Hoare Logic and Model Checking

Jean Pichon-Pharabod
University of Cambridge

CST Part II – 2017/18

Acknowledgements

These slides are heavily based on previous versions by Mike Gordon, Alan Mycroft, and Kasper Svendsen.

Thanks to Mistral Contrastin, Victor Gomes, and Ian Orton for reporting mistakes.

Motivation

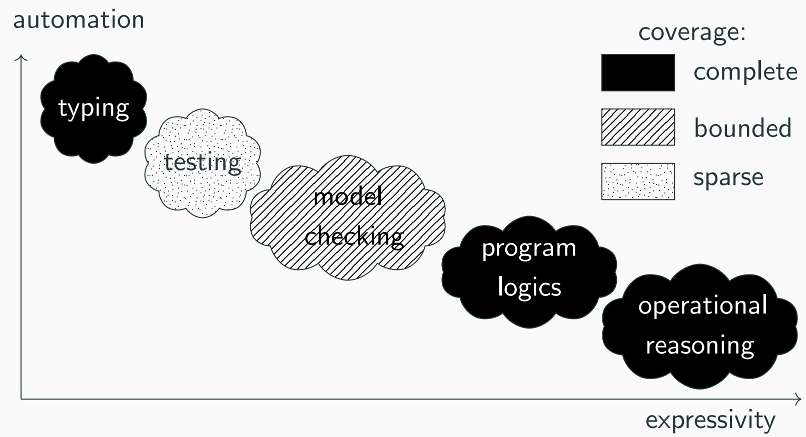
We often fail to write programs that meet our expectations, which we phrased in their specifications:

- we fail to write programs that meet their specification;
- we fail to write specifications that meet our expectations.

Addressing the former issue is called verification, and addressing the latter is called validation.

Background

There are many verification & validation techniques of varying coverage, expressivity, level of automation, ..., for example:



Choice of technique

More expressive and complete techniques lead to more confidence.

It is important to choose the right set of verification & validation techniques for the task at hand:

- verified designs may still not work;
- verification can give a false sense of security;
- verification can be very expensive and time-consuming.

More heavyweight techniques should be used together with testing, not as a replacement.

Course structure

This course is about two techniques, their underlying ideas, how to use them, and why they are correct:

- **Hoare logic** (Lectures 1-6);
- **Model checking** (Lectures 7-12).

These are not just techniques, but also ways of thinking about programs.

Lecture plan

Lecture 1: Informal introduction to Hoare logic

Lecture 2: Formal semantics of Hoare logic

Lecture 3: Examples, loop invariants, and total correctness

Lecture 4: Mechanised program verification

Lecture 5: Separation logic

Lecture 6: Examples in separation logic

Hoare logic

Hoare logic

Hoare logic is a formalism for relating the **initial** and **terminal** state of a program.

Hoare logic was invented in 1969 by Tony Hoare, inspired by earlier work of Robert Floyd.
There was little-known prior work by Alan Turing.

Hoare logic is still an active area of research.

Hoare logic

Hoare logic uses **partial correctness triples** (also "Hoare triples") for specifying and reasoning about the behaviour of programs:

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

Here, C is a command, and P and Q are state predicates:

- P is called the precondition, and describes the initial state;
- Q is called the postcondition, and describes the terminal state.

Hoare logic

To define a Hoare logic, we need four main components:

- the programming language that we want to reason about, along with its dynamic (e.g. operational) semantics;
- an assertion language for defining state predicates, along with a semantics;
- an interpretation of Hoare triples;
- a (sound) proof system for deriving Hoare triples.

This lecture will introduce each component informally.
In the coming lectures, we will cover the formal details.

The WHILE language

The WHILE language

WHILE is the prototypical imperative language. Programs consist of commands, which include branching, iteration, and assignment:

$$
\begin{array}{rcl}
C & ::= & \textbf{skip} \\
 & | & C_1; C_2 \\
 & | & V := E \\
 & | & \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2 \\
 & | & \textbf{while } B \textbf{ do } C
\end{array}
$$

Here, V is a variable, E is an arithmetic expression, which evaluates to a natural number, and B is a boolean expression, which evaluates to a boolean.

States are mappings from variables to natural numbers.

The WHILE language

The grammar for arithmetic expressions and boolean expressions includes the usual arithmetic operations and comparison operators, respectively:

$$
\begin{array}{rcll}
E & ::= & N \mid V \mid E_1 + E_2 & \textit{arithmetic expressions} \\
 & | & E_1 - E_2 \mid E_1 \times E_2 \mid \cdots & \\
 & & & \\
B & ::= & T \mid F \mid E_1 = E_2 & \textit{boolean expressions} \\
 & | & E_1 \leq E_2 \mid E_1 \geq E_2 \mid \cdots &
\end{array}
$$

Note that expressions do not have side effects.

The assertion language

Hoare logic

State predicates P and Q can refer to program variables from C, and will be written using standard mathematical notations together with **logical operators** like:

- $\wedge$ ("and"), $\vee$ ("or"), $\neg$ ("not"), and $\Rightarrow$ ("implies")

For instance, the predicate $X = Y + 1 \wedge Y > 0$ describes states in which the variable Y contains a positive value, and the value of X is equal to the value of Y plus 1.

Partial correctness triples

The partial correctness triple $\{P\}\ C\ \{Q\}$ holds if and only if:

- assuming C is executed in an initial state satisfying P,
- and assuming moreover that this execution terminates,
- then the terminal state of the execution satisfies Q.

For instance,

- $\{X = 1\}\ X := X + 1\ \{X = 2\}$ holds;
- $\{X = 1\}\ X := X + 1\ \{X = 3\}$ does not hold.

Partial correctness

Partial correctness triples are called **partial** because they only specify the intended behaviour of terminating executions.

For instance, $\{X = 1\}$ **while** $X > 0$ **do** $X := X + 1\ \{X = 0\}$ holds, because the given program never terminates when executed from an initial state where X is 1.

Hoare logic also features total correctness triples that strengthen the specification to require termination.

Total correctness

The total correctness triple $[P]\ C\ [Q]$ holds if and only if:

- assuming C is executed in an initial state satisfying P,
- then the execution terminates,
- and the terminal state satisfies Q.

There is no standard notation for total correctness triples; we will use $[P]\ C\ [Q]$.

Total correctness

The following total correctness triple does not hold:

$$[X = 1]\ \textbf{while}\ X > 0\ \textbf{do}\ X := X + 1\ [X = 0]$$

- the loop never terminates when executed from an initial state where X is positive.

The following total correctness triple does hold:

$$[X = 0]\ \textbf{while}\ X > 0\ \textbf{do}\ X := X + 1\ [X = 0]$$

- the loop always terminates immediately when executed from an initial state where X is zero.

Total correctness

Informally: total correctness = termination + partial correctness.

It is often easier to show partial correctness and termination separately.

Termination is usually straightforward to show, but there are examples where it is not: no one knows whether the program below terminates for all values of X:

$$\begin{array}{l} \textbf{while } X > 1 \textbf{ do} \\ \quad \textbf{if } ODD(X) \textbf{ then } X := 3 \times X + 1 \textbf{ else } X := X \ DIV \ 2 \end{array}$$

Microsoft's T2 tool is used to prove termination of systems code.

Specifications

Simple examples

$\{\bot\}\ C\ \{Q\}$

- this says nothing about the behaviour of C, because $\bot$ never holds for any initial state.

$\{\top\}\ C\ \{Q\}$

- this says that whenever C halts, Q holds.

$\{P\}\ C\ \{\top\}$

- this holds for every precondition P and command C, because $\top$ always holds in the terminate state.

Simple examples

$[P]\ C\ [\top]$

- this says that C always terminates when executed from an initial state satisfying P.

$[\top]\ C\ [Q]$

- this says that C always terminates, and ends up in a state where Q holds.

Auxiliary variables

Consider a program C that computes the maximum value of two variables X and Y and stores the result in a variable Z.

Is this a good specification for C?

$$\{\top\}\ C\ \{(X \leq Y \Rightarrow Z = Y) \land (Y \leq X \Rightarrow Z = X)\}$$

No! Take C to be $X := 0; Y := 0; Z := 0$, then C satisfies the above specification. The postcondition should refer to the **initial** values of X and Y.

In Hoare logic we use **auxiliary variables** which do not occur in the program to refer to the initial value of variables in postconditions.

Auxiliary variables

For instance, $\{X = x \land Y = y\}\ C\ \{X = y \land Y = x\}$ expresses that if C terminates, then it exchanges the values of variables X and Y.

Here, x and y are auxiliary variables (also "ghost variables"), which are not allowed to occur in C, and are only used to name the initial values of X and Y.

Informal convention: program variables are uppercase, and auxiliary variables are lowercase.

Formal proof system for Hoare logic

Hoare logic

We will now introduce a natural deduction proof system for partial correctness triples due to Tony Hoare.

The logic consists of a set of **inference rule schemas** for deriving consequences from premises.

If S is a statement, we will write $\vdash S$ to mean that the statement S is derivable. We will have two derivability judgements:

- $\vdash P$, for derivability of assertions; and
- $\vdash \{P\}\ C\ \{Q\}$, for derivability of partial correctness triples.

Inference rule schemas

The inference rule schemas of Hoare logic will be specified as follows:

$$\frac{\vdash S_1 \qquad \cdots \qquad \vdash S_n}{\vdash S}$$

This expresses that S may be deduced from assumptions S_1, ..., S_n.

These are schemas that may contain meta-variables.

Proof trees

A proof tree for $\vdash S$ in Hoare logic is a tree with $\vdash S$ at the root, constructed using the inference rules of Hoare logic, where all nodes are shown to be derivable, so leaves require no further derivations:

$$\cfrac{\cfrac{\overline{\vdash S_1} \qquad \overline{\vdash S_2}}{\vdash S_3} \qquad \overline{\vdash S_4}}{\vdash S}$$

We typically write proof trees with the root at the bottom.

Formal proof system for Hoare logic

$$\overline{\vdash \{P\}\ \textbf{skip}\ \{P\}} \qquad \overline{\vdash \{P[E/V]\}\ V := E\ \{P\}}$$

$$\frac{\vdash \{P\}\ C_1\ \{Q\} \qquad \vdash \{Q\}\ C_2\ \{R\}}{\vdash \{P\}\ C_1; C_2\ \{R\}}$$

$$\frac{\vdash \{P \wedge B\}\ C_1\ \{Q\} \qquad \vdash \{P \wedge \neg B\}\ C_2\ \{Q\}}{\vdash \{P\}\ \textbf{if}\ B\ \textbf{then}\ C_1\ \textbf{else}\ C_2\ \{Q\}}$$

$$\frac{\vdash \{P \wedge B\}\ C\ \{P\}}{\vdash \{P\}\ \textbf{while}\ B\ \textbf{do}\ C\ \{P \wedge \neg B\}}$$

$$\frac{\vdash P_1 \Rightarrow P_2 \qquad \vdash \{P_2\}\ C\ \{Q_2\} \qquad \vdash Q_2 \Rightarrow Q_1}{\vdash \{P_1\}\ C\ \{Q_1\}}$$

The skip rule

$$\overline{\vdash \{P\}\ \textbf{skip}\ \{P\}}$$

The **skip** rule expresses that any assertion that holds before **skip** is executed also holds afterwards.

P is a meta-variable ranging over an arbitrary state predicate.

For instance, $\vdash \{X = 1\}\ \textbf{skip}\ \{X = 1\}$.

The assignment rule

$$\frac{}{\vdash \{P[E/V]\}\ V := E\ \{P\}}$$

Here, $P[E/V]$ means the assertion P with the expression E substituted for all occurrences of the variable V.

For instance,

$$\{X + 1 = 2\}\ X := X + 1\ \{X = 2\}$$

$$\{Y + X = Y + 10\}\ X := Y + X\ \{X = Y + 10\}$$

The assignment rule

The assignment rule reads right-to-left; could we use another rule that reads more easily?

Consider the following plausible alternative assignment rules:

$$\frac{}{\vdash \{P\}\ V := E\ \{P[E/V]\}}$$

We can instantiate this rule to obtain the following triple, which does not hold:

$$\{X = 0\}\ X := 1\ \{1 = 0\}$$

The rule of consequence

$$\frac{\vdash P_1 \Rightarrow P_2 \qquad \vdash \{P_2\}\ C\ \{Q_2\} \qquad \vdash Q_2 \Rightarrow Q_1}{\vdash \{P_1\}\ C\ \{Q_1\}}$$

The rule of consequence allows us to strengthen preconditions and weaken postconditions.

Note: the $\vdash P \Rightarrow Q$ hypotheses are a different kind of judgment.

For instance, from $\{X + 1 = 2\}\ X := X + 1\ \{X = 2\}$,
we can deduce $\{X = 1\}\ X := X + 1\ \{X = 2\}$.

Sequential composition

$$\frac{\vdash \{P\}\ C_1\ \{Q\} \qquad \vdash \{Q\}\ C_2\ \{R\}}{\vdash \{P\}\ C_1; C_2\ \{R\}}$$

If the postcondition of C_1 matches the precondition of C_2, we can derive a specification for their sequential composition.

For example, if we have deduced:

- $\{X = 1\}\ X := X + 1\ \{X = 2\}$
- $\{X = 2\}\ X := X + 1\ \{X = 3\}$

we may deduce that $\{X = 1\}\ X := X + 1; X := X + 1\ \{X = 3\}$.

The conditional rule

$$\frac{\vdash \{P \wedge B\}\ C_1\ \{Q\} \qquad \vdash \{P \wedge \neg B\}\ C_2\ \{Q\}}{\vdash \{P\}\ \textbf{if}\ B\ \textbf{then}\ C_1\ \textbf{else}\ C_2\ \{Q\}}$$

For instance, to prove that

$$\vdash \{\top\}\ \textbf{if}\ X \geq Y\ \textbf{then}\ Z := X\ \textbf{else}\ Z := Y\ \{Z = max(X, Y)\}$$

It suffices to prove that $\vdash \{\top \wedge X \geq Y\}\ Z := X\ \{Z = max(X, Y)\}$ and $\vdash \{\top \wedge \neg(X \geq Y)\}\ Z := Y\ \{Z = max(X, Y)\}$.

The loop rule

$$\frac{\vdash \{P \wedge B\}\ C\ \{P\}}{\vdash \{P\}\ \textbf{while}\ B\ \textbf{do}\ C\ \{P \wedge \neg B\}}$$

The loop rule says that

- if P is an invariant of the loop body when the loop condition succeeds, then P is an invariant for the whole loop, and
- if the loop terminates, then the loop condition failed.

We will return to be problem of finding loop invariants.

Conjunction and disjunction rule

$$\frac{\vdash \{P_1\}\ C\ \{Q\} \qquad \vdash \{P_2\}\ C\ \{Q\}}{\vdash \{P_1 \vee P_2\}\ C\ \{Q\}}$$

$$\frac{\vdash \{P\}\ C\ \{Q_1\} \qquad \vdash \{P\}\ C\ \{Q_2\}}{\vdash \{P\}\ C\ \{Q_1 \wedge Q_2\}}$$

These rules are useful for splitting up proofs.

Any proof with these rules could be done without using them

- i.e. they are theoretically redundant (proof omitted),
- however, they are useful in practice.

Summary

Hoare logic is a formalism for reasoning about the behaviour of programs by relating their initial and terminal state.

It uses an assertion logic based on first-order logic to reason about program states, and extends this with Hoare triples to reason about the programs.

Papers of historical interest:

- C. A. R. Hoare. An axiomatic basis for computer programming. 1969.
- R. W. Floyd. Assigning meanings to programs. 1967.
- A. M. Turing. Checking a large routine. 1949.

Hoare Logic and Model Checking

Jean Pichon-Pharabod
University of Cambridge

CST Part II – 2017/18

Semantics of Hoare logic

Semantics of Hoare logic

Recall: to define a Hoare logic, we need four main components:

- the programming language that we want to reason about, along with its dynamic semantics;
- an assertion language for defining state predicates, along with a semantics;
- an interpretation of Hoare triples;
- a (sound) proof system for deriving Hoare triples.

This lecture defines a formal semantics of Hoare logic, and introduces some meta-theoretic results about Hoare logic (soundness & completeness).

Dynamic semantics of WHILE

Dynamic semantics of WHILE

The dynamic semantics of WHILE will be given in the form of a "big-step" operational semantics.

The reduction relation, written $\langle C, s\rangle \Downarrow s'$, expresses that the command C reduces to the terminal state s' when executed from initial state s.

Dynamic semantics of WHILE

More precisely, these "states" are stores, which are functions from variables to integers:

$$\mathit{Store} \stackrel{\mathsf{def}}{=} \mathit{Var} \to \mathbb{Z}$$

These are **total** functions, and define the current value of every program and auxiliary variable.

This models WHILE with arbitrary precision integer arithmetic. A more realistic model might use 32-bit integers and require reasoning about overflow, etc.

Dynamic semantics of WHILE

The reduction relation is defined inductively by a set of rules.

To reduce an assignment, we first evaluate the expression E using the current store, and update the store with the value of E:

$$\frac{\mathcal{E}[\![E]\!](s) = n}{\langle V := E, s\rangle \Downarrow s[V \mapsto n]}$$

We use functions $\mathcal{E}[\![E]\!](s)$ and $\mathcal{B}[\![B]\!](s)$ to evaluate arithmetic expressions and boolean expressions in a given store s, respectively.

Semantics of expressions

$\mathcal{E}[\![E]\!](s)$ evaluates arithmetic expression E to an integer in store s:

$$\mathcal{E}[\![-]\!](=) : \mathit{Exp} \times \mathit{Store} \to \mathbb{Z}$$

$$\begin{aligned}
\mathcal{E}[\![N]\!](s) &= N \\
\mathcal{E}[\![V]\!](s) &= s(V) \\
\mathcal{E}[\![E_1 + E_2]\!](s) &= \mathcal{E}[\![E_1]\!](s) + \mathcal{E}[\![E_2]\!](s) \\
&\vdots
\end{aligned}$$

This semantics is too simple to handle operations such as division, which fails to evaluate to an integer on some inputs.

Semantics of boolean expressions

$\mathcal{B}[\![B]\!](s)$ evaluates boolean expression B to an boolean in store s:

$$\mathcal{B}[\![-]\!](=) : BExp \times Store \to \mathbb{B}$$

$$\mathcal{E}[\![T]\!](s) = \top$$
$$\mathcal{E}[\![F]\!](s) = \bot$$
$$\mathcal{E}[\![E_1 \leq E_2]\!](s) = \begin{cases} \top & \text{if } \mathcal{E}[\![E_1]\!](s) \leq \mathcal{E}[\![E_2]\!](s) \\ \bot & \text{otherwise} \end{cases}$$
$$\vdots$$

Big-step operational semantics of WHILE

$$\frac{\mathcal{E}[\![E]\!](s) = n}{\langle V := E, s\rangle \Downarrow s[V \mapsto n]} \qquad \frac{\langle C_1, s\rangle \Downarrow s' \qquad \langle C_2, s'\rangle \Downarrow s''}{\langle C_1; C_2, s\rangle \Downarrow s''}$$

$$\frac{\mathcal{B}[\![B]\!](s) = \top \qquad \langle C_1, s\rangle \Downarrow s'}{\langle \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2, s\rangle \Downarrow s'} \qquad \frac{\mathcal{B}[\![B]\!](s) = \bot \qquad \langle C_2, s\rangle \Downarrow s'}{\langle \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2, s\rangle \Downarrow s'}$$

$$\frac{\mathcal{B}[\![B]\!](s) = \top \qquad \langle C, s\rangle \Downarrow s' \qquad \langle \textbf{while } B \textbf{ do } C, s'\rangle \Downarrow s''}{\langle \textbf{while } B \textbf{ do } C, s\rangle \Downarrow s''}$$

$$\frac{\mathcal{B}[\![B]\!](s) = \bot}{\langle \textbf{while } B \textbf{ do } C, s\rangle \Downarrow s} \qquad \frac{}{\langle \textbf{skip}, s\rangle \Downarrow s}$$

Meta-theory

Note that the dynamic semantics of WHILE is deterministic:

$$\langle C, s\rangle \Downarrow s' \wedge \langle C, s\rangle \Downarrow s'' \Rightarrow s' = s''$$

We have already implicitly used this in the definition of total correctness triples.

Without this property, we would have to specify whether all reductions or just some reductions were required to terminate.

Meta-theory

We will need the following expression substitution property later to prove soundness of the Hoare assignment rule:

$$\boxed{\mathcal{E}[\![E_1[E_2/V]]\!](s) = \mathcal{E}[\![E_1]\!](s[V \mapsto \mathcal{E}[\![E_2]\!](s)])}$$

The expression substitution property follows by induction on E_1.

Case $E_1 \equiv N$:

$$\mathcal{E}[\![N[E_2/V]]\!](s) = N = \mathcal{E}[\![N]\!](s[V \mapsto \mathcal{E}[\![E_2]\!](s)])$$

Meta-theory

$$\mathcal{E}[\![E_1[E_2/V]]\!](s) = \mathcal{E}[\![E_1]\!](s[V \mapsto \mathcal{E}[\![E_2]\!](s)])$$

Case $E_1 \equiv V$:

$$\begin{aligned}\mathcal{E}[\![V'[E_2/V]]\!](s) &= \begin{cases} \mathcal{E}[\![E_2]\!](s) & \text{if } V = V' \\ s(V') & \text{if } V \neq V' \end{cases} \\ &= \mathcal{E}[\![V']\!](s[V \mapsto \mathcal{E}[\![E_2]\!](s)])\end{aligned}$$

Meta-theory

$$\mathcal{E}[\![E_1[E_2/V]]\!](s) = \mathcal{E}[\![E_1]\!](s[V \mapsto \mathcal{E}[\![E_2]\!](s)])$$

Case $E_1 \equiv E_a + E_b$:

$$\begin{aligned}&\mathcal{E}[\![(E_a + E_b)[E_2/V]]\!](s) \\ &\quad = \mathcal{E}[\![E_a[E_2/V]]\!](s) + \mathcal{E}[\![E_b[E_2/V]]\!](s) \\ &\quad = \mathcal{E}[\![E_a]\!](s[V \mapsto \mathcal{E}[\![E_2]\!](s)]) + \mathcal{E}[\![E_b]\!](s[V \mapsto \mathcal{E}[\![E_2]\!](s)]) \\ &\quad = \mathcal{E}[\![E_a + E_b]\!](s[V \mapsto \mathcal{E}[\![E_2]\!](s)])\end{aligned}$$

Semantics of assertions

The language of assertions

Now, we have formally defined the semantics of the WHILE language that we wish to reason about.

The next step is to formalise the assertion language that we will use to reason about states of WHILE programs.

We take the language of assertions to be an instance of (single-sorted) first-order logic with equality.

Knowledge of first-order logic is assumed. We will review some basic concepts now.

Review of first-order logic

Recall that in first-order logic there are two syntactic classes:

- terms, which denote values (e.g., numbers), and
- assertions, which denote properties that may be true or false.

Assertions are built out of predicates on terms, and logical connectives ($\wedge$, $\vee$, etc.).

Since we are reasoning about WHILE states, our assertions will describe properties of WHILE states.

Review of first-order logic: Terms

Terms may contain variables like x, X, y, Y, z, Z etc.

Terms, like 1 and $4 + 5$, that do not contain any free variables are called ground terms.

We use conventional notation, e.g. here are some terms:

$$\begin{gathered} X, \quad y, \quad Z, \\ 1, \quad 2, \quad 325, \\ -X, \quad -(X+1), \quad (x \times y) + Z, \\ \sqrt{(1+x^2)}, \quad X!, \quad sin(x), \quad rem(X, Y) \end{gathered}$$

Review of first-order logic: Atomic assertions

Examples of atomic assertions are:

$$\bot, \quad \top, \quad X = 1, \quad R < Y, \quad X = R + (Y \times Q)$$

$\top$ and $\bot$ are atomic assertions that are always (respectively) true and false.

Other atomic assertions are built from terms using predicates, e.g.

$$ODD(X), \quad PRIME(3), \quad X = 1, \quad (X+1)^2 \geq x^2$$

Here ODD, $PRIME$, and $\geq$ are examples of predicates ($\geq$ is written using infix notation) and X, 1, 3, $X + 1$, $(X + 1)^2$ and x^2 are examples of terms.

Review of first-order logic: Atomic assertions

In general, first-order logic is parameterised over a signature that defines non-logical function symbols ($+, -, \times, ...$) and predicate symbols (ODD, $PRIME$, etc.).

We will be using a particular instance with a signature that includes the usual functions and predicates on integers.

Review of first-order logic: Compound assertions

Compound assertions are built up from atomic assertions using the usual logical connectives:

$$\wedge\ (conjunction), \vee\ (disjunction), \Rightarrow\ (implication)$$

and quantification:

$$\forall\ (universal), \exists\ (existential)$$

Negation, $\neg P$, is a shorthand for $P \Rightarrow \bot$.

The assertion language

The formal syntax of the assertion language is given below:

$$\begin{array}{rcll} P, Q & ::= & \bot \mid \top \mid B \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q & \textit{assertions} \\ & \mid & \forall x.\, P \mid \exists x.\, P \mid t_1 = t_2 \mid p(t_1, \ldots, t_n) & \\ & & & \\ t & ::= & E \mid f(t_1, \ldots, t_n) & \textit{terms} \end{array}$$

Note that assertions quantify over logical variables.

Here p and f range over an unspecified set of predicates and functions, respectively, that includes the usual mathematical operations on integers.

Semantics of terms

$[\![t]\!](s)$ defines the meaning of a term t in a store s:

$$[\![-]\!](=) : \mathit{Term} \times \mathit{Store} \to \mathbb{Z}$$

$$\begin{aligned} [\![E]\!](s) &\stackrel{\mathit{def}}{=} \mathcal{E}[\![E]\!](s) \\ [\![f(t_1, \ldots, t_n)]\!](s) &\stackrel{\mathit{def}}{=} [\![f]\!]([\![t_1]\!](s), \ldots, [\![t_n]\!](s)) \end{aligned}$$

We assume $[\![f]\!]$ is given by the implicit signature.

Semantics of assertions

$[\![P]\!]$ defines the set of stores that satisfy the assertion P:

$$[\![-]\!] : \mathit{Assertion} \to \mathcal{P}(\mathit{Store})$$

$$\begin{aligned} [\![\bot]\!] &= \emptyset \\ [\![\top]\!] &= \mathit{Store} \\ [\![B]\!] &= \{s \mid \mathcal{B}[\![B]\!](s) = \top\} \\ [\![P \vee Q]\!] &= [\![P]\!] \cup [\![Q]\!] \\ [\![P \wedge Q]\!] &= [\![P]\!] \cap [\![Q]\!] \\ [\![P \Rightarrow Q]\!] &= \{s \mid s \in [\![P]\!] \Rightarrow s \in [\![Q]\!]\} \end{aligned}$$

Semantics of assertions (continued)

$$
\begin{aligned}
[\![\forall x.\, P]\!] &= \{s \mid \forall v.\, s[x \mapsto v] \in [\![P]\!]\} \\
[\![\exists x.\, P]\!] &= \{s \mid \exists v.\, s[x \mapsto v] \in [\![P]\!]\} \\
[\![t_1 = t_2]\!] &= \{s \mid [\![t_1]\!](s) = [\![t_2]\!](s)\} \\
[\![p(t_1, ..., t_n)]\!] &= \{s \mid [\![p]\!]([\![t_1]\!](s), ..., [\![t_2]\!](s))\}
\end{aligned}
$$

We assume $[\![p]\!]$ is given by the implicit signature.

This interpretation is related to the forcing relation you used in Part IB "Logic and Proof":

$$s \in [\![P]\!] \Leftrightarrow s \models P$$

Substitutions

We use $t[E/V]$ and $P[E/V]$ to denote t and P with E substituted for every occurrence of program variable V, respectively.

Since our quantifiers bind logical variables, and all free variables in E are program variables, there is no issue with variable capture.

Substitution property

The term and assertion semantics satisfy a similar substitution property to the expression semantics:

- $[\![t[E/V]]\!](s) = [\![t]\!](s[V \mapsto \mathcal{E}[\![E]\!](s)])$
- $s \in [\![P[E/V]]\!] \Leftrightarrow s[V \mapsto \mathcal{E}[\![E]\!](s)] \in [\![P]\!]$

They are easily provable by induction on t and P, respectively. (Exercise)

Semantics of Hoare logic

Semantics of partial correctness triples

Now that we have formally defined the dynamic semantics of WHILE and our assertion language, we can define the formal meaning of our triples.

A partial correctness triple asserts that if the given command terminates when executed from an initial state that satisfies the precondition, then the terminal state must satisfy the postcondition:

$$\models \{P\}\ C\ \{Q\} \stackrel{\mathit{def}}{=} \forall s, s'.\, s \in [\![P]\!] \wedge \langle C, s\rangle \Downarrow s' \Rightarrow s' \in [\![Q]\!]$$

Semantics of total correctness triples

A total correctness triple asserts that when the given command is executed from an initial state that satisfies the precondition, then it must terminate in a terminal state that satisfies the postcondition:

$$\models [P]\ C\ [Q] \stackrel{\mathit{def}}{=} \forall s.\, s \in [\![P]\!] \Rightarrow \exists s'.\, \langle C, s\rangle \Downarrow s' \wedge s' \in [\![Q]\!]$$

Since WHILE is deterministic, if one terminating execution satisfies the postcondition, then all terminating executions satisfy the postcondition.

Meta-theory of Hoare logic

Now, we have a syntactic proof system for deriving Hoare triples, $\vdash \{P\}\ C\ \{Q\}$, and a formal definition of the meaning of our Hoare triples, $\models \{P\}\ C\ \{Q\}$.

How are these related?

We might hope that any triple that can be derived syntactically holds semantically (soundness), and that any triple that holds semantically is syntactically derivable (completeness).

This is **not** the case: Hoare logic is sound but not complete.

Soundness of Hoare logic

Theorem (Soundness)
If $\vdash \{P\}\ C\ \{Q\}$ *then* $\models \{P\}\ C\ \{Q\}$.

Soundness expresses that any triple derivable using the syntactic proof system holds semantically.

Soundness can be proved by induction on the $\vdash \{P\}\ C\ \{Q\}$ derivation:

- it suffices to show, for each inference rule, that if each hypothesis holds semantically, then the conclusion holds semantically.

Soundness of the assignment rule

$$\models \{P[E/V]\}\ V := E\ \{P\}$$

Assume $s \in [\![P[E/V]]\!]$ and $\langle V := E, s\rangle \Downarrow s'$.

From the substitution property, it follows that

$$s[V \mapsto \mathcal{E}[\![E]\!](s)] \in [\![P]\!]$$

and from the reduction relation, it follows that $s' = s[V \mapsto \mathcal{E}[\![E]\!](s)]$. Hence, $s' \in [\![P]\!]$.

Soundness of the loop inference rule

If $\models \{P \wedge B\}\ C\ \{P\}$ then $\models \{P\}$ **while** B **do** $C\ \{P \wedge \neg B\}$

Assume $\models \{P \wedge B\}\ C\ \{P\}$.

We will prove $\models \{P\}$ **while** B **do** $C\ \{P \wedge \neg B\}$ by proving the following stronger property by strong induction on n:

$$\forall n. \forall s, s'. s \in [\![P]\!] \wedge \langle \textbf{while } B \textbf{ do } C, s\rangle \Downarrow^n s' \Rightarrow s' \in [\![P \wedge \neg B]\!]$$

where $\langle C, s\rangle \Downarrow^n s'$ indicates a reduction in n steps.

Soundness of the loop inference rule

If $\models \{P \wedge B\}\ C\ \{P\}$, then
$\forall n. \forall s, s'. s \in [\![P]\!] \wedge \langle \textbf{while } B \textbf{ do } C, s\rangle \Downarrow^n s' \Rightarrow s' \in [\![P \wedge \neg B]\!]$

Case $n = 1$: assume $s \in [\![P]\!]$ and $\langle \textbf{while } B \textbf{ do } C, s\rangle \Downarrow^1 s'$. Since the loop reduced in one step, B must have evaluated to false: $\mathcal{B}[\![B]\!](s) = \bot$ and $s' = s$. Hence, $s' = s \in [\![P \wedge \neg B]\!]$.

Case $n > 1$: assume $s \in [\![P]\!]$ and $\langle \textbf{while } B \textbf{ do } C, s\rangle \Downarrow^n s'$. Since the loop reduced in more than one step, B must have evaluated to true: $\mathcal{B}[\![B]\!](s) = \top$ and there exists an s'', n_1 and n_2 such that $\langle C, s\rangle \Downarrow^{n_1} s''$, $\langle \textbf{while } B \textbf{ do } C, s''\rangle \Downarrow^{n_2} s'$ with $n = n_1 + n_2 + 1$.

From the $\models \{P \wedge B\}\ C\ \{P\}$ assumption, it follows that $s'' \in [\![P]\!]$, and by the induction hypothesis, $s' \in [\![P \wedge \neg B]\!]$.

Completeness

Completeness is the converse property of soundness:
If $\models \{P\}\ C\ \{Q\}$ then $\vdash \{P\}\ C\ \{Q\}$.

Hoare logic inherits the incompleteness of first-order logic and is therefore **not** complete.

Completeness

To see why, consider the triple $\{\top\}$ **skip** $\{P\}$.

By unfolding the meaning of this triple, we get:

$$\models \{\top\}\ \textbf{skip}\ \{P\} \Leftrightarrow \forall s.\, s \in [\![P]\!]$$

If could deduce any true triple using Hoare logic, we would be able to deduce any true statement of the assertion logic using Hoare logic.

Since the assertion logic (first-order logic) is **not** complete, this is not the case.

Relative completeness

The previous argument showed that because the assertion logic is not complete, then neither is Hoare logic.

However, Hoare logic is **relatively complete** for our simple language:

- Relative completeness expresses that any failure to prove $\vdash \{P\}\ C\ \{Q\}$ for a valid statement $\models \{P\}\ C\ \{Q\}$ can be traced back to a failure to prove $\vdash R$ for some valid arithmetic statement R.

Decidability

Finally, Hoare logic is not decidable.

The triple $\{\top\}\ C\ \{\bot\}$ holds if and only if C does not terminate. Hence, since the Halting problem is undecidable, so is Hoare logic.

Summary

We have defined a dynamic semantics for the WHILE language, and a formal semantics for a Hoare logic for WHILE.

We have shown that the formal Hoare logic proof system from the last lecture is sound with respect to this semantics, but not complete.

Supplementary reading on soundness and completeness:

- Glynn Winskel. The Formal Semantics of Programming Languages: An Introduction. Chapters 6–7.

Hoare Logic and Model Checking

Jean Pichon-Pharabod
University of Cambridge

CST Part II – 2017/18

Introduction

In the past lectures, we have given

- a notation for specifying the intended behaviour of programs;
- a proof system for proving that programs satisfy their intended specification; and
- a semantics capturing the precise meaning of this notation

Now, we are going to look at ways of finding proofs, including:

- derived rules & backwards reasoning;
- finding invariants; and
- ways of annotating programs prior to proving.

We are also going to look at proof rules for total correctness.

Forward and backwards reasoning

Forward & backwards reasoning

The proof rules we have seen so far are best suited for **forward** directed reasoning, where a proof tree is constructed starting from the leaves, going towards the root.

For instance, consider a proof of

$$\vdash \{X = a\}\ X := X + 1\ \{X = a + 1\}$$

using the assignment rule:

$$\frac{}{\vdash \{P[E/V]\}\ V := E\ \{P\}}$$

Forward reasoning

It is often more natural to work **backwards**, starting from the root of the proof tree, and generating new subgoals until all the nodes have been shown to be derivable.

We can **derive** rules better suited for backwards reasoning.

For instance, we can derive this backwards assignment rule:

$$\frac{\vdash P \Rightarrow Q[E/V]}{\vdash \{P\}\ V := E\ \{Q\}}$$

Backwards sequenced assignment rule

The sequence rule can already be applied bottom up, but requires us to guess an assertion R:

$$\frac{\vdash \{P\}\ C_1\ \{R\} \qquad \vdash \{R\}\ C_2\ \{Q\}}{\vdash \{P\}\ C_1; C_2\ \{Q\}}$$

In the case of a command sequenced before an assignment, we can avoid having to guess R with the sequenced assignment rule:

$$\frac{\vdash \{P\}\ C\ \{Q[E/V]\}}{\vdash \{P\}\ C; V := E\ \{Q\}}$$

This is easily derivable using the sequencing rule and the backwards assignment rule (exercise).

Backwards reasoning

In the same way, we can derive a backwards reasoning rule for loops by building in consequence:

$$\frac{\vdash P \Rightarrow I \qquad \vdash \{I \wedge B\}\ C\ \{I\} \qquad \vdash I \wedge \neg B \Rightarrow Q}{\vdash \{P\}\ \textbf{while}\ B\ \textbf{do}\ C\ \{Q\}}$$

This rule still requires us to guess I to apply it bottom-up.

Backwards reasoning proof rules

$$\frac{\vdash P \Rightarrow Q}{\vdash \{P\}\ \textbf{skip}\ \{Q\}} \qquad \frac{\vdash \{P\}\ C_1\ \{R\} \qquad \vdash \{R\}\ C_2\ \{Q\}}{\vdash \{P\}\ C_1; C_2\ \{Q\}}$$

$$\frac{\vdash P \Rightarrow Q[E/V]}{\vdash \{P\}\ V := E\ \{Q\}} \qquad \frac{\vdash \{P\}\ C\ \{Q[E/V]\}}{\vdash \{P\}\ C; V := E\ \{Q\}}$$

$$\frac{\vdash P \Rightarrow I \qquad \vdash \{I \wedge B\}\ C\ \{I\} \qquad \vdash I \wedge \neg B \Rightarrow Q}{\vdash \{P\}\ \textbf{while}\ B\ \textbf{do}\ C\ \{Q\}}$$

$$\frac{\vdash \{P \wedge B\}\ C_1\ \{Q\} \qquad \vdash \{P \wedge \neg B\}\ C_2\ \{Q\}}{\vdash \{P\}\ \textbf{if}\ B\ \textbf{then}\ C_1\ \textbf{else}\ C_2\ \{Q\}}$$

Finding loop invariants

A verified factorial implementation

We wish to verify that the following command computes the factorial of X and stores the result in Y.

$$\textbf{while } X \neq 0 \textbf{ do } (Y := Y \times X; X := X - 1)$$

First we need to formalise the specification:

- Factorial is only defined for non-negative numbers, so X should be non-negative in the initial state.
- The terminal state of Y should be equal to the factorial of the initial state of X.
- The implementation assumes that Y is equal to 1 initially.

A verified factorial implementation

This corresponds to the following partial correctness Hoare triple:

$$\begin{array}{l} \{X = x \wedge X \geq 0 \wedge Y = 1\} \\ \quad \textbf{while } X \neq 0 \textbf{ do } (Y := Y \times X; X := X - 1) \\ \{Y = x!\} \end{array}$$

Here, '!' denotes the usual mathematical factorial function.

Note that we used an auxiliary variable x to record the initial value of X and relate the terminal value of Y with the initial value of X.

How does one find an invariant?

$$\frac{\vdash P \Rightarrow I \qquad \vdash \{I \wedge B\}\ C\ \{I\} \qquad \vdash I \wedge \neg B \Rightarrow Q}{\vdash \{P\}\ \textbf{while } B \textbf{ do } C\ \{Q\}}$$

Here, I is an invariant that

- must hold initially;
- must be preserved by the loop body when B is true; and
- must imply the desired postcondition when B is false.

How does one find an invariant?

$$\frac{\vdash P \Rightarrow I \qquad \vdash \{I \wedge B\}\ C\ \{I\} \qquad \vdash I \wedge \neg B \Rightarrow Q}{\vdash \{P\}\ \textbf{while}\ B\ \textbf{do}\ C\ \{Q\}}$$

The invariant I should express

- what **has been done so far** and what **remains to be done**;
- that nothing has been done initially; and
- that nothing remains to be done when B is false.

A verified factorial implementation

$\{X = x \wedge X \geq 0 \wedge Y = 1\}$
  $\textbf{while}\ X \neq 0\ \textbf{do}\ (Y := Y \times X; X := X - 1)$
$\{Y = x!\}$

Take I to be $Y \times X! = x! \wedge X \geq 0$. Then it suffices to prove

- $X = x \wedge X \geq 0 \wedge Y = 1 \Rightarrow I$
- $\{I \wedge X \neq 0\}\ Y := Y \times X; X := X - 1\ \{I\}$
- $I \wedge X = 0 \Rightarrow Y = x!$

The first and last proof obligation follow by basic arithmetic.

Proof rules

$$\frac{\vdash P \Rightarrow Q}{\vdash \{P\}\ \textbf{skip}\ \{Q\}} \qquad \frac{\vdash \{P\}\ C_1\ \{R\} \qquad \vdash \{R\}\ C_2\ \{Q\}}{\vdash \{P\}\ C_1; C_2\ \{Q\}}$$

$$\frac{\vdash P \Rightarrow Q[E/V]}{\vdash \{P\}\ V := E\ \{Q\}} \qquad \frac{\vdash \{P\}\ C\ \{Q[E/V]\}}{\vdash \{P\}\ C; V := E\ \{Q\}}$$

$$\frac{\vdash P \Rightarrow I \qquad \vdash \{I \wedge B\}\ C\ \{I\} \qquad \vdash I \wedge \neg B \Rightarrow Q}{\vdash \{P\}\ \textbf{while}\ B\ \textbf{do}\ C\ \{Q\}}$$

$$\frac{\vdash \{P \wedge B\}\ C_1\ \{Q\} \qquad \vdash \{P \wedge \neg B\}\ C_2\ \{Q\}}{\vdash \{P\}\ \textbf{if}\ B\ \textbf{then}\ C_1\ \textbf{else}\ C_2\ \{Q\}}$$

Proof outlines

In the literature, hand-written proofs in Hoare logic are often written as informal **proof outlines** instead of proof trees.

Proof outlines are code listings annotated with Hoare logic assertions between statements.

Proof outlines

Here is an example of a proof outline for the second proof obligation for the factorial function:

$$
\begin{array}{l}
\{Y \times X! = x! \wedge X \geq 0 \wedge X \neq 0\} \\
\{(Y \times X) \times (X - 1)! = x! \wedge (X - 1) \geq 0\} \\
\quad Y := Y \times X; \\
\{Y \times (X - 1)! = x! \wedge (X - 1) \geq 0\} \\
\quad X := X - 1 \\
\{Y \times X! = x! \wedge X \geq 0\}
\end{array}
$$

Proof outlines

Writing out full proof trees or proof outlines by hand is tedious and error-prone even for simple programs.

In the next lecture, we will look at using mechanisation to check our proofs and help discharge trivial proof obligations.

A verified fibonacci implementation

Imagine we want to prove that the following fibonacci implementation satisfies the given specification:

$$
\begin{array}{l}
\{X = 0 \wedge Y = 1 \wedge Z = 1 \wedge 1 \leq N \wedge N = n\} \\
\textbf{while } (Z < N) \textbf{ do} \\
\quad (Y := X + Y; X := Y - X; Z := Z + 1) \\
\{Y = \mathit{fib}(n)\}
\end{array}
$$

First we need to understand the implementation:

- the Z variable is used to count loop iterations; and
- Y and X are used to compute the fibonacci number.

A verified fibonacci implementation

$$
\begin{array}{l}
\{X = 0 \wedge Y = 1 \wedge Z = 1 \wedge 1 \leq N \wedge N = n\} \\
\textbf{while } (Z < N) \textbf{ do} \\
\quad (Y := X + Y; X := Y - X; Z := Z + 1) \\
\{Y = \mathit{fib}(n)\}
\end{array}
$$

Take $I \equiv Y = \mathit{fib}(Z) \wedge X = \mathit{fib}(Z - 1)$, then we have to prove:

- $X = 0 \wedge Y = 1 \wedge Z = 1 \wedge 1 \leq N \wedge N = n \Rightarrow I$
- $\{I \wedge (Z < N)\}\ Y := X + Y; X := Y - X; Z := Z + 1\ \{I\}$
- $(I \wedge \neg(Z < N)) \Rightarrow Y = \mathit{fib}(n)$

Do all these hold? The first two do (Exercise!)

A verified fibonacci implementation

$$\{X = 0 \land Y = 1 \land Z = 1 \land 1 \leq N \land N = n\}$$
$$\textbf{while } (Z < N) \textbf{ do}$$
$$(Y := X + Y; X := Y - X; Z := Z + 1)$$
$$\{Y = \mathit{fib}(n)\}$$

While $Y = \mathit{fib}(Z) \land X = \mathit{fib}(Z - 1)$ **is an invariant**, it is not strong enough to establish the desired post-condition.

We need to know that when the loop terminates, then $Z = n$.
It suffices to strengthen the invariant to:

$$Y = \mathit{fib}(Z) \land X = \mathit{fib}(Z - 1) \land Z \leq N \land N = n$$

Total correctness

Total correctness

So far, we have many concerned ourselves with partial correctness.
What about total correctness?

Recall, total correctness = partial correctness + termination.

The total correctness triple, $[P]\ C\ [Q]$ holds if and only if

- whenever C is executed in a state satisfying P, then C terminates, and the terminal state satisfies Q.

Total correctness

while commands are the only commands that might not terminate.

Except for the **while** rule, all the rules described so far are sound for total correctness as well as partial correctness.

Total correctness

The **while** rule is not sound for total correctness:

$$\dfrac{\dfrac{\dfrac{}{\vdash \{\top\}\ X := X\ \{\top\}}}{\vdash \{\top \wedge \top\}\ X := X\ \{\top\}}}{\vdash \{\top\}\ \textbf{while true do}\ X := X\ \{\top \wedge \neg\top\}} \quad \vdash \top \wedge \neg\top \Rightarrow \bot$$
$$\overline{\vdash \{\top\}\ \textbf{while true do}\ X := X\ \{\bot\}}$$

If the **while** rule were sound for total correctness, then this would show that **while true do** $X := X$ always terminates in a state satisfying $\bot$.

Total correctness

We need an alternative total correctness **while** rule that ensures the loop always terminates.

The idea is to show that some non-negative quantity decreases on each iteration of the loop.

This decreasing quantity is called a variant.

Total correctness

In the rule below, the variant is E, and the fact that it decreases is specified with an auxiliary variable n:

$$\frac{\vdash [P \wedge S \wedge (E = n)]\ C\ [P \wedge (E < n)] \qquad \vdash P \wedge S \Rightarrow E \geq 0}{\vdash [P]\ \textbf{while}\ S\ \textbf{do}\ C\ [P \wedge \neg S]}$$

The second hypothesis ensures the variant is non-negative.

Total correctness

Using the rule of consequence, we can derive the following backwards reasoning total correctness **while** rule:

$$\frac{\begin{array}{c}\vdash P \Rightarrow I \qquad \vdash I \wedge \neg S \Rightarrow Q \\ \vdash I \wedge S \Rightarrow E \geq 0 \qquad \vdash [I \wedge S \wedge (E = n)]\ C\ [I \wedge (E < n)]\end{array}}{\vdash [P]\ \textbf{while}\ S\ \textbf{do}\ C\ [Q]}$$

Total correctness: Factorial example

Consider the factorial computation we looked at before:

$$
\begin{array}{l}
[X = x \wedge X \geq 0 \wedge Y = 1] \\
\quad \textbf{while } X \neq 0 \textbf{ do } (Y := Y \times X; X := X - 1) \\
[Y = x!]
\end{array}
$$

By assumption, X is non-negative and decreases in each iteration of the loop.

To verify that this factorial implementation terminates, we can thus take the variant E to be X.

Total correctness: Factorial example

$$
\begin{array}{l}
[X = x \wedge X \geq 0 \wedge Y = 1] \\
\quad \textbf{while } X \neq 0 \textbf{ do } (Y := Y \times X; X := X - 1) \\
[Y = x!]
\end{array}
$$

Take I to be $Y \times X! = x! \wedge X \geq 0$, and E to be X.

Then we have to show that

- $X = x \wedge X \geq 0 \wedge Y = 1 \Rightarrow I$
- $[I \wedge X \neq 0 \wedge (X = n)]\ Y := Y \times X; X := X - 1\ [I \wedge (X < n)]$
- $I \wedge X = 0 \Rightarrow Y = x!$
- $I \wedge X \neq 0 \Rightarrow X \geq 0$

Total correctness

The relation between partial and total correctness is informally given by the equation

Total correctness = partial correctness + termination

This is captured formally by the following inference rules:

$$
\frac{\vdash \{P\}\ C\ \{Q\} \qquad \vdash [P]\ C\ [\top]}{\vdash [P]\ C\ [Q]} \qquad\qquad \frac{\vdash [P]\ C\ [Q]}{\vdash \{P\}\ C\ \{Q\}}
$$

Summary: Total correctness

We have given rules for total correctness.

They are similar to those for partial correctness.

The main difference is in the **while** rule:

- **while** commands are the only ones that can fail to terminate;
- for **while** commands, we must prove that a non-negative expression is decreased by the loop body.

Hoare Logic and Model Checking

Jean Pichon-Pharabod
University of Cambridge

CST Part II – 2017/18

Mechanised Program Verification

It is clear that proofs can be long and boring even if the programs being verified are quite simple.

In this lecture, we will sketch the architecture of a simple semi-automated program verifier, and justify it using the rules of Hoare logic.

Our goal is automate the routine parts of proofs in Hoare logic.

Mechanisation

Recall that it is impossible to design a decision procedure to decide automatically the truth or falsehood of any arbitrary mathematical statement.

This does not mean that one cannot have procedures that will prove many useful statements.

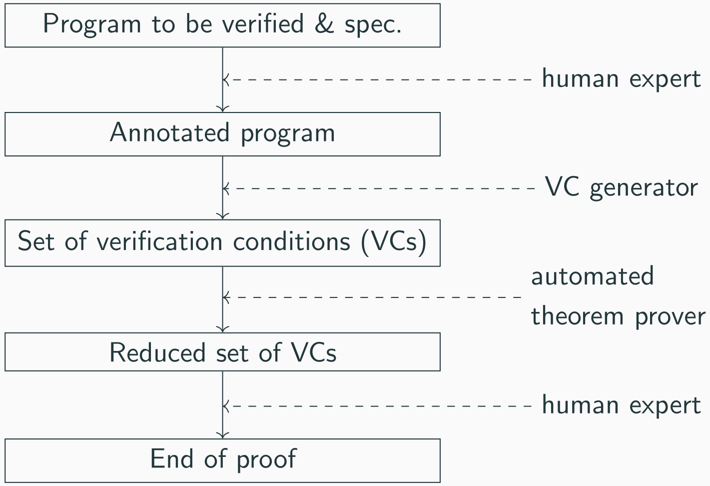
In practice, it is quite possible to build a system that will mechanise the boring and routine aspects of verification

Mechanisation

The standard approach to this will be described in the course.

- The ideas are very old:
  JC King's 1969 CMU PhD, Stanford verifier in 1970s.
- This approach is used by actual program verifiers,
  e.g. Gypsy and SPARK verifiers.
- It provides a verification front end to different provers
  (see the Why3 system).

Architecture of a verifier



VC generator

The VC generator takes as input an annotated program along with the desired specification.

From these inputs, it generates a set of verification conditions (VCs) expressed in first-order logic.

These VCs have the property that if they hold, then the original program satisfies the desired specification.

Since the VCs are expressed in first-order logic, we can use standard FOL theorem provers to discharge VCs.

Using a verifier

The three steps in proving $\{P\}\ C\ \{Q\}$ with a verifier:

1. The program C is **annotated** by inserting assertions expressing conditions that are meant to hold whenever execution reaches the given annotation.
2. A set of logical statements called verification conditions is then generated from the annotated program and desired specification.
3. An automated theorem prover attempts to prove as many of the verification conditions it can, leaving the rest to the user.

Using a verifier

Verifiers are not a silver bullet!

- Inserting appropriate annotations is tricky, and requires a good understanding of how the program works.
- The verification conditions left over from step 3 may bear little resemblance to annotations and specification written by the user.

Example

We will illustrate the process with the following example:

$$\begin{array}{l} \{\top\} \\ \quad R := X;\, Q := 0; \\ \quad \textbf{while } Y \leq R \textbf{ do} \\ \qquad (R := R - Y;\, Q := Q + 1) \\ \{X = R + Y \times Q \wedge R < Y\} \end{array}$$

Example

Step 1 is to annotate the program with two assertions, ϕ_1 and ϕ_2:

$$\begin{array}{l} \{\top\} \\ \quad R := X;\, Q := 0;\, \{R = X \wedge Q = 0\} \longleftarrow \phi_1 \\ \quad \textbf{while } Y \leq R \textbf{ do } \{X = R + Y \times Q\} \longleftarrow \phi_2 \\ \qquad (R := R - Y;\, Q := Q + 1) \\ \{X = R + Y \times Q \wedge R < Y\} \end{array}$$

The annotations ϕ_1 and ϕ_2 state conditions which are intended to hold whenever control reaches them.

Control reaches ϕ_1 once, and reaches ϕ_2 each time the loop body is executed; ϕ_2 should thus be a loop invariant.

Example

Step 2 will generate the following four VCs for our example:

1. $\top \Rightarrow (X = X \wedge 0 = 0)$
2. $(R = X \wedge Q = 0) \Rightarrow (X = R + (Y \times Q))$
3. $(X = R+(Y\times Q))\wedge Y \leq R) \Rightarrow (X = (R-Y)+(Y\times(Q+1)))$
4. $(X = R+(Y\times Q))\wedge\neg(Y \leq R) \Rightarrow (X = R+(Y\times Q)\wedge R < Y)$

Note that these are statements of arithmetic: the constructs of our programming language have been “compiled away”.

Step 3 uses an automated theorem prover to discharge as many VCs as possible, and lets the user prove the rest manually.

Annotation of commands

An annotated command is a command with extra assertions embedded within it.

A command is **properly annotated** if assertions have been inserted at the following places:

- just before C2 in C1;C2 if C2 is not an assignment command;
- just after the word **do** in **while** commands.

The inserted assertions should express the conditions one expects to hold whenever control reaches the assertion.

Annotation of specifications

A properly annotated specification is a specification $\{P\}\ C\ \{Q\}$ where C is a properly annotated command.

Example: To be properly annotated, assertions should be at points ℓ_1 and ℓ_2 of the specification below:

$$
\begin{array}{l}
\{X = n\} \\
\quad Y := 1; \longleftarrow \ell_1 \\
\quad \textbf{while } X = 0 \textbf{ do} \longleftarrow \ell_2 \\
\quad\quad (Y := Y \times X; X := X - 1) \\
\{X = 0 \wedge Y = n!\}
\end{array}
$$

Generating VCs

Next, we need to specify the VC generator.

We will specify it as a function $VC(P, C, Q)$ that gives a set of verification conditions for a properly annotated specification.

The function will be defined by recursion on C, and is easily implementable.

Backwards reasoning proof rules (recap)

$$
\frac{\vdash P \Rightarrow Q}{\vdash \{P\}\ \textbf{skip}\ \{Q\}}
\qquad
\frac{\vdash \{P\}\ C_1\ \{R\} \qquad \vdash \{R\}\ C_2\ \{Q\}}{\vdash \{P\}\ C_1; C_2\ \{Q\}}
$$

$$
\frac{\vdash P \Rightarrow Q[E/V]}{\vdash \{P\}\ V := E\ \{Q\}}
\qquad
\frac{\vdash \{P\}\ C\ \{Q[E/V]\}}{\vdash \{P\}\ C; V := E\ \{Q\}}
$$

$$
\frac{\vdash P \Rightarrow I \qquad \vdash \{I \wedge B\}\ C\ \{I\} \qquad \vdash I \wedge \neg B \Rightarrow Q}{\vdash \{P\}\ \textbf{while}\ B\ \textbf{do}\ C\ \{Q\}}
$$

$$
\frac{\vdash \{P \wedge B\}\ C_1\ \{Q\} \qquad \vdash \{P \wedge \neg B\}\ C_2\ \{Q\}}{\vdash \{P\}\ \textbf{if}\ B\ \textbf{then}\ C_1\ \textbf{else}\ C_2\ \{Q\}}
$$

Justification of VCs

To prove soundness of the verifier, the VC generator should have the property that if all the VCs generated for $\{P\}\ C\ \{Q\}$ hold, then $\vdash \{P\}\ C\ \{Q\}$ should be derivable in Hoare Logic.

Formally,

$$\forall C, P, Q.\, (\forall \phi \in VC(P, C, Q). \vdash \phi) \Rightarrow (\vdash \{P\}\ C\ \{Q\})$$

This will be proven by induction on C.

VC for assignments

$$VC(P, V := E, Q) \stackrel{def}{=} \{P \Rightarrow Q[E/V]\}$$

Example: The verification condition for

$$\{X = 0\}\ X := X + 1\ \{X = 1\}$$

is $X = 0 \Rightarrow (X + 1) = 1$.

VC for assignments

To justify the VCs generated for an assignment, we need to show

$$\text{if} \vdash P \Rightarrow Q[E/V] \text{ then} \vdash \{P\}\ V := E\ \{Q\}$$

which holds by the backwards reasoning assignment rule.

This is one of the base cases for the inductive proof of

$$(\forall \phi \in VC(P, C, Q). \vdash \phi) \Rightarrow (\vdash \{P\}\ C\ \{Q\})$$

VCs for conditionals

$$\begin{aligned} &VC(P, \textbf{if } S \textbf{ then } C_1 \textbf{ else } C_2, Q) \stackrel{def}{=} \\ &\quad VC(P \wedge S, C_1, Q) \cup VC(P \wedge \neg S, C_2, Q) \end{aligned}$$

Example: The verification conditions for

$$\{\top\}\ \textbf{if } X \geq Y \textbf{ then } R := X \textbf{ else } R := Y\ \{R = max(X, Y)\}$$

are

- the VCs for $\{\top \wedge X \geq Y\}\ R := X\ \{R = max(X, Y)\}$, and
- the VCs for $\{\top \wedge \neg(X \geq Y)\}\ R := Y\ \{R = max(X, Y)\}$

VCs for conditionals

To justify the VCs generated for a conditional, we need to show

$$\psi(C_1) \wedge \psi(C_2) \Rightarrow \psi(\textbf{if } S \textbf{ then } C_1 \textbf{ else } C_2)$$

where

$$\psi(C) \stackrel{def}{=} \forall P, Q. (\forall \phi \in VC(P, C, Q). \vdash \phi) \Rightarrow (\vdash \{P\}\ C\ \{Q\})$$

This is one of the inductive cases of the proof, and $\psi(C_1)$ and $\psi(C_2)$ are the induction hypotheses.

VCs for conditionals

Let $\psi(C) \stackrel{def}{=} \forall P, Q. (\forall \phi \in VC(P, C, Q). \vdash \phi) \Rightarrow (\vdash \{P\}\ C\ \{Q\})$

Assume $\psi(C_1)$, $\psi(C_2)$. To show that $\psi(\textbf{if } S \textbf{ then } C_1 \textbf{ else } C_2)$, assume $\forall \phi \in VC(P, \textbf{if } S \textbf{ then } C_1 \textbf{ else } C_2, Q). \vdash \phi$

From the definition of $VC(P, \textbf{if } S \textbf{ then } C_1 \textbf{ else } C_2, Q)$, it follows that $\forall \phi \in VC(P \wedge S, C_1, Q). \vdash \phi$ and $\forall \phi \in VC(P \wedge \neg S, C_2, Q). \vdash \phi$

By the induction hypotheses $\psi(C_1)$ and $\psi(C_2)$, it follows that $\vdash \{P \wedge S\}\ C_1\ \{Q\}$ and $\vdash \{P \wedge \neg S\}\ C_2\ \{Q\}$

By the conditional rule, $\vdash \{P\}\ \textbf{if } S \textbf{ then } C_1 \textbf{ else } C_2\ \{Q\}$

VCs for sequences

Since we have restricted the domain of VC to be properly annotated specifications, we can assume that for any sequence $C_1; C_2$

- it has been annotated with an intermediate assertion, or
- C_2 is an assignment.

We define VC for each of these two cases:

$$VC(P, C_1; \{R\}\ C_2, Q) \stackrel{def}{=} VC(P, C_1, R) \cup VC(R, C_2, Q)$$
$$VC(P, C; V := E, Q) \stackrel{def}{=} VC(P, C, Q[E/V])$$

VCs for sequences

Example

$$\begin{aligned}
& VC(X = x \wedge Y = y,\ R := X; X := Y; Y := R,\ X = y \wedge Y = x) \\
= \ & VC(X = x \wedge Y = y,\ R := X; X := Y,\ (X = y \wedge Y = x)[R/Y]) \\
= \ & VC(X = x \wedge Y = y,\ R := X; X := Y,\ X = y \wedge R = x) \\
= \ & VC(X = x \wedge Y = y,\ R := X,\ (X = y \wedge R = x)[Y/X]) \\
= \ & VC(X = x \wedge Y = y,\ R := X,\ Y = y \wedge R = x) \\
= \ & \{X = x \wedge Y = y \Rightarrow (Y = y \wedge R = x)[X/R]\} \\
= \ & \{X = x \wedge Y = y \Rightarrow (Y = y \wedge X = x)\}
\end{aligned}$$

VCs for sequences

To justify the VCs, we have to prove that

$$\psi(C_1) \wedge \psi(C_2) \Rightarrow \psi(C_1; \{R\}\ C_2), \qquad \text{and}$$
$$\psi(C) \Rightarrow \psi(C; V := E)$$

where $\psi(C) \stackrel{def}{=} \forall P, Q. (\forall \phi \in VC(P, C, Q). \vdash \phi) \Rightarrow (\vdash \{P\}\ C\ \{Q\})$

These proofs are left as exercises, and you are strongly encouraged to try to prove one of them yourselves!

VCs for loops

A properly annotated loop has the form

$$\textbf{while } S \textbf{ do } \{R\}\ C$$

We use the annotation R as the invariant, and generate the following VCs:

$$VC(P, \textbf{while } B \textbf{ do } \{R\}\ C, Q) \overset{\mathit{def}}{=} \{P \Rightarrow R, R \wedge \neg B \Rightarrow Q\} \cup VC(R \wedge B, C, R)$$

VCs for loops

To justify the VCs for loops, we have to prove that

$$\psi(C) \Rightarrow \psi(\textbf{while } B \textbf{ do } \{R\}\ C)$$

where $\psi(C) \overset{\mathit{def}}{=} \forall P, Q. (\forall \phi \in VC(P, C, Q). \vdash \phi) \Rightarrow (\vdash \{P\}\ C\ \{Q\})$

Assume $\forall \phi \in VC(P, C, Q). \vdash \phi$.

Then $\vdash P \Rightarrow R$, $\vdash R \wedge \neg B \Rightarrow Q$ and $\forall \phi \in VC(R \wedge B, C, R). \vdash \phi$.
Hence, by the induction hypothesis, $\vdash \{R \wedge B\}\ C\ \{R\}$.

It follows by the backwards-reasoning rule for loops that

$$\vdash \{P\}\ \textbf{while } B \textbf{ do } C\ \{Q\}$$

Summary of VCs

We have outlined the design of a semi-automated program verifier based on Hoare logic.

It takes annotated specifications, and generates a set of first-order logic statements that, if provable, ensure the specification is provable.

Intelligence is required to provide the annotations and help the automated theorem prover.

The soundness of the verifier is justified using a simple inductive argument, and uses many of the derived rules for backwards reasoning from the last lecture.

Other uses for Hoare triples

So far we have assumed P, C and Q were given and focused on proving $\vdash \{P\}\ C\ \{Q\}$.

If we are given P and C, can we infer a Q?
Is there a best such Q? ('strongest postcondition')

If we are given C and Q, can we infer a P?
Is there a best such P? ('weakest precondition')

If we are given P and Q, can we infer a C?
('program refinement' or 'program synthesis')

Weakest preconditions

If C is a command and Q is an assertion, then informally $wlp(C, Q)$ is the weakest assertion P such that $\{P\}\ C\ \{Q\}$ holds.

If P and Q are assertions, then P is 'weaker' than Q if $Q \Rightarrow P$. Thus, we are looking for a function wlp such that $\{P\}\ C\ \{Q\} \Leftrightarrow P \Rightarrow wlp(C, Q)$.

Dijkstra gives rules for computing weakest liberal preconditions for deterministic loop-free code:

$$
\begin{aligned}
wlp(V := E, Q) &= Q[E/V] \\
wlp(C1; C2, Q) &= wlp(C1, wp(C2, Q)) \\
wlp(\textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2, Q) &= (B \Rightarrow wlp(C_1, Q)) \wedge \\
&\quad\ (\neg B \Rightarrow wlp(C_2, Q))
\end{aligned}
$$

Weakest preconditions

While the following property holds for loops

$$
\begin{aligned}
&wlp(\textbf{while } B \textbf{ do } C, Q) \Leftrightarrow \\
&\quad \textbf{if } B \textbf{ then } wlp(C, wlp(\textbf{while } B \textbf{ do } C, Q)) \textbf{ else } Q
\end{aligned}
$$

it does not define $wlp(\textbf{while } B \textbf{ do } C, Q)$ as a finite formula.

In general, one cannot compute a finite formula for $wlp(\textbf{while } B \textbf{ do } C, Q)$.

If C is loop-free, then we can take the VC for $\{P\}\ C\ \{Q\}$ to be $P \Rightarrow wlp(C, Q)$, without requiring C to be annotated.

Program refinement

We have focused on proving programs meet specifications.

An alternative is to construct a program that is correct by construction, by refining a specification into a program.

Rigorous development methods such as the B-Method, SPARK and the Vienna Development Method (VDM) are based on this idea.

For more: "Programming From Specifications" by Carroll Morgan.

Conclusion

Several practical tools for program verification are based on the idea of generating VCs from annotated programs:

- Gypsy (1970s);
- SPARK (current tool for Ada, used in aerospace & defence).

Weakest liberal preconditions can be used to reduce the number of annotations required in loop-free code.

Hoare Logic and Model Checking

Jean Pichon-Pharabod
University of Cambridge

CST Part II – 2017/18

Pointers

Pointers and state

So far, we have been reasoning about a language without pointers, where all values were numbers.

In this lecture, we will extend the WHILE language with pointers, and introduce an extension of Hoare logic, called separation logic, to simplify reasoning about pointers.

Pointers and state

$$
\begin{array}{rcll}
E & ::= & N \mid \mathbf{null} \mid V \mid E_1 + E_2 & \textit{arithmetic expressions} \\
 & \mid & E_1 - E_2 \mid E_1 \times E_2 \mid \cdots & \\
 & & & \\
B & ::= & T \mid F \mid E_1 = E_2 & \textit{boolean expressions} \\
 & \mid & E_1 \leq E_2 \mid E_1 \geq E_2 \mid \cdots & \\
 & & & \\
C & ::= & \mathbf{skip} \mid C_1; C_2 \mid V := E & \textit{commands} \\
 & \mid & \mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2 & \\
 & \mid & \mathbf{while}\ B\ \mathbf{do}\ C & \\
 & \mid & V := [E] \mid [E_1] := E_2 & \\
 & \mid & V := \mathbf{cons}(E_1, ..., E_n) \mid \mathbf{dispose}(E) &
\end{array}
$$

Pointers and state

Commands are now evaluated also with respect to a **heap** h that stores the current value of allocated locations.

Reading, writing, and disposing of pointers fails if the given location is not currently allocated.

Fetch assignment command: $V := [E]$

- evaluates E to a location ℓ, and assigns the current value of ℓ to V; faults if ℓ is not currently allocated.

Pointers and state

Heap assignment command: $[E_1] := E_2$

- evaluates E_1 to a location ℓ and E_2 to a value v, and updates the heap to map ℓ to v; faults if ℓ is not currently allocated.

Pointer disposal command: $\mathbf{dispose}(E)$

- evaluates E to a location ℓ, and deallocates location ℓ from the heap; faults if ℓ is not currently allocated.

Pointers and state

Allocation assignment command: $V := \mathbf{cons}(E_1, \ldots, E_n)$

- chooses n **consecutive** unallocated locations, say $\ell_1, \ldots, \ell_n$, evaluates $E_1, \ldots, E_n$ to values $v_1, \ldots, v_n$, updates the heap to map ℓ_i to v_i for each i, and assigns ℓ_1 to V.

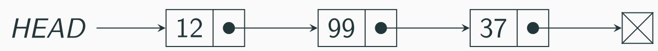
Allocation never fails.

The language supports pointer arithmetic: e.g.,

$$X := \mathbf{cons}(0, 1); Y := [X + 1]$$

Pointers and state

In this extended language, we can work with proper data structures, like the following singly-linked list:



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

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

Operational semantics

Pointers and state

For the WHILE language, we modelled the state as a function assigning values (numbers) to all variables:

$$s \in \mathit{Store} \stackrel{\mathit{def}}{=} \mathit{Var} \to \mathbb{Z}$$

To model pointers, we will split the state into a **stack** and a **heap**:

- a stack maps program variables to values, and
- a heap maps locations to values

$$\mathit{State} \stackrel{\mathit{def}}{=} \mathit{Store} \times \mathit{Heap}$$

Pointers and state

Values now includes both numbers and locations

$$\mathit{Val} \stackrel{\mathit{def}}{=} \mathbb{Z} + \mathit{Loc}$$

Locations are modelled as natural numbers

$$\ell \in \mathit{Loc} \stackrel{\mathit{def}}{=} \mathbb{N}$$

To model allocation, we model the heap as a **finite** function, that is, a partial function with a finite domain:

$$\mathit{Store} \stackrel{\mathit{def}}{=} \mathit{Var} \to \mathit{Val} \qquad \mathit{Heap} \stackrel{\mathit{def}}{=} \mathit{Loc} \stackrel{\mathit{fin}}{\rightharpoonup} \mathit{Val}$$

Pointers and state

WHILE_p programs can fail in several ways:

- dereferencing an invalid pointer;
- invalid pointer arithmetic.

To model failure, we introduce a distinguished failure value $\lightning$, and adapt the semantics:

$$\begin{aligned}
\mathcal{E}[\![-]\!] &: \mathit{Exp} \times \mathit{Store} \to \{\lightning\} + \mathit{Val}\\
\mathcal{B}[\![-]\!] &: \mathit{BExp} \times \mathit{Store} \to \{\lightning\} + \mathbb{B}\\
\Downarrow &: \mathcal{P}(\mathit{Cmd} \times \mathit{State} \times (\{\lightning\} \cup \mathit{State}))
\end{aligned}$$

Pointer dereference

$$\frac{\mathcal{E}[\![E]\!](s) = \ell \qquad \ell \in dom(h)}{\langle V := [E], (s, h)\rangle \Downarrow (s[V \mapsto h(l)], h)}$$

$$\frac{\mathcal{E}[\![E]\!](s) = \ell \qquad \ell \notin dom(h)}{\langle V := [E], (s, h)\rangle \Downarrow \lightning}$$

Pointer assignment

$$\frac{\begin{array}{c}\mathcal{E}[\![E_1]\!](s) = \ell \qquad \mathcal{E}[\![E_2]\!](s) = v \\ \ell \in dom(h) \qquad v \neq \lightning\end{array}}{\langle E_1 := [E_1], (s, h)\rangle \Downarrow (s, h[\ell \mapsto v])}$$

$$\frac{\mathcal{E}[\![E_1]\!](s) = \ell \qquad \ell \notin dom(h)}{\langle E_1 := [E_2], (s, h)\rangle \Downarrow \lightning} \qquad \frac{\mathcal{E}[\![E_2]\!](s) = \lightning}{\langle E_1 := [E_2], (s, h)\rangle \Downarrow \lightning}$$

Reasoning about pointers

Reasoning about pointers

In standard Hoare logic, we can syntactically approximate the set of program variables that might be affected by a command C:

$$\begin{aligned} mod(\textbf{skip}) &= \emptyset \\ mod(X := E) &= \{X\} \\ mod(C_1; C_2) &= mod(C_1) \cup mod(C_2) \\ mod(\textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2) &= mod(C_1) \cup mod(C_2) \\ mod(\textbf{while } B \textbf{ do } C) &= mod(C) \end{aligned}$$

The rule of constancy

The rule of constancy expresses that assertions that do not refer to variables modified by a command are automatically preserved during its execution:

$$\frac{\vdash \{P\}\ C\ \{Q\} \qquad mod(C) \cap FV(R) = \emptyset}{\vdash \{P \wedge R\}\ C\ \{Q \wedge R\}}$$

This rule is derivable in standard Hoare logic.

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

Reasoning about pointers

Imagine we extended Hoare logic with a new assertion, $E_1 \hookrightarrow E_2$, for asserting that location E_1 currently contains the value E_2, and extend the proof system with the following rule:

$$\frac{}{\vdash \{\top\}\ [E_1] := E_2\ \{E_1 \hookrightarrow E_2\}}$$

Then we lose the rule of constancy:

$$\frac{\vdash \{\top\}\ [X] := 1\ \{X \hookrightarrow 1\}}{\vdash \{\top \wedge Y \hookrightarrow 0\}\ [X] := 1\ \{X \hookrightarrow 1 \wedge Y \hookrightarrow 0\}}$$

(the post-condition is false if X and Y refer to the same location).

Reasoning about pointers

In the presence of pointers, syntactically distinct variables can refer to the same location. Updates made through one variable can thus influence the state referenced by other variables.

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

$$\frac{}{\vdash \{E_1 \neq E_3 \wedge E_3 \hookrightarrow E_4\}\ [E_1] := E_2\ \{E_1 \hookrightarrow E_2 \wedge E_3 \hookrightarrow E_4\}}$$

Separation logic

Separation logic

Separation logic (SL) is an extension of Hoare logic (HL) that simplifies reasoning about mutable state by using new connectives to control aliasing.

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.

Separation logic

Separation logic introduces two new concepts for reasoning about mutable state:

- **ownership**: Separation logic assertions do not just describe properties of the current state, they also assert ownership of part of the heap.
- **separation**: Separation logic introduces a new connective, written $P * Q$, for asserting that the parts of the heap owned by P and Q are **disjoint**.

This makes it easy to describe data structures without sharing.

Separation logic

Separation logic introduces a new assertion, written $E_1 \mapsto E_2$, for reasoning about individual heap cells.

The points-to assertion $E_1 \mapsto E_2$

- asserts that the current value of heap location E_1 is E_2, and
- asserts ownership of heap location E_1.

Meaning of separation logic assertions

The semantics of a separation logic assertion, written $[\![P]\!](s)$, is a set of heaps that satisfy the assertion P.

The intended meaning is that if $h \in [\![P]\!](s)$, then P asserts ownership of any locations in $dom(h)$.

The heaps $h \in [\![P]\!](s)$ are thus referred to as **partial heaps**, since they only contain the locations owned by P.

The empty heap assertion only holds for the empty heap:

$$[\![emp]\!](s) \stackrel{def}{=} \{[]\}$$

Meaning of separation logic assertions

The points-to assertion $E_1 \mapsto E_2$ asserts ownership of the location referenced by E_1, and that this location currently contains E_2:

$$
\begin{aligned}
[\![E_1 \mapsto E_2]\!](s) \stackrel{def}{=} \{h \mid\ & dom(h) = \{\mathcal{E}[\![E_1]\!](s)\} \\
& \wedge h(\mathcal{E}[\![E_1]\!](s)) = \mathcal{E}[\![E_2]\!](s)\}
\end{aligned}
$$

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

$$
\begin{aligned}
[\![P * Q]\!](s) \stackrel{def}{=} \{h \mid\ & \exists h_1, h_2.\ h = h_1 \uplus h_2 \\
& \wedge h_1 \in [\![P]\!](s) \wedge h_2 \in [\![Q]\!](s)\}
\end{aligned}
$$

We use $h_1 \uplus h_2$, which is equal to $h_1 \cup h_2$ when it is defined, but is only defined when $dom(h_1) \cap dom(h_2) = \emptyset$.

Examples of separation logic assertions

1. $X \mapsto E_1 * Y \mapsto E_2$

   This assertion is unsatisfiable in a state where X and Y refer to the same location, since $X \mapsto E_1$ and $Y \mapsto E_2$ would both assert ownership of the same location.

   The following heap satisfies the assertion:

   $$X \longrightarrow \boxed{E_1} \qquad \boxed{E_2} \longleftarrow Y$$

2. $X \mapsto E * X \mapsto E$

   This assertion is not satisfiable.

Meaning of separation logic assertions

The first-order primitives are interpreted much like for Hoare logic:

$$
\begin{aligned}
[\![\bot]\!](s) &\stackrel{def}{=} \emptyset \\
[\![\top]\!](s) &\stackrel{def}{=} Heap \\
[\![P \wedge Q]\!](s) &\stackrel{def}{=} [\![P]\!](s) \cap [\![Q]\!](s) \\
[\![P \vee Q]\!](s) &\stackrel{def}{=} [\![P]\!](s) \cup [\![Q]\!](s) \\
[\![P \Rightarrow Q]\!](s) &\stackrel{def}{=} \{h \mid h \in [\![P]\!](s) \Rightarrow h \in [\![Q]\!](s)\} \\
&\vdots
\end{aligned}
$$

Examples of separation logic assertions

3. $X \mapsto E_1 \wedge Y \mapsto E_2$

   This asserts that either X and Y alias each other and $E_1 = E_2$:

   $$X \longrightarrow \boxed{E_1} \longleftarrow Y$$

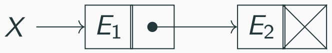
   or they refer to distinct locations:

   $$X \longrightarrow \boxed{E_1} \qquad \boxed{E_2} \longleftarrow Y$$

Examples of separation logic assertions

4. $X \mapsto Y * Y \mapsto X$

5. $X \mapsto E_1, Y * Y \mapsto E_2, \textbf{null}$



Here, $X \mapsto E_1, ..., E_n$ is shorthand for

$$X \mapsto E_1 * (X+1) \mapsto E_2 * \cdots * (X+n-1) \mapsto E_n$$

Summary: separation logic assertions

Separation logic assertions **describe** properties of the current state and assert **ownership** of parts of the current heap.

Separation logic controls aliasing of pointers by asserting that assertions own **disjoint** heap parts.

Separation logic triples

Separation logic triples

Separation logic extends the assertion language, but uses the same Hoare triples to reason about the behaviour of programs

$$\vdash \{P\}\ C\ \{Q\} \qquad\qquad \vdash [P]\ C\ [Q]$$

but with a different meaning.

Our SL triples extend the meaning of our HL triples in two ways:

- they ensure that our WHILE_p programs do not fail;
- they require that we respect the ownership discipline associated with assertions.

Separation logic triples

Separation logic triples require that we assert ownership in the precondition of any heap cells modified.

For instance, the following triple asserts ownership of the location denoted by X, and stores the value 2 at this location:

$$\vdash \{X \mapsto 1\}\ [X] := 2\ \{X \mapsto 2\}$$

However, the following triple is not valid, because it updates a location that it may not be the owner of:

$$\nvdash \{Y \mapsto 1\}\ [X] := 2\ \{Y \mapsto 1\}$$

Framing

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

The idea is to require that all triples must preserve any assertions disjoint from the precondition. This is captured by the frame rule:

$$\frac{\vdash \{P\}\ C\ \{Q\} \qquad mod(C) \cap FV(R) = \emptyset}{\vdash \{P * R\}\ C\ \{Q * R\}}$$

The assertion R is called the frame.

Framing

How does preserving all frames force triples to assert ownership of heap cells they modify?

Imagine that the following triple did hold and preserved all frames:

$$\{Y \mapsto 1\}\ [X] := 2\ \{Y \mapsto 1\}$$

In particular, it would preserve the frame $X \mapsto 1$:

$$\{Y \mapsto 1 * X \mapsto 1\}\ [X] := 2\ \{Y \mapsto 1 * X \mapsto 1\}$$

This triple definitely does not hold, since the location referenced by X contains 2 in the terminal state.

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 does assert ownership of X:

$$\{X \mapsto 1\}\ [X] := 2\ \{X \mapsto 2\}$$

If we frame on $X \mapsto 1$, then we get the following triple which holds vacuously since no initial states satisfies $X \mapsto 1 * X \mapsto 1$:

$$\{X \mapsto 1 * X \mapsto 1\}\ [X] := 2\ \{X \mapsto 2 * X \mapsto 1\}$$

Meaning 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 C terminates in a terminal state when executed from an initial heap $h_1 \uplus h_F$ where h_1 satisfies P then the terminal state has the form $h_1' \uplus h_F$ where h_1' satisfies Q.

This bakes in the requirement that triples must satisfy framing, by requiring that they preserve all disjoint frames h_F.

Meaning of separation logic triples

Written formally, the meaning is:

$$
\begin{aligned}
&\models \{P\}\ C\ \{Q\} \stackrel{def}{=} \\
&\quad (\forall s, h.\ h \in [\![P]\!](s) \Rightarrow \neg(\langle C, (s, h)\rangle \Downarrow \lightning)) \wedge \\
&\quad (\forall s, s', h, h', h_F.\ dom(h) \cap dom(h_F) = \emptyset \wedge \\
&\qquad h \in [\![P]\!](s) \wedge \langle C, (s, h \uplus h_F)\rangle \Downarrow (s', h') \\
&\qquad\quad \Rightarrow \exists h_1'.\ h' = h_1' \uplus h_F \wedge h_1' \in [\![Q]\!](s'))
\end{aligned}
$$

Summary

Separation logic is an extension of Hoare logic with new primitives to simplify reasoning about pointers.

Separation logic extends Hoare logic with notions of **ownership** and **separation** to control aliasing and reason about shared mutable data structures.

Papers of historical interest:

- John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures.

Hoare Logic and Model Checking

Jean Pichon-Pharabod
University of Cambridge

CST Part II – 2017/18

Introduction

In the previous lecture, we saw the informal concepts that separation logic is based on.

This lecture will

- introduce a formal proof system for separation logic;
- present examples to illustrate the power of separation logic.

The lecture will be focused on partial correctness.

A proof system for separation logic

Separation logic

Separation logic inherits all the partial correctness rules from Hoare logic that you have already seen, and extends them with

- the frame rule;
- rules for each new heap primitive.

Some of the derived rules for plain Hoare logic no longer hold for separation logic (e.g., the rule of constancy).

The frame rule

The frame rule expresses that separation logic triples always preserve any resources disjoint from the precondition:

$$\frac{\vdash \{P\}\ C\ \{Q\} \qquad mod(C) \cap FV(R) = \emptyset}{\vdash \{P * R\}\ C\ \{Q * R\}}$$

The second hypothesis ensures that the frame R does not refer to any program variables modified by the command C.

The heap assignment rule

Separation logic triples must assert ownership of any heap cells modified by the command. The heap assignment rule thus asserts ownership of the heap location being assigned:

$$\vdash \{E_1 \mapsto \_ * E_2 = \_\}\ [E_1] := E_2\ \{E_1 \mapsto E_2\}$$

It also requires that evaluating E_2 does not fault.
($E \mapsto \_$ is short for $\exists v.\, E \mapsto v$, and $E = \_$ is short for $\exists v.\, E = v$)

Exercise: Why is $E_1 = \_$ not necessary in the precondition?

The heap dereference rule

Separation logic triples must ensure the command does not fault. The heap dereference rule thus asserts ownership of the given heap location to ensure the location is allocated in the heap.

$$\vdash \{E \mapsto v \land X = x\}\ X := [E]\ \{E[x/X] \mapsto v \land X = v\}$$

Here, the auxiliary variable x is used to refer to the initial value of X in the postcondition.

Separation logic

The assignment rule introduces a new points-to assertion for each newly allocated location:

$$\vdash \{X = x\}\ X := cons(E_1, ..., E_n)\ \{X \mapsto E_1[x/X], ..., E_n[x/X]\}$$

The deallocation rule destroys the points-to assertion for the location to not be available anymore:

$$\vdash \{E \mapsto \_\}\ \mathbf{dispose}(E)\ \{emp\}$$

Swap example

To illustrate these rules, consider the following code snippet:

$$C_{swap} \equiv A := [X]; B := [Y]; [X] := B; [Y] := A;$$

We want to show that it swaps the values in the locations referenced by X and Y, when X and Y do not alias:

$$\{X \mapsto v_1 * Y \mapsto v_2\}\ C_{swap}\ \{X \mapsto v_2 * Y \mapsto v_1\}$$

Swap example

Below is a proof-outline of the main steps:

$$\{X \mapsto v_1 * Y \mapsto v_2\}$$
$$A := [X];$$
$$\{X \mapsto v_1 * Y \mapsto v_2 \wedge A = v_1\}$$
$$B := [Y];$$
$$\{X \mapsto v_1 * Y \mapsto v_2 \wedge A = v_1 \wedge B = v_2\}$$
$$[X] := B;$$
$$\{X \mapsto B * Y \mapsto v_2 \wedge A = v_1 \wedge B = v_2\}$$
$$[Y] := A;$$
$$\{X \mapsto B * Y \mapsto A \wedge A = v_1 \wedge B = v_2\}$$
$$\{X \mapsto v_2 * Y \mapsto v_1\}$$

Swap example

To prove this first triple, we use the heap dereference rule to derive:

$$\{X \mapsto v_1 \wedge A = a\}\ A := [X]\ \{X[a/A] \mapsto v_1 \wedge A = v_1\}$$

Then we existentially quantify the auxiliary variable a:

$$\{\exists a.\, X \mapsto v_1 \wedge A = a\}\ A := [X]\ \{\exists a.\, X[a/A] \mapsto v_1 \wedge A = v_1\}$$

Applying the rule of consequence, we obtain:

$$\{X \mapsto v_1\}\ A := [X]\ \{X \mapsto v_1 \wedge A = v_1\}$$

Since $A := [X]$ does not modify Y, we can frame on $Y \mapsto v_2$:

$$\{X \mapsto v_1 * Y \mapsto v_2\}\ A := [X]\ \{(X \mapsto v_1 \wedge A = v_1) * Y \mapsto v_2\}$$

Lastly, by the rule of consequence, we obtain:

$$\{X \mapsto v_1 * Y \mapsto v_2\}\ A := [X]\ \{X \mapsto v_1 * Y \mapsto v_2 \wedge A = v_1\}$$

Swap example

For the last application of consequence, we need to show that:

$$\vdash (X \mapsto v_1 \wedge A = v_1) * Y \mapsto v_2 \Rightarrow X \mapsto v_1 * Y \mapsto v_2 \wedge A = v_1$$

To prove this, we need proof rules for the new separation logic primitives.

Separation logic assertions

Separation conjunction is commutative and associative operator with *emp* as a neutral element:

$$\vdash P * Q \Leftrightarrow Q * P$$
$$\vdash (P * Q) * R \Leftrightarrow P * (Q * R)$$
$$\vdash P * emp \Leftrightarrow P$$

Separation conjunction is monotone with respect to implication:

$$\frac{\vdash P_1 \Rightarrow Q_1 \qquad \vdash P_2 \Rightarrow Q_2}{\vdash P_1 * P_2 \Rightarrow Q_1 * Q_2}$$

Separation logic assertions

Separating conjunction distributes over disjunction and semi-distributes over conjunction:

$$\vdash (P \vee Q) * R \Leftrightarrow (P * R) \vee (Q * R)$$
$$\vdash (P \wedge Q) * R \Rightarrow (P * R) \wedge (Q * R)$$

Taking $R \equiv X \mapsto 1 \vee Y \mapsto 1$, $P \equiv X \mapsto 1$ and $Q \equiv X \mapsto 1$ yields a counterexample to distributivity over conjunction in the other direction:

$$\not\models (P * R) \wedge (Q * R) \Rightarrow (P \wedge Q) * R$$

Separation logic assertions

An assertion is **pure** if it does not contain *emp*, $\mapsto$, or $\hookrightarrow$.

Separating conjunction and conjunction collapse for pure assertions:

$\vdash P \wedge Q \Rightarrow P * Q$	when P or Q is pure
$\vdash P * Q \Rightarrow P \wedge Q$	when P and Q are pure
$\vdash (P \wedge Q) * R \Leftrightarrow P \wedge (Q * R)$	when P is pure

Verifying abstract data types

Verifying ADTs

Separation logic is very well-suited for specifying and reasoning about data structures typically found in standard libraries such as lists, queues, stacks, etc.

To illustrate, we will specify and verify a library for working with linked lists in separation logic.

A linked list library

First, we need to define a memory representation for our linked lists.

We will use a singly-linked list, starting from some designated *head* variable that refers to the first element of the list and terminating with a *null* pointer.

For instance, we will represent a list containing the values $12, 99$, and 37 as follows:

Representation predicates

To formalise the memory representation, separation logic uses **representation predicates** that relate an abstract description of the state of the data structure with its concrete memory representations.

For our example, we want a predicate $list(head, \alpha)$ that relates a mathematical list, α, with its memory representation.

To define such a predicate formally, we need to extend the assertion logic to reason about mathematical lists, support for predicates and inductive definitions. We will elide these details.

Representation predicates

We are going to define the $list(head, \alpha)$ predicate by induction on the list α. We need to consider two cases: the empty list and an element x appended to a list β.

An empty list is represented as a *null* pointer:

$$list(head, []) \stackrel{def}{=} head = null$$

The list $x :: \beta$ is represented by a reference to two consecutive heap cells that contain the value x and a representation of the rest of the list, respectively:

$$list(head, x :: \beta) \stackrel{def}{=} \exists y.\, head \mapsto x * (head + 1) \mapsto y * list(y, \beta)$$

Representation predicates

The representation predicate allows us to specify the behaviour of the list operations by their effect on the abstract state of the list

Imagine C_{push} is an implementation of an push operation that pushes the value stored in variable X to the front of the list referenced by variable $HEAD$, and stores a reference to the new list in $HEAD$.

We can specify this operation in terms of its behaviour on the abstract state of the list as follows:

$$\{list(HEAD, \alpha) \wedge X = x\}\ C_{add}\ \{list(HEAD, x :: \alpha)\}$$

Representation predicates

We can specify all the operations of the library in a similar manner:

$$\{emp\} \quad C_{new} \quad \{list(HEAD, [])\}$$

$$\{list(HEAD, \alpha) \wedge X = x\} \quad C_{push} \quad \{list(HEAD, x :: \alpha)\}$$

$$\begin{array}{rll} \{list(HEAD, x :: \alpha)\} & C_{pop} & \{list(HEAD, \alpha) \wedge RET = x\} \\ \{list(HEAD, [])\} & C_{pop} & \{list(HEAD, []) \wedge RET = null\} \end{array}$$

$$\{list(HEAD, \alpha)\} \quad C_{delete} \quad \{emp\}$$

Implementation of *push*

The *push* operation stores the *HEAD* pointer pointer into a temporary variable Y before allocating two consecutive heap cells for the new list element and updating *HEAP*:

$$C_{push} \equiv Y := HEAD; HEAD := \mathbf{cons}(X, Y)$$

We wish to prove it satisfies the following specification:

$$\{list(HEAD, \alpha) \wedge X = x\} \; C_{push} \; \{list(HEAD, x :: \alpha)\}$$

Proof outline for *push*

Here is a proof outline for the *push* operation:

$$\begin{array}{l} \{list(HEAD, \alpha) \wedge X = x\} \\ Y := HEAD \\ \{list(Y, \alpha) \wedge X = x\} \\ HEAD := \mathbf{cons}(X, Y) \\ \{list(Y, \alpha) * HEAD \mapsto X, Y \wedge X = x\} \\ \{list(HEAD, X :: \alpha) \wedge X = x\} \\ \{list(HEAD, x :: \alpha)\} \end{array}$$

For the **cons** step, we frame off $list(Y, \alpha) \wedge X = x$.

Implementation of *delete*

The *delete* operation iterates down over the list, deallocating nodes until it reaches the end of the list.

$$\begin{array}{l} C_{delete} \equiv X := HEAD; \\ \qquad\qquad \mathbf{while}\ X \neq NULL\ \mathbf{do} \\ \qquad\qquad\quad Y := [X + 1]; \mathbf{dispose}(X); \mathbf{dispose}(X + 1); X := Y \end{array}$$

To prove that *delete* satisfies its intended specification,

$$\{list(HEAD, \alpha)\} \; C_{delete} \; \{emp\}$$

we need a suitable invariant: that we own the rest of the list.

Proof outline for *delete*

$\{list(HEAD, \alpha)\}$

$X := HEAD;$

$\{list(X, \alpha)\}$

$\{\exists \alpha.\ list(X, \alpha)\}$

while $X \neq NULL$ **do**

$\quad \{\exists \alpha.\ list(X, \alpha) \wedge X \neq NULL\}$

$\quad (Y := [X + 1];$ **dispose**$(X);$ **dispose**$(X + 1); X := Y)$

$\quad \{\exists \alpha.\ list(X, \alpha)\}$

$\{list(X, \alpha) \wedge \neg(X \neq NULL)\}$

$\{emp\}$

Proof outline for the loop body of *delete*

To verify the loop body, we need a lemma to unfold the list representation predicate in the non-null case:

$\{\exists \alpha.\ list(X, \alpha) \wedge X \neq NULL\}$

$\{\exists v, t, \alpha.\ X \mapsto v, t * list(t, \alpha)\}$

$Y := [X + 1];$

$\{\exists v, \alpha.\ X \mapsto v, Y * list(Y, \alpha)\}$

dispose$(X);$ **dispose**$(X + 1);$

$\{\exists \alpha.\ list(Y, \alpha)\}$

$X := Y$

$\{\exists \alpha.\ list(X, \alpha)\}$

Concurrency (not examinable)

Concurrency

Imagine extending our WHILE_p language with a parallel composition construct, $C_1 || C_2$, which executes the two statements C_1 and C_2 in parallel.

The statement $C_1 || C_2$ reduces by interleaving execution steps of C_1 and C_2, until both have terminated, before continuing program execution.

For instance, $(X := 0 || X := 1); print(X)$ will randomly print 0 or 1.

Concurrency

Adding parallelism complicates reasoning by introducing the possibility of concurrent interference on shared state.

While separation logic does extend to reason about general concurrent interference, we will focus on two common idioms of concurrent programming with limited forms of interference:

- **disjoint concurrency**;
- **well-synchronised shared state**.

Disjoint concurrency

Disjoint concurrency refers to multiple commands potentially executing in parallel, but all working on **disjoint** state.

Parallel implementations of divide-and-conquer algorithms can often be expressed using disjoint concurrency.

For instance, in a parallel merge sort, the recursive calls to merge sort operate on disjoint parts of the underlying array.

Disjoint concurrency

The proof rule for disjoint concurrency requires us to split our resources into two disjoint parts, P_1 and P_2, and give each parallel command ownership of one of them:

$$\frac{\vdash \{P_1\}\ C_1\ \{Q_1\} \qquad \vdash \{P_2\}\ C_2\ \{Q_2\} \\ mod(C_1) \cap FV(P_2, Q_2) = mod(C_2) \cap FV(P_1, Q_1) = \emptyset}{\vdash \{P_1 * P_2\}\ C_1 || C_2\ \{Q_1 * Q_2\}}$$

The third hypothesis ensures C_1 does not modify any program variables used in the specification of C_2, and vice versa.

Disjoint concurrency example

Here is a simple example to illustrate two parallel increment operations that operate on disjoint parts of the heap:

$$\begin{array}{c}
\{X \mapsto 3 * Y \mapsto 4\} \\
\begin{array}{lcl}
\{X \mapsto 3\} & & \{Y \mapsto 4\} \\
A := [X]; [X] := A + 1 & || & B := [Y]; [Y] := B + 1 \\
\{X \mapsto 4\} & & \{Y \mapsto 5\}
\end{array} \\
\{X \mapsto 4 * Y \mapsto 5\}
\end{array}$$

Well-synchronised shared state

Well-synchronised shared state refers to the common concurrency idiom of using locks to ensure exclusive access to state shared between multiple threads.

To reason about locking, Concurrent separation logic extends separation logic with **lock invariants** that describe the resources protected by locks.

When acquiring a lock, the acquiring thread takes ownership of the lock invariant and when releasing the lock, must give back ownership of the lock invariant.

Well-synchronised shared state

To illustrate, consider a simplified setting with a single global lock.

We write $I \vdash \{P\}\ C\ \{Q\}$ to indicate that we can derive the given triple assuming the lock invariant is I.

$$I \vdash \{emp\}\ \textbf{acquire}\ \{I * locked\}$$
$$I \vdash \{I * locked\}\ \textbf{release}\ \{emp\}$$

where I is not allowed to refer to any program variables.

The *locked* resource ensures the lock can only be released by the thread that currently has the lock.

Well-synchronised shared state example

To illustrate, consider a program with two threads that both access a number stored in shared heap cell at location X in parallel.

Thread A increments X by 1 twice, and thread B increments X by 2. The threads use a lock to ensure their accesses are well-synchronised.

Assuming X initially contains an even number, we wish to prove that X is still even after the two parallel threads have terminated.

Well-synchronised shared state example

First, we need to define a lock invariant.

The lock invariant needs to own the shared heap cell at location x and should express that it always contains an even number:

$$I \stackrel{def}{=} \exists v.\, x \mapsto v * even(v)$$

Well-synchronised shared state example

Assuming the lock invariant I is $\exists v.\, x \mapsto v * even(v)$, we have:

$$\{X = x \wedge emp\}$$

$$\begin{array}{lcl}
\{X = x \wedge emp\} & & \{X = x \wedge emp\} \\
\textbf{acquire}; & & \textbf{acquire}; \\
\{X = x \wedge I * locked\} & & \{X = x \wedge I * locked\} \\
\begin{array}{l} A := [X]; [X] := A + 1; \\ B := [X]; [X] := B + 1; \end{array} & || & C := [X]; [X] := C + 2; \\
\{X = x \wedge I * locked\} & & \{X = x \wedge I * locked\} \\
\textbf{release}; & & \textbf{release}; \\
\{X = x \wedge emp\} & & \{X = x \wedge emp\}
\end{array}$$

$$\{X = x \wedge emp\}$$

Summary

Abstract data types are specified using representation predicates which relate an abstract model of the state of the data structure with a concrete memory representation.

Separation logic supports reasoning about well-synchronised concurrent programs, using lock invariants to guard access to shared state.

Papers of historical interest:

- Peter O'Hearn. Resources, Concurrency and Local Reasoning.