Reasoning about Resources: an Introduction to Separation Logic

Aquinas Hobor and Martin Henz

Topics

- Extending Hoare Logic
- Hoare Logic Scalability
- Separation Logic
- Resources other than memory
- Reasoning about Concurrency

Hoare Logic so far

- The Hoare Logic presented in class in the past two weeks was very simple:
  - Simple control flow
    - Sequence
    - While
    - If-Then
  - Simple data model
    - Local variables as the only places to store data
    - Only kinds of values are numeric
- This is not very realistic for computation

Features we would like to reason about

- More complex control flow
  - Other kinds of loops (for, do-while, etc.)
  - Goto (well, maybe we don't want to encourage this… but what if we want to reason about assembly code?)
  - Case/Switch analysis
  - Exceptions
  - Functions

Features we would like to reason about

- More complex data
  - Memory
  - Different kinds of data (strings, arrays, records, objects, …)
  - Input / Output
- Hybrid features
  - Function pointers, …

The obvious problem

- As we continue to add features, the logic becomes increasingly complex, both to use and to prove sound.

$$\frac{\forall\, x{:}\,\tau.\ (P\ x) = (\Box\ (P\ x)) \qquad \forall\, x : \tau.\ (Q\ x) = (\Box\ (Q\ x)) \qquad G \vDash \top * f :_{\pi} \tau\ \{P\}\ \{Q\} \qquad G = \bigcirc\ G}{G, R, B \vdash \{P(v_i)\}\ \texttt{call}\ f\ (v_i)\ \{Q(v_i)\}}$$

This rule comes from a language called C minor, which has a number of these features.

The obvious problem

- As we continue to add features, the logic becomes increasingly complex, both to use and to prove sound.

$$\frac{\forall\, x{:}\tau.\ (P\ x) = (\Box\ (P\ x)) \qquad \forall\, x : \tau.\ (Q\ x) = (\Box\ (Q\ x)) \qquad G \vDash \top * f :_{\pi} \tau\ \{P\}\ \{Q\} \qquad G = \bigcirc\ G}{G, R, B \vdash \{P(v_i)\}\ \texttt{call}\ f\ (v_i)\ \{Q(v_i)\}}$$

Actually, the real rule here is more complex – this is only half of the premises…

The full rule in Coq…

```
Axiom semax_call_basic :
  forall G R B A Z P Q x F sig ret id sh a bl (v: val) vl
       (necP: forall x, boxy necM (P x))
       (necQ: forall x, boxy necM (Q x))
       (Hret: List.length (opt2list ret) = List.length (opt2list (sig_res sig)))
       (HG:  G |-- TT * (fun_id id sh A P Q))
       (Gclosed: closed G),
        semax G R B
          (global_id id =# v && a =# v && bl =#* vl &&
           prepost_match_sig (P x) (Q x) sig &&
            ([]F * ^rho own_all (opt2list ret)  * apply (P x) vl))
          (Scall ret (sig_args sig) a bl Z)
          (Ex_ vl' :_ ,
             idlist2exprlist (opt2list ret) =#* vl' &&
             ([]F * ^rho own_all (opt2list ret)   * apply (Q x) vl')).
```

Memory

- We will start with only one new feature: memory
- A memory m is a function from addresses to values
  - To keep things simpler, both will be numeric values
- We need to add two new instructions
  - x := [e] (Load)
  - $[e_1] := [e_2]$ (Store)

Semantics

- Recall from previous lecture that our states σ were pairs of locals γ and code k.

- We will add a new element to the state: states σ are now triples of memory m, locals γ, and code k.
  - $\sigma = (m, \gamma, k)$

- It is simple to define the operational semantics of load and store.

Semantics

$$\frac{\gamma \vdash e \Downarrow n \qquad \gamma' := [x \to m(n)]\,\gamma}{(m, \gamma, x := [e] \bullet k) \mapsto (m, \gamma', k)}$$

$$\frac{\gamma \vdash e_1 \Downarrow n_1 \qquad \gamma \vdash e_2 \Downarrow n_2 \qquad m' = [n_1 \to n_2]\,m}{(m, \gamma, [e_1] := e_2 \bullet k) \mapsto (m', \gamma, k)}$$

New assertion

- We also define a new assertion, written $e_1 \mapsto e_2$, which means that the memory location e_1 contains e_2.

$$(m, \gamma) \vDash e_1 \mapsto e_2$$

$$\equiv$$

$$\exists\, n_1, n_2.\ (\gamma \vdash e_1 \Downarrow n_1) \wedge (\gamma \vdash e_2 \Downarrow n_2) \wedge (m(n_1) = n_2)$$

- Using this new assertion, we can write some natural-looking Hoare rules for load and store.

Two reasonable (but not perfect) rules

- Rule for Load:

$$\{e_1 \mapsto e_2\} \quad v := [e_1] \quad \{v = e_2\}$$

- Rule for Store:

$$\{ \text{True} \} \quad [e_1] := e_2 \quad \{e_1 \mapsto e_2\}$$

Two reasonable (but not perfect) rules

- Rule for Load:

$$\{e_1 \mapsto e_2\} \quad v := [e_1] \quad \{v = e_2\}$$

- Rule for Store:

$$\{ \text{True} \} \quad [e_1] := e_2 \quad \{e_1 \mapsto e_2\}$$

These rules are not quite perfect since, for example, in the load rule if e_2 contains v then $\{v = e_2\}$ will not hold.

Two reasonable (but not perfect) rules

- Rule for Load:

$$\{e_1 \mapsto e_2\} \quad v := [e_1] \quad \{v = e_2\}$$

- Rule for Store:

$$\{ \text{True} \} \quad [e_1] := e_2 \quad \{e_1 \mapsto e_2\}$$

But as a first approximation, they are ok. We have more significant concerns to worry about.

Reasoning about multiple pointers

- Consider the following proposed Hoare program/proof:

$$\{\ x \mapsto 2 \ \wedge \ y \mapsto 3\ \} \quad [x] := 4 \quad \{x \mapsto 4 \ \wedge \ y \mapsto 3\ \}$$

Is it reasonable?

Reasoning about multiple pointers

- Consider the following proposed Hoare program/proof:

$$\{ x \mapsto 2 \ \wedge \ y \mapsto 3 \} \quad [x] := 4 \quad \{x \mapsto 4 \ \wedge \ y \mapsto 3 \}$$

Is it reasonable?

Ya, it looks ok.

Reasoning about multiple pointers

- What about this one:

$$\{ x \mapsto 2 \ \wedge \ y \mapsto 2 \} \quad [x] := 4 \quad \{x \mapsto 4 \ \wedge \ y \mapsto 2 \}$$

Is it reasonable?

Reasoning about multiple pointers

- What about this one:

$$\{ x \mapsto 2 \ \wedge \ y \mapsto 2 \} \quad [x] := 4 \quad \{x \mapsto 4 \ \wedge \ y \mapsto 2 \}$$

Is it reasonable?

Unfortunately, maybe not: what if x and y are aliased?

Reasoning about multiple pointers

- What about this one:

$\{ x \mapsto 2 \ \wedge \ y \mapsto 2 \}$ **[x] := 4** $\{x \mapsto 4 \ \wedge \ y \mapsto 2 \}$

Is it reasonable?

Unfortunately, no: what if x and y are aliased?

In that case, the postcondition is $\{x \mapsto 4 \ \wedge \ y \mapsto 4 \}$

Reasoning about multiple pointers

- So really:

$$\{ x \mapsto 2 \ \wedge \ y \mapsto 2 \}$$

$$[x] := 4$$

$$\{(x \mapsto 4 \ \wedge \ y \mapsto 2) \vee (x \mapsto 4 \ \wedge \ y \mapsto 4) \}$$

This is a sound rule.

Reasoning about multiple pointers

- So really:

$$\{ x \mapsto 2 \ \wedge \ y \mapsto 2 \}$$

$$[x] := 4$$

$$\{(x \mapsto 4 \ \wedge \ y \mapsto 2) \vee (x \mapsto 4 \ \wedge \ y \mapsto 4) \}$$

This is a sound rule.

But… it's pretty ugly as a pattern: the size of the postcondition just doubled!

Reasoning about multiple pointers

- What about if we had three pointers?

$$\{ x \mapsto 2 \ \wedge \ y \mapsto 2 \wedge \ z \mapsto 2 \}$$

[x] := 4

$$\{(x \mapsto 4 \ \wedge \ y \mapsto 2 \wedge \ z \mapsto 2) \vee$$
$$(x \mapsto 4 \ \wedge \ y \mapsto 4 \wedge \ z \mapsto 2) \vee$$
$$(x \mapsto 4 \ \wedge \ y \mapsto 2 \wedge \ z \mapsto 4) \vee$$
$$(x \mapsto 4 \ \wedge \ y \mapsto 4 \wedge \ z \mapsto 4) \ \}$$

Uh oh… the size of the postcondition is growing exponentially in the number of variables.

Reasoning about multiple pointers

- Maybe we can explicitly reason about aliasing in the precondition:

$$\{ x \mapsto 2 \ \wedge \ y \mapsto 2 \wedge x \neq y \}$$

$$[x] := 4$$

$$\{(x \mapsto 4 \ \wedge \ y \mapsto 2)\}$$

- At least the postcondition is not too bad now.

Reasoning about multiple pointers

- Maybe we can explicitly reason about aliasing in the precondition:

$$\{ x \mapsto 2 \ \wedge \ y \mapsto 2 \wedge x \neq y \}$$

$$[x] := 4$$

$$\{(x \mapsto 4 \ \wedge \ y \mapsto 2)\}$$

- At least the postcondition is not too bad now.

Reasoning about multiple pointers

- What about if we had three pointers?

$$\{ x \mapsto 2 \ \wedge \ y \mapsto 2 \wedge \ z \mapsto 2 \wedge x \neq y \wedge x \neq z \wedge y \neq z \}$$

$$[x] := 4$$

$$\{(x \mapsto 4 \ \wedge \ y \mapsto 2 \wedge \ z \mapsto 2)\}$$

- The postcondition is ok, but our precondition is growing larger.

Reasoning about multiple pointers

- What about if we had three pointers?

$$\{ x \mapsto 2 \ \wedge \ y \mapsto 2 \wedge \ z \mapsto 2 \wedge x \neq y \wedge x \neq z \wedge y \neq z \}$$

$$[x] := 4$$

$$\{(x \mapsto 4 \ \wedge \ y \mapsto 2 \wedge \ z \mapsto 2)\}$$

- In fact, the precondition is growing with the square of the number of variables.

- So this is better than before (exponential is much worse than polynomial), but it is still not ideal.

Reasoning about multiple pointers

- Pointer aliasing was **the** major problem with using Hoare logic to verify real programs for 30 years.

- Real programs can have hundreds of pointers. Tracking aliasing information was effectively impossible by hand, and even by machine was not easy (and led to other problems).

- Other approaches were needed…

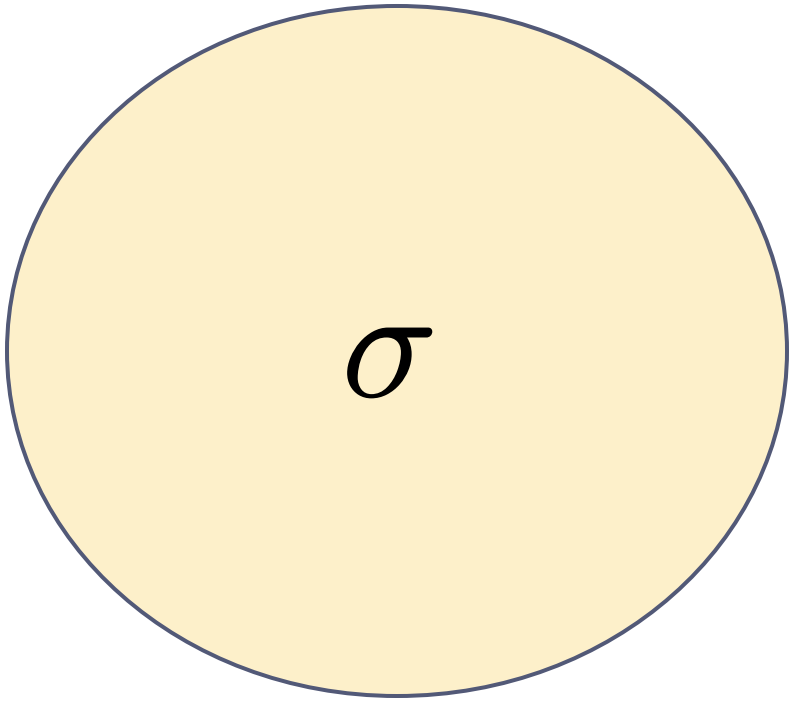
Separation Logic

- Around 10 years ago, Peter O'Hearn and John Reynolds developed an idea that allowed logical formulas to reason about resource usage.

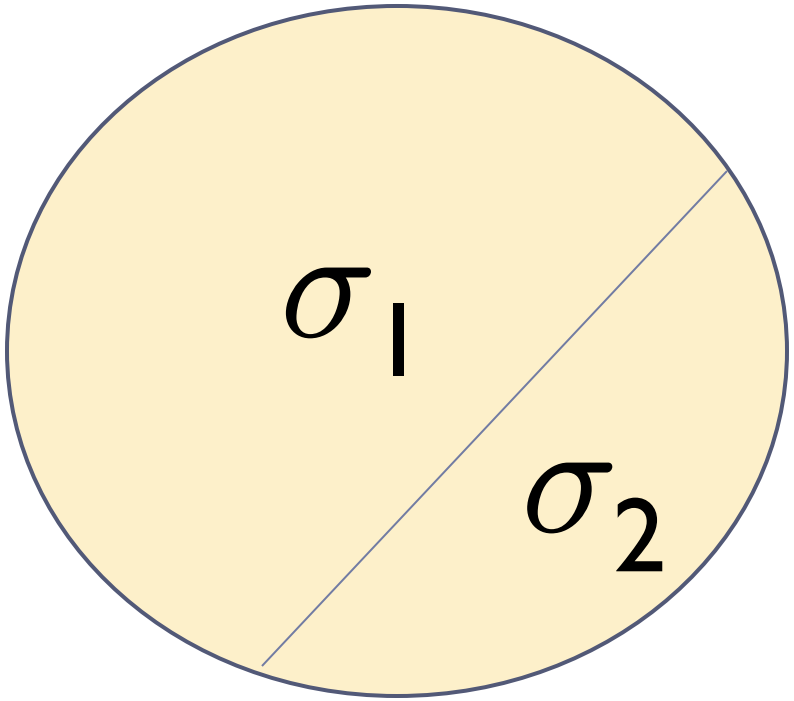
- The idea is that you add a new operator, "*", called the separating conjunction, to your formulas.

- $\sigma \models P * Q$ means, you can divide σ into two parts, σ_1 and σ_2, such that $\sigma_1 \models P$ and $\sigma_2 \models Q$.

Separation, pictorially

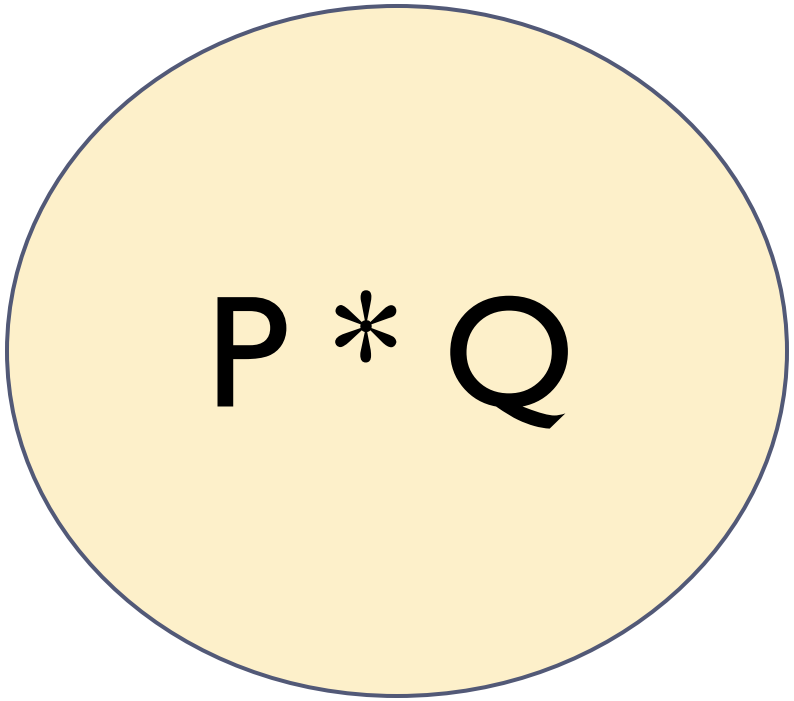


Separation, pictorially



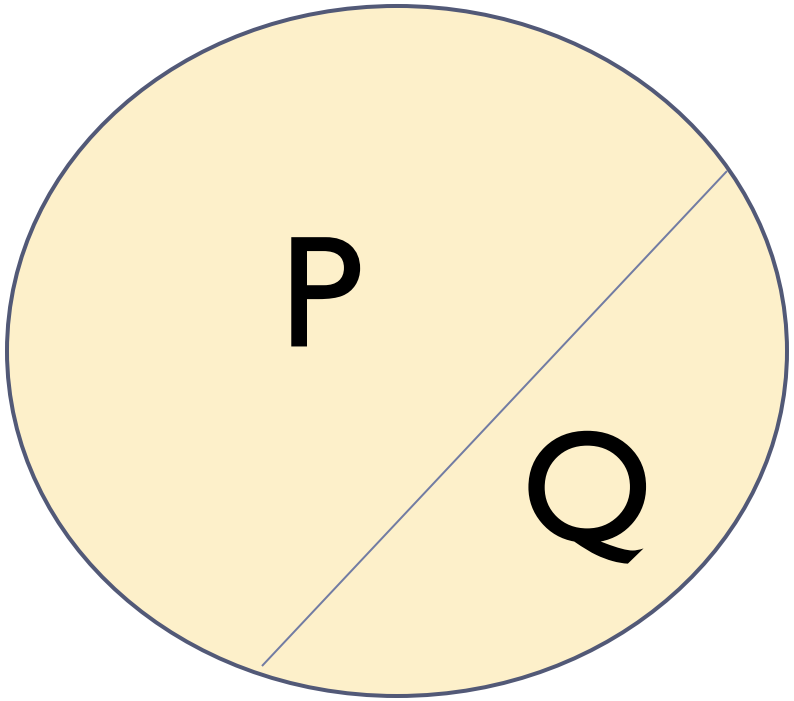
Where $\sigma = \sigma_1 \oplus \sigma_2$

Separation, pictorially



$\sigma \vDash P * Q$

Separation, pictorially



$\sigma_1 \vDash P \wedge \sigma_2 \vDash Q$

Formally

- The separating conjunction is defined as follows:

$$\sigma \vDash P * Q$$

$$\equiv$$

$$\exists\, \sigma_1, \sigma_2.\ \ \sigma_1 \oplus \sigma_2 = \sigma \quad \wedge$$
$$\sigma_1 \vDash P \quad \wedge \quad \sigma_2 \vDash Q$$

Joining

- What does this symbol "$\oplus$" mean?

$$\sigma_1 \oplus \sigma_2 = \sigma$$

- Can be a bit tricky to define, but informally it means that σ is the union of all of the resources "owned" by σ_1 and all of the resources "owned' by σ_2.

- It has nice properties, like:
  - $\sigma_1 \oplus \sigma_2 = \sigma_2 \oplus \sigma_1$
  - $\sigma_1 \oplus (\sigma_2 \oplus \sigma_3) = (\sigma_1 \oplus \sigma_2) \oplus \sigma_3$

Disjoint union

- However, there is one important point: we (almost always) require that σ_1 and σ_2 be **disjoint**. That is, resources owned by one cannot be owned by the other.
- That is, the two resources are separate.
- What resources do we care about here?

Resources

- Memory cells!
- Each memory cell will be its own resource.
- So what does { $x \mapsto 2 \quad * \quad y \mapsto 2$ } mean?

Resources

- Memory cells!
- Each memory cell will be its own resource.
- So what does { $x \mapsto 2 \quad * \quad y \mapsto 2$ } mean?
- That the memory can be split into two disjoint regions; the first region satisfies $x \mapsto 2$ and the second also satisfies $y \mapsto 2$

Resources

- Memory cells!
- Each memory cell will be its own resource.
- So what does { $x \mapsto 2 \quad * \quad y \mapsto 2$ } mean?
- That the memory can be split into two disjoint regions; the first region satisfies $x \mapsto 2$ and the second also satisfies $y \mapsto 2$
- That is, x and y are not aliased.

Store rule for separation logic

- Separating conjunction means that the store rule's postcondition is easy to state:

$$\{ x \mapsto 2 \ * \ y \mapsto 2 \}$$
$$[x] := 4$$
$$\{x \mapsto 4 \ * \ y \mapsto 2\}$$

- The point is that y is not used in this store

Store rule for separation logic

- The rule looks just as easy with three variables

$$\{ x \mapsto 2 \ * \ y \mapsto 2 * \ z \mapsto 2\}$$

$$[x] := 4$$

$$\{x \mapsto 4 \ * \ y \mapsto 2 \ * z \mapsto 2\}$$

- So this is looking much nicer… but it gets better.

Frame Rule

- Consider these two examples a bit closer:

$$\{ x \mapsto 2 \ * \ y \mapsto 2 \}$$
$$[x] := 4$$
$$\{x \mapsto 4 \ * \ y \mapsto 2\}$$

$$\{ x \mapsto 2 \ * \ y \mapsto 2 * \ z \mapsto 2\}$$
$$[x] := 4$$
$$\{x \mapsto 4 \ * \ y \mapsto 2 \ * z \mapsto 2\}$$

Frame Rule

- Consider these two examples a bit closer:

$$\{ x \mapsto 2 \ * \ (y \mapsto 2) \}$$

$$[x] := 4$$

$$\{x \mapsto 4 \ * \ (y \mapsto 2)\}$$

$$\{ x \mapsto 2 \ * \ (y \mapsto 2 \ * \ z \mapsto 2)\}$$

$$[x] := 4$$

$$\{x \mapsto 4 \ * \ (y \mapsto 2 \ * z \mapsto 2)\}$$

Frame Rule

- The red part of the formula is completely unusued – and unaffected – by the statement in question

$$\{ x \mapsto 2 \ * \ (y \mapsto 2) \}$$
$$[x] := 4$$
$$\{x \mapsto 4 \ * \ (y \mapsto 2)\}$$

$$\{ x \mapsto 2 \ * \ (y \mapsto 2 \ * \ z \mapsto 2)\}$$
$$[x] := 4$$
$$\{x \mapsto 4 \ * \ (y \mapsto 2 \ * z \mapsto 2)\}$$

Frame Rule

- Really something very general is going on:

$$\{\ x \mapsto 2 \ \ * \ \ F\ \}$$

$$[x] := 4$$

$$\{x \mapsto 4 \ \ * \ \ F\}$$

$$\{\ x \mapsto 2 \ \ * \ F\ \}$$

$$[x] := 4$$

$$\{x \mapsto 4 \ \ * \ \ F\}$$

Frame Rule

- F here is called the **frame**

$$\{ x \mapsto 2 \ * \ F \}$$

$$[x] := 4$$

$$\{x \mapsto 4 \ * \ F\}$$

- And in fact, there is a general rule called the Frame Rule:

$$\frac{\{\psi\} \ c \ \{\phi\}}{\{\psi * F\} \ c \ \{\phi * F\}}$$

Frame Rule

- Note that the Frame Rule is only true because of the *
- Good:

$$\frac{\{\psi\}\ \mathsf{c}\ \{\phi\}}{\{\psi * \mathsf{F}\}\ \mathsf{c}\ \{\phi * \mathsf{F}\}}$$

- No good:

$$\frac{\{\psi\}\ \mathsf{c}\ \{\phi\}}{\{\psi \wedge \mathsf{F}\}\ \mathsf{c}\ \{\phi \wedge \mathsf{F}\}}$$

- Why?

Frame Rule

- Because F might contain pointers and we need to know that there is no aliasing.

- Good:

$$\frac{\{x \mapsto 3\}\ \ [x] := 4\ \ \{x \mapsto 4\}}{\{x \mapsto 3 * y \mapsto 3\}\ [x] := 4\ \ \{x \mapsto 4 * y \mapsto 3\}}$$

- No good since x and y may be aliased:

$$\frac{\{x \mapsto 3\}\ \ [x] := 4\ \ \{x \mapsto 4\}}{\{x \mapsto 3 \wedge y \mapsto 3\}\ [x] := 4\ \ \{x \mapsto 4 \wedge y \mapsto 3\}}$$

I have cheated a bit so far…

- The definition of $e_1 \mapsto e_2$ given before:

$$(m, \gamma) \vDash e_1 \mapsto e_2$$
$$\equiv$$
$$\exists\, n_1, n_2.\ (\gamma \vdash e_1 \Downarrow n_1) \wedge (\gamma \vdash e_2 \Downarrow n_2) \wedge (m(n_1) = n_2)$$

- This turns out to be not quite what we want in separation logic. With this definition, $\sigma \vDash x \mapsto 3$ could be true of many memories, as long as at location x the memory contained a 3.

Two memories (only 5 locations total)

1	4
2	8
3	15
4	16
5	23

$m \vDash 3 \mapsto 15$

1	8
2	16
3	15
4	23
5	42

$m \vDash 3 \mapsto 15$

What if memories were partial functions?

1	8
2	16

3	15

$m \vDash 3 \mapsto 15$

$m \vDash 1 \mapsto 8 * 2 \mapsto 16$

Emp

- We'd like to add a special predicate, called **emp**, that is only true of the empty heap

$m \vDash 3 \mapsto 15$

$m \vDash \text{emp}$

The revised definitions

- The definition of $e_1 \mapsto e_2$ given before:

$$(m, \gamma) \vDash e_1 \mapsto e_2$$
$$\equiv$$
$$\exists\, n_1, n_2.\ (\gamma \vdash e_1 \Downarrow n_1) \wedge (\gamma \vdash e_2 \Downarrow n_2) \wedge (m(n_1) = n_2)$$

- The new definition:

$$(m, \gamma) \vDash e_1 \mapsto e_2$$
$$\equiv$$
$$\exists\, n_1, n_2.\ (\gamma \vdash e_1 \Downarrow n_1) \wedge (\gamma \vdash e_2 \Downarrow n_2) \wedge (m(n_1) = n_2) \wedge \mathrm{dom}(m) = \{n_1\}$$

Emp

$$(m, \gamma) \vDash e_1 \mapsto e_2 \quad \equiv$$

$$\exists\, n_1, n_2.\ (\gamma \vdash e_1 \Downarrow n_1) \wedge (\gamma \vdash e_2 \Downarrow n_2) \wedge (m(n_1) = n_2) \wedge \mathrm{dom}(m) = \{n_1\}$$

- That is, this is only true of the singleton heap that only contains the resource at location n_1.

- We define emp as:

$$(m, \gamma) \vDash \mathrm{emp} \quad \equiv$$

$$\mathrm{dom}(m) = \{\ \}$$

What if memories were partial functions?

1	8
2	16

3	15

$m_1 \vDash 3 \mapsto 15$

$m_2 \vDash 1 \mapsto 8 * 2 \mapsto 16$

What if memories were partial functions?

1	8
2	16
3	15

$m \vDash 3 \mapsto 15 * 1 \mapsto 8 * 2 \mapsto 16$

Where $m_1 \oplus m_2 = m$

Advantages of this approach

- We can now explicitly reason about memory allocation/freeing

$$\{ \text{emp} \} \quad v = \text{new}(3) \quad \{v \mapsto 3\}$$

$$\{e_1 \mapsto e_2\} \quad \text{free } e_1 \quad \{ \text{emp} \}$$

Let us suppose we have some program P, and we know

$$\{\text{emp}\}\ P\ \{\text{emp}\}$$

What can we conclude?

Advantages of this approach

- We can now explicitly reason about memory allocation/freeing

$$\{ \text{emp} \} \quad v = \text{new}(3) \quad \{v \mapsto 3\}$$

$$\{e_1 \mapsto e_2\} \quad \text{free } e_1 \quad \{ \text{emp} \}$$

That P has freed all the memory it allocated before exiting.

Advantages of separation logic

- Many programs have these kinds of bugs:
  - Use of memory before allocation
  - Inadvertent use of aliased memory
  - Double free of memory (usually segfaults)
  - Allocate memory but never free it (memory leak)
- Separation logic allows one to verify that a program does not have those kinds of bugs

Advantages of separation logic

Also, the frame rule

$$\frac{\{\psi\}\ \ c\ \ \{\phi\}}{\{\psi * F\}\ \ c\ \ \{\phi * F\}}$$

is hugely powerful, since it enables local reasoning.

- That is, when you are verifying some statement, you can ignore all of the parts of the state that are unrelated.

- The result is that tools based on separation logic can mechanically verify programs that are 50k lines long (size of many embedded systems and device drivers).

Using Separation Logic for other resources

- A natural observation is that there are many other kinds of resources that programs use
  - Network ports
  - Disk space
  - Portions of the graphical interface
  - OS resources
  - CPU time
  - …
- One active area of research is extending separation logic to reason about these kinds of resources.

Resource management

- It turns out, many of these resources are used in a similar way to memory cells.

- For example, network connections:

Network Connections	Memory Cells
require initialization	require allocation
should be released before exit	should be freed before exit
should not be released twice	should not be freed twice
connection aliasing dangerous	memory aliasing dangerous

Resource management

- Given these kinds of similarities, one could imagine how a network-connection-aware separation logic might work:
- {emp} x = newHTTP(addr) { Conn (x, 80, addr)}
- {Conn (x, 80, addr)} releaseHTTP(x) {emp}
- {Conn (x, 80, addr)} send(x, y) {Conn (x, 80, addr)}
- Etc.

Concurrency

- As you probably know, a concurrent program is a program that is executing multiple pieces of code at the same time.
- Almost all non-safety-critical machines today are running many programs at the same time; in turn many of those programs have multiple threads of execution.
- It's easy to have a simple Windows machine with 100+ threads running at the same time.

Concurrency and Formal Reasoning

- Why "non-safety-critical"?
- Because concurrent programming is really hard – and so the code is always filled with bugs.
- This is (kind of) acceptable if it means that you lose a few pages of your paper…
- … but totally unacceptable if the airplane decides to reboot in the middle of the trip

Scary fact…

- Most Airbus planes in the sky today use a PowerPC processor to run their flight guidance systems.
- Unfortunately, the processor they use is known to have bugs in the way it handles concurrency.
- Good thing all of the code is single-threaded…

Why not verify concurrent programs?

- Historically, it was much too hard. Consider:

$$
\begin{array}{l} S_1; \\ S_2; \\ S_3; \\ S_4; \end{array} \quad \Big\| \quad \begin{array}{l} C_1; \\ C_2; \\ C_3; \\ C_4; \end{array}
$$

Here we write $C_1 \,||\, C_2$ to mean that we execute C_1 and C_2 in parallel.

Why not verify concurrent programs?

- Historically, it was much too hard. Consider:

$$
\begin{array}{l} S_1; \\ S_2; \\ S_3; \\ S_4; \end{array} \quad \| \quad \begin{array}{l} C_1; \\ C_2; \\ C_3; \\ C_4; \end{array}
$$

The problem is that these instructions can be executed (say, from the perspective of the memory controlled) in any order: $S_1 \mapsto S_2 \mapsto C_1 \mapsto \ldots$

Why not verify concurrent programs?

- Historically, it was much too hard. Consider:

$$
\begin{array}{l} S_1; \\ S_2; \\ S_3; \\ S_4; \end{array} \quad \| \quad \begin{array}{l} C_1; \\ C_2; \\ C_3; \\ C_4; \end{array}
$$

With only 4 instructions in each thread, there are a huge number of choices. The pre/postconditions almost instantly become too large to handle (large disjunctions).

Why not verify concurrent programs?

- Historically, it was much too hard. Consider:

$$\begin{array}{l} S_1; \\ S_2; \\ S_3; \\ S_4; \end{array} \quad \| \quad \begin{array}{l} C_1; \\ C_2; \\ C_3; \\ C_4; \end{array}$$

Actually, the real picture is even worse: real processors execute instructions out of order – in a way that can be observed from other threads. (Let's have mercy and ignore this ugly truth for the rest of this lecture.)

Why not verify concurrent programs?

- Historically, it was much too hard. Consider:

$$
\begin{array}{l} S_1; \\ S_2; \\ S_3; \\ S_4; \end{array} \quad \| \quad \begin{array}{l} C_1; \\ C_2; \\ C_3; \\ C_4; \end{array}
$$

Of course, all of this complexity is directly related to the reason that concurrent programs are so hard to write!

Applying Hoare Logic

- We would like to have a rule that looked like this:

$$\frac{\{P_1\}\ C_1\ \{Q_1\} \qquad \{P_2\}\ C_2\ \{Q_2\}}{\{P_1 \wedge P_2\} \quad C_1 \parallel C_2 \quad \{Q_1 \wedge Q_2\}}$$

- Unfortunately, this rule is very hard to use since it usually unsound. Where does the difficulty come in?

Applying Hoare Logic

- We would like to have a rule that looked like this:

$$\frac{\{P_1\}\ C_1\ \{Q_1\} \qquad \{P_2\}\ C_2\ \{Q_2\}}{\{P_1 \wedge P_2\} \quad C_1 \parallel C_2 \quad \{Q_1 \wedge Q_2\}}$$

- The problem is that C_1 and C_2 may interfere with each other: then the postconditions will not hold.

Applying Separation Logic

- What happens if we replace $\wedge$ with * ?

$$\frac{\{P_1\}\ C_1\ \{Q_1\} \qquad \{P_2\}\ C_2\ \{Q_2\}}{\{P_1 * P_2\} \quad C_1 \parallel C_2 \quad \{Q_1 * Q_2\}}$$

- Is this rule sound?

Applying Separation Logic

- What happens if we replace $\wedge$ with * ?

$$\frac{\{P_1\}\ C_1\ \{Q_1\} \qquad \{P_2\}\ C_2\ \{Q_2\}}{\{P_1 * P_2\} \quad C_1 \,||\, C_2 \quad \{Q_1 * Q_2\}}$$

- Yes! Since the separating conjunction requires that the state satisfying P_1 is disjoint from the state satisfying P_2, we can run C_1 and C_2 in parallel and they will not hurt each other.

Applying Separation Logic

- What happens if we replace $\wedge$ with * ?

$$\frac{\{P_1\}\ C_1\ \{Q_1\} \qquad \{P_2\}\ C_2\ \{Q_2\}}{\{P_1 * P_2\} \quad C_1 \,||\, C_2 \quad \{Q_1 * Q_2\}}$$

- Still, those of you familiar with concurrency may detect a problem: it is not that the rule is unsound, but maybe it won't be very useful for proving things about common programs. What might the problem be?

Applying Separation Logic

- What happens if we replace $\wedge$ with * ?

$$\frac{\{P_1\}\ C_1\ \{Q_1\} \qquad \{P_2\}\ C_2\ \{Q_2\}}{\{P_1 * P_2\} \quad C_1 \,||\, C_2 \quad \{Q_1 * Q_2\}}$$

- The issue is that usually when we run things in parallel, we want the threads to be able to cooperate towards a common goal. This means that threads must communicate somehow – but the rule above seems to imply that each thread is running in isolation.

Concurrency 101: Locks

- How do threads usually (safely) communicate? The most basic technique is called a lock.

- A lock is just a memory location. There is a protocol that is used: for example, if the location contains "0" then the lock is "locked" and if it contains "1" then the lock is "unlocked"

- There are two basic operations on a lock: an operation called `lock`, and another called `unlock`.

Informal semantics

- The command `lock`(v) does the following: first it reads the location v; if it is 0 ("locked") then it waits for awhile and then tries again. Once it is 1 "unlocked" then it sets it to 0 (and then the `lock` command is done).

- The key point is that between reading the "1" and writing the "0", no other thread can execute. That is, the read-write pair is atomic.

- Assuming that this is the only way to change a "1" to a "0", this means that at most one thread holds the lock at a time.

Informal semantics

- The command `unlock`(v) is operationally much simpler: just set the memory cell v to 1 "unlocked".

- Of course, it is extremely dangerous to just unlock locks that the thread has not previously locked…

- … since in that case, more than one thread would think that it has exclusive ownership of the lock.

Why are locks used?

- The basic reason is that a lock protects some resource (usually piece of memory) that the threads use. When a thread wants to use the shared resource, it (starts to) grab the lock.

- Once it has it, then the thread uses the shared resource, confident that no other thread will use the resource in an invalid way.

- When it is done, the thread unlocks the lock (and afterwards does not use the resource).

How can we model this behavior?

- Idea: associate each lock with a formula R – this formula is called the resource invariant.
- The resource invariant will describe which resources are used.
- It is usually also very useful for the invariant to state the protocol that must be followed when using the resource.
- Example: $R = \exists\, n.\quad 3 \mapsto n + n$

What does it mean?

- Example: $R = \exists\ n.\quad 3 \mapsto n + n$

1. The lock protects memory cell 3.
2. Once a thread locks the lock it can assume that the contents of memory cell 3 is even.
3. The thread will have to ensure that memory cell 3 contains an even number before unlocking.

A new assertion and Hoare rules

- We write $\ell \rightsquigarrow R$ to mean that ℓ is a lock with resource invariant R.

- We can now define some very nice Hoare rules for lock and unlock:

$$\{\ell \rightsquigarrow R\} \quad \texttt{lock}\ \ell \quad \{(\ell \rightsquigarrow R) * R\}$$

$$\{(\ell \rightsquigarrow R) * R\} \quad \texttt{unlock}\ \ell \quad \{\ell \rightsquigarrow R\}$$

A new assertion and Hoare rules

- We write $\ell \rightsquigarrow R$ to mean that ℓ is a lock with resource invariant R.

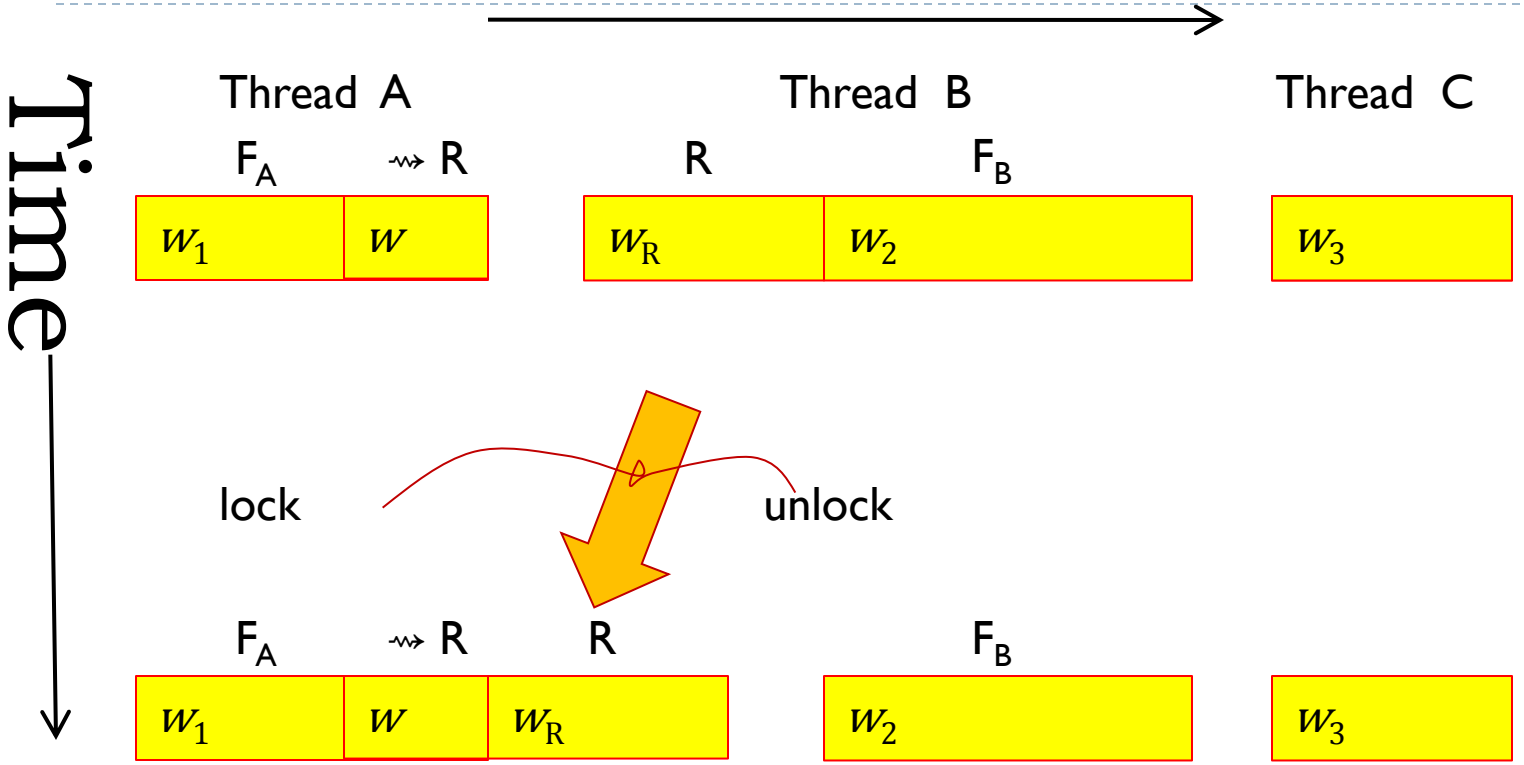
- We can now define some very nice Hoare rules for lock and unlock:

$$\{\ell \rightsquigarrow R\} \quad \texttt{lock}\ \ell \quad \{(\ell \rightsquigarrow R) * R\}$$

$$\{(\ell \rightsquigarrow R) * R\} \quad \texttt{unlock}\ \ell \quad \{\ell \rightsquigarrow R\}$$

- Perhaps a picture would help…

Space



$$\frac{}{\{\ \rightsquigarrow R\}\ \texttt{lock}\quad\{(\ \rightsquigarrow R) * R\}}\ \text{(lock rule)}$$

$$\frac{}{\{F_A * (\ \rightsquigarrow R)\}\ \texttt{lock}\quad\{F_A * (\ \rightsquigarrow R) * R\}}\ \text{(frame rule)}$$

Verification of Example Program

```
[l] := 0;
makelock l (∃y. x ↦ y+y);
[x] := 0;
unlock l;
fork child(l);
…
lock l;
   [x] := [x] + 1;
   [x] := [x] + 1;
unlock l;
```

Verification of Example Program

$\{F * l \twoheadrightarrow (\exists y.\, x \mapsto y+y)\}$

```
lock l;
```

$\{F * l \twoheadrightarrow (\exists y.\, x \mapsto y+y) * (\exists y.\, x \mapsto y+y)\}$

```
   [x] := [x] + 1;
   [x] := [x] + 1;
unlock l;
```

Verification of Example Program

```
lock l;

{F * l ⇝ (∃y. x ↦ y+y) * (∃y. x ↦ y+y)}
   [x] := [x] + 1;
{F * l ⇝ (∃y. x ↦ y+y) * (∃y. x ↦ y+y + 1)}

   [x] := [x] + 1;
unlock l;
```

Verification of Example Program

```
lock l;
   [x] := [x] + 1;
```

$\{F * l \twoheadrightarrow (\exists y.\, x \mapsto y{+}y) * (\exists y.\, x \mapsto y{+}y + 1)\}$

```
   [x] := [x] + 1;
```

$\{F * l \twoheadrightarrow (\exists y.\, x \mapsto y{+}y) * (\exists y.\, x \mapsto y{+}y + 2)\}$

```
unlock l;
```

Verification of Example Program

```
lock l;
   [x] := [x] + 1;
```

$\{F * l \twoheadrightarrow (\exists y.\, x \mapsto y{+}y) * (\exists y.\, x \mapsto y{+}y + 1)\}$

```
   [x] := [x] + 1;
```

$\{F * l \twoheadrightarrow (\exists y.\, x \mapsto y{+}y) * (\exists y.\, x \mapsto y{+}y + 2)\}$

```
unlock l;
```

$(\exists y.\, x \mapsto y{+}y)$

Verification of Example Program

```
lock l;
   [x] := [x] + 1;
```

$\{F * l \rightsquigarrow (\exists y.\, x \mapsto y{+}y) * (\exists y.\, x \mapsto y{+}y + 1)\}$

```
   [x] := [x] + 1;
```

$\{F * l \rightsquigarrow (\exists y.\, x \mapsto y{+}y) * (\exists y.\, x \mapsto y{+}y)\}$

```
unlock l;
```

Verification of Example Program

```
lock l;
   [x] := [x] + 1;
   [x] := [x] + 1;
```

$\{F * l \twoheadrightarrow (\exists y.\, x \mapsto y+y) * (\exists y.\, x \mapsto y+y)\}$

```
unlock l;
```

$\{F * l \twoheadrightarrow (\exists y.\, x \mapsto y+y)\}$

Verification of Example Program

$\{F * l \rightsquigarrow (\exists y.\, x \mapsto y{+}y)\}$

```
lock l;
```

$\{F * l \rightsquigarrow (\exists y.\, x \mapsto y{+}y) * (\exists y.\, x \mapsto y{+}y)\}$

```
   [x] := [x] + 1;
```

$\{F * l \rightsquigarrow (\exists y.\, x \mapsto y{+}y) * (\exists y.\, x \mapsto y{+}y + 1)\}$

```
   [x] := [x] + 1;
```

$\{F * l \rightsquigarrow (\exists y.\, x \mapsto y{+}y) * (\exists y.\, x \mapsto y{+}y + 2)\}$

$\{F * l \rightsquigarrow (\exists y.\, x \mapsto y{+}y) * (\exists y.\, x \mapsto y{+}y)\}$

```
unlock l;
```

$\{F * l \rightsquigarrow (\exists y.\, x \mapsto y{+}y)\}$

Verification of Example Program

$\{x \mapsto \_ * l \mapsto \_\}$
`[l] := 0;`
$\{x \mapsto \_ * l \mapsto 0\}$
`makelock l` $(\exists y.\, x \mapsto y+y)$`;`
$\{x \mapsto \_ * l \rightarrowtail (\exists y.\, x \mapsto y+y)\}$
`[x] := 0;`
$\{x \mapsto 0 * l \rightarrowtail (\exists y.\, x \mapsto y+y)\}$
$\{x \mapsto (\exists y.\, x \mapsto y+y) * l \rightarrowtail (\exists y.\, x \mapsto y+y)\}$
`unlock l;`
$\{l \rightarrowtail (\exists y.\, x \mapsto y+y)\}$
`fork child(l);`
$\{l \rightarrowtail (\exists y.\, x \mapsto y+y)\}$

`lock l;`
$\{l \rightarrowtail (\exists y.\, x \mapsto y+y) * (\exists y.\, x \mapsto y+y)\}$
`  [x] := [x] + 1;`
$\{l \rightarrowtail (\exists y.\, x \mapsto y+y) * (\exists y.\, x \mapsto y+y+1)\}$
`  [x] := [x] + 1;`
$\{l \rightarrowtail (\exists y.\, x \mapsto y+y) * (\exists y.\, x \mapsto y+y+2)\}$
$\{l \rightarrowtail (\exists y.\, x \mapsto y+y) * (\exists y.\, x \mapsto y+y)\}$
`unlock l;`
$\{l \rightarrowtail (\exists y.\, x \mapsto y+y)\}$

Lessons

```
{x ↦ _ * l ↦ _}
[l] := 0;
{x ↦ _ * l ↦ 0}
makelock l (∃y. x ↦ y+y);
{x ↦ _ * l ⇝ (∃y. x ↦ y+y)}
[x] := 0;
{x ↦ 0 * l ⇝ (∃y. x ↦ y+y)}
{x ↦ (∃y. x ↦ y+y) * l ⇝ (∃y. x ↦ y+y)}
unlock l;
{l ⇝ (∃y. x ↦ y+y)}
fork child(l);
{l ⇝ (∃y. x ↦ y+y)}

lock l;
{l ⇝ (∃y. x ↦ y+y) * (∃y. x ↦ y+y)}
  [x] := [x] + 1;
{l ⇝ (∃y. x ↦ y+y) * (∃y. x ↦ y+y + 1)}
  [x] := [x] + 1;
{l ⇝ (∃y. x ↦ y+y) * (∃y. x ↦ y+y + 2)}
{l ⇝ (∃y. x ↦ y+y) * (∃y. x ↦ y+y)}
unlock l;
{l ⇝ (∃y. x ↦ y+y)}
```

A) Many details!

(Actually, some omitted!)

B) Machine-checking is key

C) Has been done for larger example programs (in Coq)

D) Machine-generation would be very helpful

Questions?