Recap

In the previous lectures, we have considered a language, WHILE, where mutability only concerned program variables.

In this lecture, we will extend the WHILE language with pointer operations on a heap, and look at the challenges Hoare logic faces when trying to reason about this language.

This will motivate introducing an extension of Hoare logic, called separation logic, to enable practical reasoning about pointers.

Syntax of \( \text{WHILE}_p \)

We introduce new commands to manipulate the heap:

\[
E ::= N \mid X \mid E_1 + E_2 \quad \text{arithmetic expressions}
\]
\[
\mid E_1 - E_2 \mid E_1 \times E_2 \mid \cdots
\]
\[
\text{null} \overset{\text{def}}{=} 0
\]

\[
B ::= T \mid F \mid E_1 = E_2 \quad \text{boolean expressions}
\]
\[
\mid E_1 \leq E_2 \mid E_1 \geq E_2 \mid \cdots
\]

\[
C ::= \text{skip} \mid C_1; C_2 \mid X := E \quad \text{commands}
\]
\[
\mid \text{if } B \text{ then } C_1 \text{ else } C_2
\]
\[
\mid \text{while } B \text{ do } C
\]
\[
\mid X := [E] \mid [E_1] := E_2
\]
\[
\mid X := \text{alloc}(E_0, \ldots, E_n)
\]
\[
\mid \text{dispose}(E)
\]
Adapting the base constructs to handle the heap

The base constructs can be adapted to handle the extended state and failure in the expected way:

\[ \begin{align*}
\ell[E](s) &= N \\
\langle X := E, (s, h) \rangle &\rightarrow (\text{skip} \cdot (s[X \rightarrow N], h)) \\
\langle \text{skip}, C_2, (s, h) \rangle &\rightarrow \langle C_2, (s, h) \rangle \\
\langle C_1, (s, h) \rangle &\rightarrow \langle C_1; C_2, (s, h) \rangle \\
\langle C_1; C_2, (s, h) \rangle &\rightarrow \langle C_1; C_2, (s, h) \rangle \\
\langle \text{if } B \text{ then } C_1 \text{ else } C_2, (s, h) \rangle &\rightarrow \langle C_1, (s, h) \rangle \\
\langle \text{if } B \text{ then } C_1 \text{ else } C_2, (s, h) \rangle &\rightarrow \langle C_2, (s, h) \rangle \\
\langle \text{while } B \text{ do } C, (s, h) \rangle &\rightarrow \langle \text{skip} \cdot (s, h) \rangle \\
\langle \text{while } B \text{ do } C, (s, h) \rangle &\rightarrow \langle C; \text{while } B \text{ do } C, (s, h) \rangle \\
\langle C_1, (s, h) \rangle &\rightarrow \langle C_1; C_2, (s, h) \rangle \\
\langle C_1; C_2, (s, h) \rangle &\rightarrow \langle C_1; C_2, (s, h) \rangle \\
\end{align*} \]

Failure

Heap assignment, dereferencing, and deallocation fail if the given locations are not currently allocated.

This is a design choice that makes \( \text{WHILE}_p \) more like a programming language, whereas having a heap with all locations always allocated would make \( \text{WHILE}_p \) more like assembly.

To explicitly model failure, we introduce a distinguished failure value \( \bot \), and adapt the semantics:

\[ \rightarrow : \mathcal{P}((\text{Cmd} \times \text{State}) \times ((\text{Cmd} \times \text{State}) \cup \{ \bot \})) \]

Why failure?

Instead of modelling failure explicitly, we could just leave the configuration stuck, but explicit failure makes things clearer and easier to state.

In particular, \( \text{WHILE}_p \) is somewhat safe in the following sense:

\[ \forall C, s, h \cdot 
\begin{align*}
\langle C, (s, h) \rangle &\rightarrow^* \bot \lor \\
\langle C, (s, h) \rangle &\rightarrow^\omega \lor \\
\exists h', s'. \langle C, (s, h) \rangle &\rightarrow^* (\text{skip}, \langle s', h' \rangle) \\
\end{align*} \]
Heap dereferencing

Dereferencing an allocated location stores the value at that location to the target program variable:

\[
\mathcal{E}[E](s) = \ell \quad \ell \in \text{dom}(h) \quad h(\ell) = N
\]

\[
\langle X := [E], (s, h) \rangle \rightarrow \langle \text{skip}, (s[X \mapsto N], h) \rangle
\]

Dereferencing an unallocated location and dereferencing something that is not a location lead to a fault:

\[
\mathcal{E}[E](s) = \ell \quad \ell \notin \text{dom}(h)
\]

\[
\langle X := [E], (s, h) \rangle \rightarrow \text{\texttt{halt}}
\]

\[
\exists \ell. \mathcal{E}[E](s) = \ell
\]

\[
\langle X := [E], (s, h) \rangle \rightarrow \text{\texttt{halt}}
\]

We could have heap dereferencing be an expression, but then expressions would fault, which would add complexity.

Heap assignment

Assigning to an allocated location updates the heap at that location with the assigned value:

\[
\mathcal{E}[E_1](s) = \ell \quad \ell \in \text{dom}(h) \quad \mathcal{E}[E_2](s) = N
\]

\[
\langle [E_1] := E_2, (s, h) \rangle \rightarrow \langle \text{skip}, (s, h[\ell \mapsto N]) \rangle
\]

Assigning to an unallocated location or to something that is not a location leads to a fault:

\[
\mathcal{E}[E_1](s) = \ell \quad \ell \notin \text{dom}(h)
\]

\[
\langle [E_1] := E_2, (s, h) \rangle \rightarrow \text{\texttt{halt}}
\]

\[
\exists \ell. \mathcal{E}[E_1](s) = \ell
\]

\[
\langle [E_1] := E_2, (s, h) \rangle \rightarrow \text{\texttt{halt}}
\]

Deallocation

Deallocating an allocated location removes that location from the heap:

\[
\mathcal{E}[E](s) = \ell \quad \ell \in \text{dom}(h)
\]

\[
\langle \text{dispose}(E), (s, h) \rangle \rightarrow \langle \text{skip}, (s, h \setminus \{\ell, h(\ell)\}) \rangle
\]

Deallocating an unallocated location or something that is not a location leads to a fault:

\[
\mathcal{E}[E](s) = \ell \quad \ell \notin \text{dom}(h)
\]

\[
\langle \text{dispose}(E), (s, h) \rangle \rightarrow \text{\texttt{halt}}
\]

\[
\exists \ell. \mathcal{E}[E](s) = \ell
\]

\[
\langle \text{dispose}(E), (s, h) \rangle \rightarrow \text{\texttt{halt}}
\]

Allocation

Allocating finds a block of unallocated locations of the right size, updates the heap at those locations with the initialisation values, and stores the start-of-block location to the target program variable:

\[
\mathcal{E}[E_0](s) = N_0 \quad \ldots \quad \mathcal{E}[E_n](s) = N_n
\]

\[
\forall i \in \{0, \ldots, n\}, \ell + i \notin \text{dom}(h)
\]

\[
\ell \neq \text{null}
\]

\[
\langle X := \text{alloc}(E_0, \ldots, E_n), (s, h) \rangle \rightarrow \langle \text{skip}, (s[X \mapsto \ell], h[\ell \mapsto N_0, \ldots, \ell + n \mapsto N_n]) \rangle
\]

Because the heap has a finite domain, it is always possible to pick a suitable \(\ell\), so allocation never faults. A real machine would run out of memory at some point.

Because of allocation, \(\text{WHILE}_p\) is not deterministic.
Pointers

WHILE$_p$ has proper pointer operations, as opposed for example to references:

- pointers can be invalid: $X := \null$ faults
- we can perform pointer arithmetic:
  - $X := \text{alloc}(37, 42); Y := [X + 1]$ ends with $Y = 42$
  - $X := \text{alloc}(0); \text{if } X = 3 \text{ then } [3] := 1 \text{ else } [X] := 2$ is safe

We do not have a separate type of pointers: we use integers as pointers.

Pointers in C have many more subtleties. For example, in C, pointers can point to the stack.

Operations on mutable data structures

For instance, this operation deletes the first element of the list:

$X := [\text{HEAD} + 1]; // lookup address of second element$
$\text{dispose}(\text{HEAD}); // deallocate first element$
$\text{dispose}(\text{HEAD} + 1);$
$\text{HEAD} := X // swing head to point to second element$

Pointers and data structures

In WHILE$_p$, we can encode data structures in the heap. For example, we can encode the mathematical list $[12, 99, 37]$ with the following singly-linked list:

$$\begin{align*}
\text{HEAD} & \rightarrow 12 \rightarrow 99 \rightarrow 37 \rightarrow \null \\
X & \rightarrow 12 \rightarrow 99 \rightarrow 37 \rightarrow \null \\
X & \rightarrow 99 \rightarrow 37 \rightarrow \null
\end{align*}$$

More concretely:

$$\begin{align*}
\text{HEAD} = 10 & \rightarrow 0 \rightarrow 7 \rightarrow 8 \rightarrow 10 \rightarrow 11 \rightarrow 121 \rightarrow 122 \\
X & \rightarrow 99 \rightarrow 121 \rightarrow 12 \rightarrow 7 \rightarrow 37 \rightarrow 0
\end{align*}$$

In WHILE, we would have had to encode that in integers, for example as $\text{HEAD} = 2^{12} \times 3^{99} \times 5^{37}$ (as in Part IB Computation theory).

Attempting to reason about pointers in Hoare logic
Attempting to reason about pointers in Hoare logic

We will show that reasoning about pointers in Hoare logic is not practicable.

To do so, we will first show what makes compositional reasoning possible in standard Hoare logic (in the absence of pointers), and then show how it fails when we introduce pointers.

Approximating modified program variables

We can syntactically overapproximate the set of program variables that might be modified by a command $C$:

$\text{mod}(\text{skip}) = \emptyset$
$\text{mod}(X := E) = \{X\}$
$\text{mod}(C_1; C_2) = \text{mod}(C_1) \cup \text{mod}(C_2)$
$\text{mod}(\text{if } B \text{ then } C_1 \text{ else } C_2) = \text{mod}(C_1) \cup \text{mod}(C_2)$
$\text{mod}(\text{while } B \text{ do } C) = \text{mod}(C)$
$\text{mod}([E_1] := E_2) = \emptyset$
$\text{mod}(X := [E]) = \{X\}$
$\text{mod}(X := \text{alloc}(E_0, \ldots, E_n)) = \{X\}$
$\text{mod}(\text{dispose}(E)) = \emptyset$

The rule of constancy

In standard Hoare logic (without the rules that we will introduce later, and thus without the new commands we have introduced), the rule of constancy expresses that assertions that do not refer to program variables modified by a command are automatically preserved during its execution:

$\vdash \{P\} C \{Q\}$

$\text{mod}(C) \cap \text{FV}(R) = \emptyset$

$\vdash \{P \land R\} C \{Q \land R\}$

This rule is admissible in standard Hoare logic.

For reference: free variables

The set of free variables of a term and of an assertion is given by

$FV(\_ : \text{Term} \to \mathcal{P}(\text{Var})$
$FV(\chi) \overset{\text{def}}{=} \{\chi\}$
$FV(f(t_1, \ldots, t_n)) \overset{\text{def}}{=} FV(t_1) \cup \ldots \cup FV(t_n)$

and

$FV(\_ : \text{Assertion} \to \mathcal{P}(\text{Var})$
$FV(\top) = FV(\bot) \overset{\text{def}}{=} \emptyset$
$FV(P \land Q) = FV(P \lor Q) = FV(P \Rightarrow Q) \overset{\text{def}}{=} FV(P) \cup FV(Q)$
$FV(\forall x. P) = FV(\exists x. P) \overset{\text{def}}{=} FV(P) \setminus \{x\}$
$FV(t_1 = t_2) \overset{\text{def}}{=} FV(t_1) \cup FV(t_2)$
$FV(p(t_1, \ldots, t_n)) \overset{\text{def}}{=} FV(t_1) \cup \ldots \cup FV(t_n)$

respectively.
Modularity and the rule of constancy

This rule is important for modularity, as it allows us to only mention the part of the state that we access.

Using the rule of constancy, we can separately verify two complicated commands:

\[ \vdash \{ P \} C_1 \{ Q \} \quad \vdash \{ R \} C_2 \{ S \} \]

and then, as long as they use different program variables, we can compose them.

For example, if \( \text{mod}(C_1) \cap FV(R) = \emptyset \) and \( \text{mod}(C_2) \cap FV(Q) = \emptyset \), we can compose them sequentially:

\[ \vdash \{ P \} C_1 \{ Q \} \quad \vdash \{ R \} C_2 \{ S \} \]

\[ \vdash \{ \exists v. E_1 \leftrightarrow v \} [E_1] := E_2 \{ E_1 \leftrightarrow E_2 \} \]

Reasoning about pointers

In the presence of pointers, we can have aliasing: syntactically distinct expressions can refer to the same location. Updates made through one expression can thus influence the state referenced by other expressions.

This complicates reasoning, as we explicitly have to track inequality of pointers to reason about updates:

\[ \vdash \{ \exists v. E_1 \leftrightarrow v \land E_1 \neq E_3 \land E_3 \leftrightarrow E_4 \} \]

We have to assume that any location is possibly modified unless stated otherwise in the precondition. This is not compositional at all, and quickly becomes unmanageable.

A bad rule for reasoning about pointers

Imagine we extended Hoare logic with a new assertion, \( t_1 \leftrightarrow t_2 \), for asserting that location \( t_1 \) currently contains the value \( t_2 \), and extended the proof system with the following (sound) rule:

\[ \vdash \{ \exists v. E_1 \leftrightarrow v \} [E_1] := E_2 \{ E_1 \leftrightarrow E_2 \} \]

Then we would lose the rule of constancy, as using it, we would be able to derive

\[ \vdash \{ \exists v. 37 \leftrightarrow v \} [37] := 42 \{ 37 \leftrightarrow 42 \} \quad \text{mod}([37] := 42) \cap FV(Y \rightarrow 0) = \emptyset \]

\[ \vdash \{ \exists v. 37 \leftrightarrow v \land Y \rightarrow 0 \} [37] := 42 \{ 37 \leftrightarrow 42 \land Y \rightarrow 0 \} \]

even if \( Y = 37 \), in which case the postcondition would require 0 to be equal to 42. There is a problem!

Separation logic
Separation logic

Separation logic is an extension of Hoare logic that enables **modular** reasoning about **resources**. It introduces new connectives to reason about the combination of **disjoint** resources.

We will use separation logic to reason about pointers in WHILE\(_p\). Our resources will be parts of the heap, and we will use the new connectives of separation logic to control aliasing.

Where a Hoare logic assertion refers to a (freely duplicable) property of the current state, a separation logic assertion asserts **ownership** of resources. Resources can be combined or compared (and exchanged), but need to be accounted for.

The points-to assertion

We introduce a new assertion, written \(t_1 \rightarrow t_2\), and read “\(t_1\) points to \(t_2\)”, to reason about individual heap cells.

The points-to assertion \(t_1 \rightarrow t_2\)

- asserts that the current value that heap location \(t_1\) maps to is \(t_2\) (like \(t_1 \rightarrow t_2\)), and
- asserts ownership of heap location \(t_1\).

For example, \(X \rightarrow Y + 1\) asserts that the current value of heap location \(X\) is \(Y + 1\), and moreover asserts ownership of that heap location.

History and terminology

Separation logic was proposed by John Reynolds in 2000, and developed further by Peter O’Hearn and Hongseok Yang around 2001. It is still a very active area of research.

There are many variants of separation logic.

In WHILE\(_p\), the heap in explicitly managed: the program is meant to dispose of heap locations itself. To be able to show that our programs do not leak memory, we are going to consider a so-called linear (or classical) separation logic. If we were not interested in reasoning about deallocation, for example because there is no garbage collector, we could use an affine (or intuitionistic) separation logic.

The separating conjunction

Separation logic extends Hoare logic with a new connective, the separating conjunction ‘\(\ast\)’, to reason about disjoint resources.

The assertion \(P \ast Q\) asserts that \(P\) and \(Q\) hold (somewhat like \(P \land Q\)); however, it also asserts that the resources (the parts of the heap) owned by \(P\) and \(Q\) are **disjoint**.

The separating conjunction has a neutral element, \(emp\), which describes the empty resource (the empty heap):

\(emp \ast P \iff P \iff P \ast emp.\)
1. \((t_1 \mapsto t_2) \ast (t_3 \mapsto t_4)\)

This assertion is unsatisfiable in a state where \(t_1 = t_3\), since \(t_1 \mapsto t_2\) and \(t_3 \mapsto t_4\) would both assert ownership of the same location.

A heap satisfying this assertion is of the following shape:

\[
\begin{array}{c}
t_1 \\
t_2 \\
t_3 \\
t_4
\end{array}
\]

2. For example,

\[
((X \mapsto 101) \ast (Y \mapsto 102)) \land X = 7 \land Y = 41
\]

is satisfied by the following heap:

\[
\begin{array}{c}
\begin{array}{c}
X \\
7
\end{array} \\
\begin{array}{c}
101 \\
102
\end{array} \\
\begin{array}{c}
Y = 7 \\
Y = 41
\end{array}
\end{array}
\]

3. \((t_1 \mapsto t_2) \ast (t_1 \mapsto t_3)\)

This assertion is not satisfiable, as \(t_1\) is not disjoint from itself.

4. \(t_1 \mapsto t_2 \land t_3 \mapsto t_4\)

This asserts that the heap is described by \(t_1 \mapsto t_2\), and also by \(t_3 \mapsto t_4\).

Therefore, \(t_1 = t_2\), and so \(t_3 = t_4\)

5. A heap satisfying

\[
(t_1 \mapsto t_2) \ast (t_2 \mapsto t_1)
\]

is of the following shape:

\[
\begin{array}{c}
t_1 \\
t_2 \\
t_1
\end{array}
\]

6. For instance, a heap satisfying

\[
(X \mapsto Y) \ast (Y \mapsto X)
\]

is of the following shape:

\[
\begin{array}{c}
\begin{array}{c}
X \\
Y
\end{array}
\end{array}
\]
Examples of separation logic assertions

7. \((X \mapsto t_0, Y) \ast (Y \mapsto t_1, \text{null})\)

Here, \(X \mapsto t_0, \ldots, t_n\) is shorthand for
\((X \mapsto t_0) \ast ((X + 1) \mapsto t_1) \ast \cdots \ast ((X + n) \mapsto t_n)\)

8. \(\exists x, y. (\text{HEAD} \mapsto 12, x) \ast (x \mapsto 99, y) \ast (y \mapsto 37, \text{null})\)

This describes our singly linked list from earlier:

Semantics of separation logic assertions

The semantics of a separation logic assertion \(P\), \([[P]]\), is the set of states (that is, pairs of a stack and a heap) that satisfy \(P\).

It is simpler to define it indirectly, through the semantics of \(P\) given a stack \(s\), written \([[P]](s)\), which is the set of heaps that, together with stack \(s\), satisfy \(P\).

Recall that we want to capture the notion of ownership:
if \(h \in [[P]](s)\), then \(P\) should assert ownership of any locations in \(\text{dom}(h)\).

The heaps \(h \in [[P]](s)\) are thus referred to as partial heaps, since they only contain the locations owned by \(P\).
Semantics of separation logic assertions: points-to

The points-to assertion $t_1 \mapsto t_2$ asserts ownership of the location referenced by $t_1$, and that this location currently contains $t_2$:

$$[t_1 \mapsto t_2](s) \overset{\text{def}}{=} \begin{cases} \{h \in \text{Heap} | \exists \ell, N. [t_1](s) = \ell \land \ell \neq \text{null} \land [t_2](s) = N \land \text{dom}(h) = \{\ell\} \land h(\ell) = N\} \end{cases}$$

$t_1 \mapsto t_2$ only asserts ownership of location $\ell$, so to capture ownership, requires $\text{dom}(h) = \{\ell\}$.

Semantics of separation logic assertions: $\ast$

Separating conjunction, $P \ast Q$, asserts that the heap can be split into two disjoint parts such that one satisfies $P$, and the other $Q$:

$$[P \ast Q](s) \overset{\text{def}}{=} \begin{cases} h \in \text{Heap} | \exists h_1, h_2. h_1 \in [P](s) \land h_2 \in [Q](s) \land h = h_1 \cup h_2 \end{cases}$$

where $h = h_1 \cup h_2$ is equal to $h = h_1 \cup h_2$, but only holds when $\text{dom}(h_1) \cap \text{dom}(h_2) = \emptyset$.

Summary: separation logic assertions

Separation logic assertions not only describe properties of the current state (as Hoare logic assertions did), but also assert ownership of parts of the current heap.

Separation logic controls aliasing of pointers by enforcing that assertions own disjoint parts of the heap.
Semantics of separation logic triples

Separation logic not only extends the assertion language, but strengthens the semantics of correctness triples in two ways:

- they ensure that commands do not fail;
- they ensure that the ownership discipline associated with assertions is respected.

Ownership and separation logic triples

Separation logic triples ensure that the ownership discipline is respected by requiring that the precondition asserts ownership of any heap cells that the command might use.

For instance, we want the following triple, which asserts ownership of location 37, stores the value 42 at this location, and asserts that after that location 37 contains value 42, to be valid:

\[ \models \{37 \mapsto 1\} \{37 := 42\} \{37 \mapsto 42\} \]

However, we do not want the following triple to be valid, because it updates a location that it is not the owner of:

\[ \not\models \{100 \mapsto 1\} \{37 := 42\} \{100 \mapsto 1\} \]

even though the precondition ensures that the postcondition is true!

Framing

How can we make this principle that triples must assert ownership of the heap cells they modify precise?

The idea is to require that all triples must preserve any assertion that asserts ownership of a part of the heap disjoint from the part of the heap that their precondition asserts ownership of.

This is exactly what the separating conjunction, \(\ast\), allows us to express.
The frame rule

This intent that all triples preserve any assertion \( R \) disjoint from the precondition, called the frame, is captured by the frame rule:

\[
\begin{align*}
\vdash \{P\} & \quad C \quad \{Q\} \\
\mod(C) \cap \text{FV}(R) = \emptyset \\
\vdash \{P \ast R\} & \quad C \quad \{Q \ast R\}
\end{align*}
\]

The frame rule is similar to the rule of constancy, but uses the separating conjunction to express separation.

We still need to be careful about program variables (in the stack), so we need \( \mod(C) \cap \text{FV}(R) = \emptyset \).

Examples of framing

This problem does not arise for triples that assert ownership of the heap cells they modify, since triples only have to preserve frames disjoint from the precondition.

For instance, consider this triple which asserts ownership of location 37:

\[
\{37 \mapsto 1\} [37] := 42 \{37 \mapsto 42\}
\]

If we frame on \( 37 \mapsto 1 \), then we get the following triple, which holds vacuously since no initial state satisfies \( 37 \mapsto 1 \ast 37 \mapsto 1 \):

\[
\{37 \mapsto 1 \ast 37 \mapsto 1\} [37] := 42 \{37 \mapsto 42 \ast 37 \mapsto 1\}
\]

Informal semantics of separation logic triples

The meaning of \( \{P\} C \{Q\} \) in separation logic is thus

- \( C \) does not fault when executed in an initial state satisfying \( P \), and
- if \( h_1 \) satisfies \( P \), and if when executed from an initial state with an initial heap \( h_1 \cup h_F \), \( C \) terminates, then the terminal heap has the form \( h'_1 \cup h_F \), where \( h'_1 \) satisfies \( Q \).

The first condition ensures that the precondition asserts ownership of all the locations that might be accessed.

The second condition bakes in the requirement that triples must satisfy framing, by requiring that they preserve all disjoint heaps \( h_F \).
Formal semantics of separation logic triples

Written formally, the semantics is:

\[ \models \{ P \} C \{ Q \} \iff \]

\((\forall s, h. h \in \llbracket P \rrbracket(s) \Rightarrow \neg((C, \langle s, h \rangle) \rightarrow^* \emptyset)) \land \)

\((\forall s, h_F, s', h'. \text{dom}(h_1) \cap \text{dom}(h_F) = \emptyset \land h_1 \in \llbracket P \rrbracket(s) \land (C, \langle s, h_1 \uplus h_F \rangle) \rightarrow^* \langle \text{skip}, \langle s', h' \rangle \rangle \rightarrow^* \emptyset) \Rightarrow \exists h'_1. h' = h'_1 \uplus h_F \land h'_1 \in \llbracket Q \rrbracket(s')) \)

We then have the semantic version of the frame rule baked in:

If \( \models \{ P \} C \{ Q \} \) and \( \text{mod}(C) \cap \text{FV}(R) = \emptyset \), then

\( \models \{ P \ast R \} C \{ Q \ast R \} \).

Summary

Separation logic is an extension of Hoare logic that enables modular reasoning about resources. It extends Hoare logic with new assertions, and refines the semantics of assertions to reason about ownership and separation.

We leverage this to control aliasing, which enables practical reasoning about pointers and mutable data structures.

In the next lecture, we will look at a proof system for separation logic, and apply separation logic to examples.

Papers of historical interest: