e) \ \{\text{emp}\}
\end{align*}
\]

Frame rule:

\[
\begin{align*}
\{A\} \ c \ \{B\} \quad \text{written}(c) \cap \text{freeVars}(F) = \emptyset & \quad \{A \times F\} \ c \ \{B \times F\}
\end{align*}
\]
Example: Merge Sort

```java
{llist(a)}
list sort(list a) {
    if (a == null || a.next == null)
        return a;
    else {
        (b, c) = split(a);
        b = sort(b);
        c = sort(c);
        return merge(b, c);
    }
}
{llist(result)}
```
Example: Splitting a Linked List

{llist(a)}
(list, list) split(list a) {
    if (a == null)
        return (null, null);
    else if (a.next == null)
        return (null, a);
    else {
        (b, c) = split(a.next.next);

        a.next.next = c;
        c = a.next;

        a.next = b;
        b = a;

        return (b, c);
    }
}

{llist(result.1) * llist(result.2)}
Example: Merging Linked Lists

```c
llist(a)* llist(b)
list merge(list a, list b) {
    if (a == null)
        return b;
    else if (b == null)
        return a;
    else if (a.data <= b.data) {
        list c = merge(a.next, b);
        a.next = c;
        return a;
    } else {
        list c = merge(a, b.next);
        b.next = c;
        return b;
    }
}
llist(result)
```
Concurrent Separation Logic [Version 1]

\[
x \not\in \text{freeVars}(e)
\]

\[
\{e \Rightarrow v\} x := ^*e \{e \Rightarrow v \ast [x = v]\}
\]

\[
\{\exists v. \ e_1 \Rightarrow v\} ^*e_1 := e_2 \{e_1 \Rightarrow e_2\}
\]

\[
\{\text{emp}\} x := \text{alloc} \{x \Rightarrow 0, 0 \ast [x \neq 0]\}
\]

\[
\{\exists a, b. \ e \Rightarrow a, b\} \text{dispose}(e) \{\text{emp}\}
\]

Frame rule: \[
\{A\} c \{B\} \quad \text{written}(c) \cap \text{freeVars}(F) = \emptyset
\]

\[
\{A \ast F\} c \{B \ast F\}
\]
A Producer-Consumer Example

list items;

void produce() {
  int n = 0;
  while (true) {
    node x = malloc();
    x.data = n++;
    x.next = items;
    items = x;
  }
}

produce() || consume();

void consume() {
  while (true) {
    while (items == null);
    node x = items.next;
    free(items);
    items = x;
  }
}
Concurrent Separation Logic [Version 2]

\[ \text{x} \notin \text{freeVars(e)} \]

\[
\{ e \leftrightarrow v \} \text{x} := *e \{ e \leftrightarrow v \cdot [x = v] \}
\]

\[
\{ \text{emp} \} \text{x} := \text{alloc} \{ x \leftrightarrow 0, 0 \cdot [x \neq 0] \}
\]

\[
\{ \exists v. e_1 \leftrightarrow v \} *e_1 := e_2 \{ e_1 \leftrightarrow e_2 \}
\]

\[
\{ \exists a, b. e \leftrightarrow a, b \} \text{dispose(e)} \{ \text{emp} \}
\]

\[ \text{Frame rule:} \quad \{ A \} c \{ B \} \quad \text{written(c)} \cap \text{freeVars(F)} = \emptyset \]

\[
\{ A * F \} c \{ B * F \}
\]
A Simpler Example, in More Detail

{global $\rightarrow$ 0}
newlock L in
  (lock(L); ++global; unlock(L))
  || (lock(L); ++global; unlock(L));
{\exists x. global $\rightarrow$ x * [x $\geq$ 0]}
Some Notations for State Relations

For now, forget about separation logic, heaps, etc. We write $P$ for a relation over program states. We write $R$, $G$, $Q$ for relations over pairs of program states, including an initial state and a final state, representing a transition.

Composing relations with a notation suggestive of “$c_1; c_2$”:

$$(R; Q)(\sigma, \sigma') \overset{\text{def}}{=} \exists \sigma''. R(\sigma, \sigma'') \land Q(\sigma'', \sigma')$$

Overload single-state relations $P$ as two-state relations:

$P(\sigma, \sigma') \overset{\text{def}}{=} P(\sigma')$
Rely-Guarantee Judgment

Precondition: what we assume is true before executing

Postcondition: what we prove is true after executing (allowed to refer to both initial and final states)

\{P, R\} \subset \{G, Q\}

Rely: summarizes what other threads are allowed to do

Guarantee: summarizes what this thread is allowed to do
A Quick Word on Atomic Commands

\[\text{atomic}(c) \equiv \text{“run } c \text{ without letting any other threads run simultaneously”}\]

Not available directly in most programming languages, but we can use it to \textbf{model} features that are. E.g.:

\[
\text{lock}(L) \equiv \{
    \text{bool gotIt} = \text{false};
    \text{while (}!\text{gotIt)}
    \quad \text{atomic(}\text{if (}L == 0\text{) }\{
    \quad L = 1;
    \quad \text{gotIt} = \text{true};
    \quad \}
    \}
\}
\]

\[
\text{unlock}(L) \equiv L = 0
\]
Rely-Guarantee Rules

\[
\begin{align*}
\{T, R\} \text{ skip } \{G, \text{ ID}\} \\
T(\sigma) &\overset{\text{def}}{=} \text{true} \\
\text{ID}(\sigma, \sigma') &\overset{\text{def}}{=} \sigma = \sigma'
\end{align*}
\]
Back to the Last Example

{global = 0} atomic(++global) || atomic(++global);
{global > 0}
The Plot Thickens

\{\text{global} = 0\}
\begin{align*}
\text{tmp} &= \text{global}; \quad \text{global} = \text{tmp} + 1 \\
\|\| & (\text{tmp} = \text{global}; \quad \text{global} = \text{tmp} + 1);
\end{align*}
\{\text{global} > 0\}