# 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, Joe Isaacs, Ian Orton, and Domagoj Stolfa for reporting mistakes.



## Motivation

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

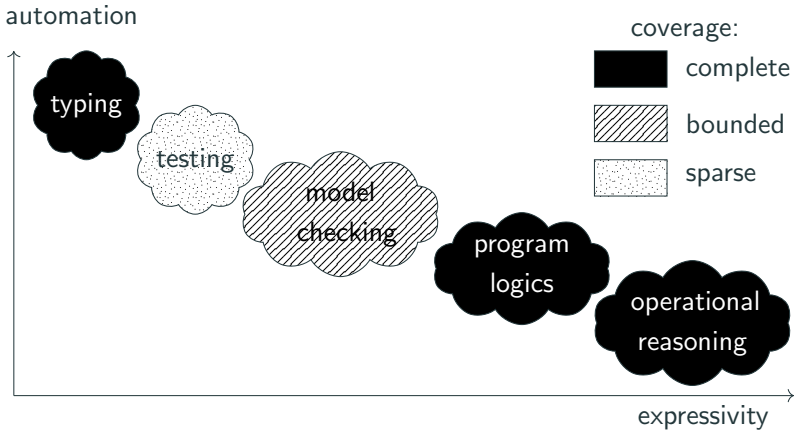
- we fail to write programs that meet their specification;
- we fail to write specifications that meet our expectations.

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.



## Partial correctness triples

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

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

is a logical statement about a command $C$,
where $P$ 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.



## Components of a Hoare logic

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

- the programming language that we want to reason about:
  its syntax and dynamic (e.g. operational) semantics;
- an assertion language for defining state predicates:
  its syntax and an interpretation;
- an interpretation of Hoare triples;
- a (sound) syntactic 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

## Commands of 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 an integer, and $B$ is a boolean expression, which evaluates to a boolean.

States are mappings from variables to integers.



## Expressions of the WHILE language

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

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

Note that expressions do not have side effects.



# Assertions and specifications

## The assertion language

Assertions (also "state predicates") $P, Q, \ldots$ include boolean expressions (which can contain program variables), combined using the usual logical operators: $\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\forall$, $\exists$, $\ldots$

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.

## Informal semantics of 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.



## Informal semantics of total correctness

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

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 $Q$.



## 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, partial correctness, and termination

Informally: total correctness = partial correctness + termination.

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 $X$:

**while** $X > 1$ **do**
  **if** $ODD(X)$ **then** $X := 3 \times X + 1$ **else** $X := X \ DIV \ 2$

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



## Examples of specifications

## Corner cases of partial correctness triples

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

- 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.



## Corner cases of total correctness triples

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

- 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.

## The need for auxiliary variables

How can we specify that a program $C$ computes the maximum of two variables $X$ and $Y$, and stores the result in a variable $Z$?

Is this a good specification for $C$?

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

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 $Y$.



# Formal proof system for Hoare logic

## Auxiliary variables

In Hoare logic, we use **auxiliary variables** (also "ghost variables", or "logical variables"), which are not allowed not occur in the program, to refer to the initial values of variables in postconditions.

Notation: program variables are uppercase, and auxiliary variables are lowercase. $v$ ranges over auxiliary variables, and concrete values are $x, y, \ldots$.

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



## 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):

$$\dfrac{\dfrac{\dfrac{}{\vdash S_1} \qquad \dfrac{}{\vdash S_2}}{\vdash S_3} \qquad \dfrac{}{\vdash S_4}}{\vdash S}$$

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



## Formal proof system for Hoare logic

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

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

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

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

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



## The skip rule

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

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,

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

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



## 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 rule:

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

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 $\vdash \{X + 1 = 2\}\ X := X + 1\ \{X = 2\}$, we can deduce $\vdash \{X = 1\}\ X := X + 1\ \{X = 2\}$.



## 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:

- $\vdash \{X = 1\}\ X := X + 1\ \{X = 2\}$
- $\vdash \{X = 2\}\ X := X \times 2\ \{X = 4\}$

we may deduce that $\vdash \{X = 1\}\ X := X + 1; X := X \times 2\ \{X = 4\}$.

## 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.



## (Redundant) Conjunction and disjunction rules

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

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

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.

In the next lecture, we will formalise the intuitions we gave today, and prove soundness of Hoare logic.

# Hoare logic

**Lecture 2: Formalising the semantics of Hoare logic**

**Jean Pichon-Pharabod**
University of Cambridge

CST Part II – 2017/18

## Semantics of Hoare logic

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

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

This lecture defines a formal semantics of Hoare logic, and introduces properties of Hoare logic (soundness & completeness).



## Semantics of Hoare logic

## Dynamic semantics of WHILE

## Dynamic semantics of WHILE

The dynamic semantics of $\mathrm{WHILE}$ 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 stacks, which are functions from variables to integers:

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

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

This models $\mathrm{WHILE}$ 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 inference rule schemas.

To reduce an assignment, we first evaluate the expression $E$ using the current stack, and update the stack 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 stack $s$, respectively.

For example, if $s(X) = 3$, then $\mathcal{E}[\![X + 2]\!](s) = 5$,
so $\langle Y := X + 2, s\rangle \Downarrow s[Y \mapsto 5]$.



## Semantics of expressions

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

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

$$\mathcal{E}[\![N]\!](s) \stackrel{\mathit{def}}{=} N$$

$$\mathcal{E}[\![V]\!](s) \stackrel{\mathit{def}}{=} s(V)$$

$$\mathcal{E}[\![E_1 + E_2]\!](s) \stackrel{\mathit{def}}{=} \mathcal{E}[\![E_1]\!](s) + \mathcal{E}[\![E_2]\!](s)$$

$$\vdots$$

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

For example, if $s(X) = 3$ and $s(Y) = 0$, then
$\mathcal{E}[\![X + 2]\!](s) = \mathcal{E}[\![X]\!](s) + \mathcal{E}[\![2]\!](s) = 3 + 2 = 5$, and
$\mathcal{E}[\![Y + 4]\!](s) = \mathcal{E}[\![Y]\!](s) + \mathcal{E}[\![4]\!](s) = 0 + 4 = 4$.

## Semantics of boolean expressions

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

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

$$\mathcal{B}[\![\mathbf{T}]\!](s) \stackrel{def}{=} \top$$

$$\mathcal{B}[\![\mathbf{F}]\!](s) \stackrel{def}{=} \bot$$

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

$$\vdots$$

For example, if $s(X) = 3$ and $s(Y) = 0$, then
$\mathcal{B}[\![X + 2 \geq Y + 4]\!](s) = \mathcal{E}[\![X + 2]\!](s) \geq \mathcal{E}[\![Y + 4]\!](s) = 5 \geq 4 = \top$.



## Big-step operational semantics of WHILE

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

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

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

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



## Example reduction in WHILE

For example, if $s(X) = 3$ and $s(Y) = 0$, then we have the following reduction derivation:

$$\frac{\mathcal{B}[\![X + 2 \geq Y + 4]\!](s) = \top \qquad \dfrac{\dfrac{}{\langle Y := 2 + X, s\rangle \Downarrow s[Y \mapsto 5]} \qquad \dfrac{}{\langle Y := Y + 1, s[Y \mapsto 5]\rangle \Downarrow s[Y \mapsto 6]}}{\langle Y := 2 + X; Y := Y + 1, s\rangle \Downarrow s[Y \mapsto 6]}}{\langle \textbf{if } X + 2 \geq Y + 4 \textbf{ then } (Y := 2 + X; Y := Y + 1) \textbf{ else } Y := 3, s\rangle \Downarrow s[Y \mapsto 6]}$$



# Properties of WHILE

## Determinacy

The dynamic semantics of $\mathrm{WHILE}$ 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 satisfy the postcondition.



## Substitution

We use $E_1[E_2/V]$ to denote $E_1$ with $E_2$ substituted for every occurrence of program variable $V$:

$$\begin{aligned}
-[=/\equiv] &: \mathit{Expr} \times \mathit{Expr} \times \mathit{Var} \to \mathit{Expr} \\
N[E_2/V] &\stackrel{\mathit{def}}{=} N \\
V'[E_2/V] &\stackrel{\mathit{def}}{=} \begin{cases} \text{if } V' = V & E_2 \\ \text{if } V' \neq V & V' \end{cases} \\
(E_a + E_b)[E_2/V] &\stackrel{\mathit{def}}{=} (E_a[E_2/V]) + (E_b[E_2/V]) \\
&\vdots
\end{aligned}$$

For example, $(X + (Y \times 2))[3 + Z/Y] = X + ((3 + Z) \times 2)$.



## Substitution property for expressions

We will use the following expression substitution property later:

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

The expression substitution property follows by induction on $E_1$.

Case $E_1 \equiv N$:

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



## Proof of substitution property: variable case

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

Case $E_1 \equiv V'$:

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

## Proof of substitution property: addition case

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

Case $E_1 \equiv E_a + E_b$:

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



# Semantics of assertions

## The language of assertions

Now, we have formally defined the dynamic 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, and
- assertions, which denote properties that may be true or false.

Since we are reasoning about WHILE states, our values will be integers, and our assertions will describe properties of WHILE states.

## Review of first-order logic: signature

In general, first-order logic is parameterised over a signature that defines 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: 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:

$$\begin{array}{ccc} X, & y, & Z, \\ 1, & 2, & 325, \\ -X, & -(X+1), & (x \times y) + Z, \\ \sqrt{(1+x^2)}, & X!, & \mathit{Kolmogorov}(x) \end{array}$$

Otherwise, we would have to write $X + 1$ as $+(X, 1)$.



## Review of first-order logic: atomic assertions

Examples of atomic assertions are:

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

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

Other atomic assertions are built from terms using predicate symbols and equality. Again, we use conventional notation:

$$X = 1, \quad (X+1)^2 \geq x^2, \quad \mathit{PRIME}(3), \quad \mathit{halts}(x)$$

Here $\geq$, *PRIME*, and *halts* are examples of predicates, and $X$, 1, $X + 1$, $(X + 1)^2$ and $x^2$ are examples of terms.

Otherwise, we would have to write $(X + 1)^2 \geq x^2$ as $\geq ({}^2(+(X, 1)), {}^2(x))$.



## Review of first-order logic: compound assertions

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

$$\wedge\ (\mathit{conjunction}), \vee\ (\mathit{disjunction}), \Rightarrow (\mathit{implication})$$

and quantification:

$$\forall\ (\mathit{universal}), \exists\ (\mathit{existential})$$

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

## The assertion language

The formal syntax of the assertion language is given below:

$$
\begin{array}{rcll}
\nu & ::= & V \mid v & & \textit{variables} \\
t & ::= & \nu \mid f(t_1, ..., t_n) & n \geq 0 & \textit{terms} \\
P, Q & ::= & \bot \mid \top \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q & & \textit{assertions} \\
& \mid & \forall v.\, P \mid \exists v.\, P \mid t_1 = t_2 \mid p(t_1, ..., t_n) & n \geq 0 & \\
\neg P & \stackrel{def}{=} & P \Rightarrow \bot & &
\end{array}
$$

Quantifiers quantify over terms, and only bind logical variables.

Here $f$ and $p$ range over an unspecified set of function symbols and predicate symbols, respectively, that includes (symbols for) the usual mathematical functions and predicates on integers.
In particular, we assume that they contain symbols that allows us to embed arithmetic expressions $E$ as terms, and boolean expressions $B$ as assertions.



## Semantics of terms

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

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

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

$$[\![f(t_1, ..., t_n)]\!](s) \stackrel{def}{=} [\![f]\!]([\![t_1]\!](s), ..., [\![t_n]\!](s))$$

We assume that the appropriate function $[\![f]\!]$ associated to each function symbol $f$ is provided along with the implicit signature.

In particular, we have $[\![E]\!](s) = \mathcal{E}[\![E]\!](s)$.



## Semantics of assertions

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

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

$$[\![\bot]\!] \stackrel{def}{=} \{s \in \mathit{Stack} \mid \bot\} = \emptyset$$

$$[\![\top]\!] \stackrel{def}{=} \{s \in \mathit{Stack} \mid \top\} = \mathit{Stack}$$

$$[\![P \vee Q]\!] \stackrel{def}{=} \{s \in \mathit{Stack} \mid s \in [\![P]\!] \vee s \in [\![Q]\!]\} = [\![P]\!] \cup [\![Q]\!]$$

$$[\![P \wedge Q]\!] \stackrel{def}{=} \{s \in \mathit{Stack} \mid s \in [\![P]\!] \wedge s \in [\![Q]\!]\} = [\![P]\!] \cap [\![Q]\!]$$

$$[\![P \Rightarrow Q]\!] \stackrel{def}{=} \{s \in \mathit{Stack} \mid s \in [\![P]\!] \Rightarrow s \in [\![Q]\!]\}$$

(continued)



## Semantics of assertions (continued)

$$[\![t_1 = t_2]\!] \stackrel{def}{=} \{s \in \mathit{Stack} \mid [\![t_1]\!](s) = [\![t_2]\!](s)\}$$

$$[\![p(t_1, ..., t_n)]\!] \stackrel{def}{=} \{s \in \mathit{Stack} \mid [\![p]\!]([\![t_1]\!](s), ..., [\![t_n]\!](s))\}$$

$$[\![\forall v.\, P]\!] \stackrel{def}{=} \{s \in \mathit{Stack} \mid \forall N.\, s[v \mapsto N] \in [\![P]\!]\}$$

$$[\![\exists v.\, P]\!] \stackrel{def}{=} \{s \in \mathit{Stack} \mid \exists N.\, s[v \mapsto N] \in [\![P]\!]\}$$

We assume that the appropriate predicate $[\![p]\!]$ associated to each predicate symbol $p$ is provided along with the implicit signature.

In particular, we have $[\![B]\!] = \{s \mid \mathcal{B}[\![B]\!](s) = \top\}$.

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:

$$(\forall v.\, P)[E/V] \stackrel{def}{=} \forall v.\, (P[E/V])$$

$$\vdots$$



# Semantics of Hoare logic

## 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: the former by using the substitution property for expressions, and the latter by using the former. (Exercise)

The latter property will be useful in the proof of soundness of the syntactic assignment rule.



## Semantics of partial correctness triples

Now that we have formally defined the dynamic semantics of $\mathrm{WHILE}$ 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:

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

## 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:

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

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

There is a blind spot here: we do not even have a way of saying that there are no other, non-terminating executions.



## Properties 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).

Hoare logic is sound but **not** complete.



## Properties of Hoare logic

## Soundness of Hoare logic

## 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 (that is what our induction hypothesis gives us), 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 $s[V \mapsto \mathcal{E}[\![E]\!](s)] \in [\![P]\!]$.

From inversion on the reduction, there exists an $N$ such that $\mathcal{E}[\![E]\!](s) = N$ and $s' = s[V \mapsto N]$, so $s' = s[V \mapsto \mathcal{E}[\![E]\!](s)]$.

Hence, $s' \in [\![P]\!]$.



## Soundness of the loop rule

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

How can we get past the fact that the loop reduction rules define the reduction of a loop in terms of itself?

We will prove $\models \{P\}$ **while** $B$ **do** $C\ \{P \wedge \neg B\}$ by proving a modified version of the property for a modified but equivalent reduction relation.



## Instrumented big-step operational semantics of WHILE

We can write an instrumented version of our big-step operational semantics of WHILE that counts how many times the body of the top-level loop is executed:

$$\frac{\mathcal{B}[\![B]\!](s) = \top \qquad \langle C, s\rangle \Downarrow s' \qquad \langle \textbf{while } B \textbf{ do } C, s'\rangle \Downarrow^n s''}{\langle \textbf{while } B \textbf{ do } C, s\rangle \Downarrow^{n+1} s''}$$

$$\frac{\mathcal{B}[\![B]\!](s) = \bot}{\langle \textbf{while } B \textbf{ do } C, s\rangle \Downarrow^0 s}$$

that is equivalent to the original dynamic semantics in the following sense:

$$\left(\langle \textbf{while } B \textbf{ do } C, s\rangle \Downarrow s'\right) \Leftrightarrow \left(\exists n.\, \langle \textbf{while } B \textbf{ do } C, s\rangle \Downarrow^n s'\right)$$

## Soundness of the loop rule: base case

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

We can prove this by a (nested) induction on $n$:

Case 0: assume $s \in [\![P]\!]$ and $\langle \textbf{while } B \textbf{ do } C, s\rangle \Downarrow^0 s'$.

Since the loop reduced in 0 iterations, $B$ must have evaluated to false: $\mathcal{B}[\![B]\!](s) = \bot$ and $s' = s$.

Since $\mathcal{B}[\![B]\!](s) = \bot$, $s \notin [\![B]\!]$, so $s \in [\![B]\!] \Rightarrow s \in [\![\bot]\!]$, so $s \in [\![B \Rightarrow \bot]\!]$, so $s \in [\![\neg B]\!]$. Therefore, $s \in [\![P \wedge \neg B]\!]$.
Hence, $s' = s \in [\![P \wedge \neg B]\!]$.



## Soundness of the loop rule: inductive case

If (IH) $\forall s, s'.\, s \in [\![P \wedge B]\!] \wedge \langle C, s\rangle \Downarrow s' \Rightarrow s' \in [\![P]\!]$, 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 $s \in [\![P]\!]$, $\langle \textbf{while } B \textbf{ do } C, s\rangle \Downarrow^{n+1} s'$, and
(nIH) $\forall s, s'.\, s \in [\![P]\!] \wedge \langle \textbf{while } B \textbf{ do } C, s\rangle \Downarrow^n s' \Rightarrow s' \in [\![P \wedge \neg B]\!]$.

Since the loop reduced in one iteration or more, $B$ must have evaluated to true: $\mathcal{B}[\![B]\!](s) = \top$, and there exists an $s^*$ such that $\langle C, s\rangle \Downarrow s^*$ and $\langle \textbf{while } B \textbf{ do } C, s^*\rangle \Downarrow^n s'$.

Since $\mathcal{B}[\![B]\!](s) = \top$, $s \in [\![B]\!]$. Therefore, $s \in [\![P \wedge B]\!]$.

From the outer induction hypothesis IH, it follows that $s^* \in [\![P]\!]$, and so by the inner induction hypothesis nIH, $s' \in [\![P \wedge \neg B]\!]$.



# Other properties of Hoare logic

## Completeness

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

Our Hoare logic inherits the incompleteness of arithmetic and is therefore **not** complete.

## Completeness

To see why, assume that, using our syntactic proof system, we can derive any triple that holds semantically.

Then, for every assertion $P$ that is true in arithmetic, that is, such that $\forall s.\, s \in [\![P]\!]$, and hence such that $\models \{\top\}$ **skip** $\{P\}$, we can derive $\vdash \{\top\}$ **skip** $\{P\}$.

Then, by examining that derivation, we have a derivation of $\vdash \top \Rightarrow P$, and hence a derivation of $\vdash P$.

Since the assertion logic (which includes arithmetic) 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 derive $\vdash \{P\}\ C\ \{Q\}$ for a statement that holds semantically can be traced back to a failure to prove $\vdash R$ for some valid arithmetic statement $R$.

In practice, completeness is not that important, and there is more focus on nice, usable rules.



## Decidability

Finally, Hoare logic is not decidable.

$\models \{\top\}\ C\ \{\bot\}$ holds if and only if $C$ does not terminate.
Moreover, we can encode Turing machines in $\mathrm{WHILE}$.
Hence, since the Halting problem is undecidable, so is Hoare logic.



## Summary

We have defined a dynamic semantics for the $\mathrm{WHILE}$ language, and a formal semantics for a Hoare logic for $\mathrm{WHILE}$.

We have shown that the syntactic 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.
- Software Foundations, Benjamin C. Pierce et al.

In the next lecture, we will look at examples of proofs in Hoare logic.

# Hoare logic

**Lecture 3: Examples in Hoare logic**

**Jean Pichon-Pharabod**
University of Cambridge

CST Part II – 2017/18

## Recap

In the past lectures, we have discussed Hoare logic: we have given

- a notation for specifying the intended behaviour of programs:

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

- a semantics capturing the precise meaning of this notation:

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

- a syntactic proof system for proving that programs satisfy their intended specification:

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

- a proof of soundness of that proof system:

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



## Introduction

Today, we will **use** Hoare logic, and look at how to find proofs.

We will first establish derived rules that make using Hoare logic easier.

Using these, we will then verify two simple programs to exercise Hoare logic, and to illustrate how to find invariants in Hoare logic.

We will also find proof rules for total correctness.



## Finding proofs

## Finding proofs: backwards reasoning

## Forward reasoning

The proof rules we have seen so far are best suited for **forward** (also "top down") 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:

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



## Proof of a simple assignment using the forward reasoning

$$\frac{\dfrac{\vdots}{\vdash X = 1 \Rightarrow X + 1 = a + 1} \quad \dfrac{}{\vdash \{(X = a + 1)[X + 1/X]\} X := X + 1 \{X = a + 1\}} \quad \dfrac{\vdots}{\vdash X = a + 1 \Rightarrow X = a + 1}}{\vdash \{X = a\} X := X + 1 \{X = a + 1\}}$$

Given that $(X = a + 1)[X + 1/X] \equiv X + 1 = a + 1$.



## Backwards reasoning & backwards assignment rule

It is often more natural to work **backwards** (also "bottom up"), 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:

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

This rule does not impose that the precondition is of a given shape, but instead that it implies an assertion of the desired shape.

## Backwards assignment rule

We can derive the backwards assignment rule by combining the assignment rule with the rule of consequence:

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



## 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 $R$ by using 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 loop rule

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 skip and conditional rules

We can also derive a backwards skip rule that builds in consequence:

$$\frac{\vdash P \Rightarrow Q}{\vdash \{P\}\ \textbf{skip}\ \{Q\}}$$

The conditional rule needs not be changed:

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

## Backwards reasoning proof rules

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

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

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

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

There is no separate rule of consequence anymore.
These rules are still relatively complete.



## Finding proofs: factorial

## Finding proofs: loop invariants

## Specifying a program computing factorial

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 specification of a program computing factorial

This corresponds to the following partial correctness triple:

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

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, meaning that it

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



## Analysing the factorial implementation

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

How does this program work?



## Observations about the factorial implementation

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

| iteration | $Y$ | $X$ |
|---|---|---|
| 0 | $1$ | $x$ |
| 1 | $1 \times x$ | $x - 1$ |
| 2 | $1 \times x \times (x - 1)$ | $x - 2$ |
| 3 | $1 \times x \times (x - 1) \times (x - 2)$ | $x - 3$ |
| $\vdots$ | $\vdots$ | $\vdots$ |
| $x$ | $1 \times x \times (x - 1) \times (x - 2) \times \cdots \times 1$ | $0$ |

$Y$ is the value computed so far, and $X!$ remains to be computed.

## An invariant for the factorial implementation

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

Take $I$ to be $Y \times X! = x! \wedge X \geq 0$.
(We need $X \geq 0$ for $X!$ to make sense.)



## Backwards reasoning proof rules (recap)

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

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

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

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



## Derivation tree of the verified factorial



## Finding proofs: proof outlines

## Proof outlines

Derivations in Hoare logic are often more readable when given as **proof outlines** instead of proof trees.

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

Sequences of Hoare logic assertions indicate reasoning about assertions.



## Finding proofs: Fibonacci

## Proof outline for the implementation of factorial

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



## A verified Fibonacci implementation

We wish to verify that the following command computes the $N$-th Fibonacci number (indexed from 1), and stores the result in $Y$. This corresponds to the following partial correctness Hoare triple:

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

Recall that the Fibonacci sequence is defined by

$$\mathit{fib}(1) = 1, \quad \mathit{fib}(2) = 1, \quad \forall n > 2.\, \mathit{fib}(n) = \mathit{fib}(n-1) + \mathit{fib}(n-2)$$

Moreover, for convenience, we assume $\mathit{fib}(0) = 0$.

## A verified Fibonacci implementation

Reasoning about the initial assignment of constants is easy.

How can we verify the loop?

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

First, we need to understand the implementation.



## Observations about the implementation of Fibonacci

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

| iteration | 0 | 1 | 2 | 3 | 4 | 5 | 6 | $\cdots$ | $n-1$ |
|---|---|---|---|---|---|---|---|---|---|
| $Y$ | 1 | 1 | 2 | 3 | 5 | 8 | 13 | $\cdots$ | $fib(n)$ |
| $X$ | 0 | 1 | 1 | 2 | 3 | 5 | 8 | $\cdots$ | $fib(n-1)$ |
| $Z$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | $\cdots$ | $n$ |



## Analysing the implementation of Fibonacci

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

$Z$ is used to count loop iterations, and $Y$ and $X$ are used to compute the Fibonacci number:
$Y$ contains the current Fibonacci number,
and $X$ contains the previous Fibonacci number.

This suggests trying the invariant
$Y = fib(Z) \wedge X = fib(Z-1) \wedge Z > 0$.
(We need $Z > 0$ for $fib(Z-1)$ to make sense.)



## Trying an invariant for the Fibonacci implementation

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

Take $I \equiv Y = fib(Z) \wedge X = fib(Z-1) \wedge Z > 0$.
Then we have to prove:

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

Do all these hold? Only the first two do. (Exercise.)

## A better invariant for the Fibonacci implementation

$\{X = 0 \wedge Y = 1 \wedge Z = 1 \wedge 1 \leq N \wedge N = n\}$

**while** $Z < N$ **do**

$(Y := X + Y; X := Y - X; Z := Z + 1)$

$\{Y = fib(n)\}$

While $Y = fib(Z) \wedge X = fib(Z-1) \wedge Z > 0$ **is an invariant**, it is not strong enough to establish the desired postcondition.

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

$$Y = fib(Z) \wedge X = fib(Z-1) \wedge Z > 0 \wedge Z \leq N \wedge N = n$$



## Proof outline for the loop of the Fibonacci implementation

$\{X = 0 \wedge Y = 1 \wedge Z = 1 \wedge 1 \leq N \wedge N = n\}$

$\{Y = fib(Z) \wedge X = fib(Z-1) \wedge Z > 0 \wedge Z \leq N \wedge N = n\}$

**while** $Z < N$ **do**

$(\{Y = fib(Z) \wedge X = fib(Z-1) \wedge Z > 0 \wedge Z \leq N \wedge N = n \wedge Z < N\}$

$\{X + Y = fib(Z+1) \wedge (X+Y) - X = fib(Z) \wedge Z + 1 > 0 \wedge Z + 1 \leq N \wedge N = n\}$

$Y := X + Y;$

$\{Y = fib(Z+1) \wedge Y - X = fib(Z) \wedge Z + 1 > 0 \wedge Z + 1 \leq N \wedge N = n\}$

$X := Y - X;$

$\{Y = fib(Z+1) \wedge X = fib(Z) \wedge Z + 1 > 0 \wedge Z + 1 \leq N \wedge N = n\}$

$\{Y = fib(Z+1) \wedge X = fib((Z+1)-1) \wedge Z + 1 > 0 \wedge Z + 1 \leq N \wedge N = n\}$

$Z := Z + 1$

$\{Y = fib(Z) \wedge X = fib(Z-1) \wedge Z > 0 \wedge Z \leq N \wedge N = n\})$

$\{Y = fib(Z) \wedge X = fib(Z-1) \wedge Z > 0 \wedge Z \leq N \wedge N = n \wedge \neg(Z < N)\}$

$\{Y = fib(n)\}$



## Summary of proof-finding

We have looked at how to find proofs:

- how "backwards" reasoning can help;
- how to find invariants.

Finding invariants is difficult!

Writing out full proof trees or even 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 simple proof obligations.



# Total correctness

## Total correctness

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

Recall: 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 commands that introduce non-termination.

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



## Unsoundness of the loop rule for total correctness

The loop rule that we have for partial correctness is not sound for total correctness:

$$\dfrac{\dfrac{\dfrac{\vdots}{\vdash (\top \wedge \mathbf{T}) \Rightarrow \top} \quad \dfrac{}{\vdash \{\top\}\ \mathbf{skip}\ \{\top\}} \quad \dfrac{\vdots}{\vdash \top \Rightarrow \top}}{\vdash \{\top \wedge \mathbf{T}\}\ \mathbf{skip}\ \{\top\}}}{\vdash \{\top\}\ \mathbf{while}\ \mathbf{T}\ \mathbf{do}\ \mathbf{skip}\ \{\top \wedge \neg \mathbf{T}\}} \quad \dfrac{\vdots}{\vdash \top \wedge \neg \mathbf{T} \Rightarrow \bot}$$
$$\vdash \{\top\}\ \mathbf{while}\ \mathbf{T}\ \mathbf{do}\ \mathbf{skip}\ \{\bot\}$$

If the loop rule were sound for total correctness, then this would show that **while T do skip** always terminates in a state satisfying $\bot$.



## Loop variants

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

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

If this is the case, then the loop terminates, as there would otherwise be an infinite decreasing sequence of natural numbers.

This decreasing quantity is called a variant.

## Loop rule for total correctness

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

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

The second hypothesis ensures that the variant is non-negative.



## Backwards reasoning total correctness loop rule

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

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



## 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

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

## Relation between partial and total correctness

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

$$\text{total correctness} = \text{partial correctness} + \text{termination}$$

This is captured formally by the following properties:

- If $\vdash \{P\}\ C\ \{Q\}$ and $\vdash [P]\ C\ [\top]$, then $\vdash [P]\ C\ [Q]$.
- If $\vdash [P]\ C\ [Q]$, then $\vdash \{P\}\ C\ \{Q\}$.



# Hoare logic

**Lecture 4: A verifier for Hoare logic**

**Jean Pichon-Pharabod**
University of Cambridge

CST Part II – 2017/18

## Summary of total correctness

We have given rules for total correctness,
similar to those for partial correctness.

Only the loop rule differs: the premises of the loop rule require that the loop body decreases a non-negative expression.

It is even possible to do amortised, asymptotic complexity analysis in Hoare logic:

- A Fistful of Dollars, Armaël Guéneau et al., ESOP 2018

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



## Introduction

Last time, we saw that that proofs in Hoare logic can involve large amounts of very error-prone bookkeeping which distract from the actual task of finding invariants, 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 to automate the routine parts of proofs in Hoare logic, and reduce the likelihood of errors.

We will also look at other perspectives on Hoare triples.

# Mechanised Program Verification

## Automated theorem proving

Recall (from Part IB Computation theory) that it is impossible to design a decision procedure determining whether arbitrary mathematical statements hold.

This does not mean that one cannot have procedures that will prove many useful statements.
For example, SMT solvers work quite well.

Using these, it is quite possible to build a system that will mechanise the routine aspects of verification.

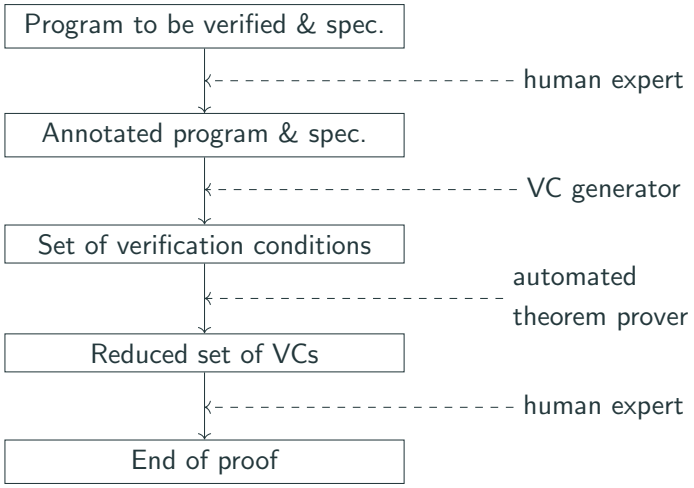


## Verification conditions

The idea is, given a program and a specification,
to generate a set of statements of first-order logic called
**verification conditions** (abbreviated VC)
such that if all the VCs hold, then the specification holds.



## 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 VCs expressed in first-order logic.

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

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



## Using a verifier

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

1. The user annotates the program by inserting assertions expressing conditions that are meant to hold whenever execution reaches the given annotation.
2. The VC generator generates the associated VCs.
3. An automated theorem prover attempts to prove as many of the VCs as it can.
4. the user proves the remaining VCs (if any).



## Limits of verifiers

Verifiers are not a silver bullet!

- Inserting appropriate annotations is tricky:
  - finding loop invariants requires a good understanding of how the program works;
  - writing assertions so as to help automated theorem provers discharge the VCs requires a good understanding of how they work.
- The verification conditions left over from step 3 may bear little resemblance to annotations and specification written by the user.



## Example use of a verifier

## Example

We will illustrate the process with the following Euclidian division example (here, $Q$ and $R$ are program variables, not assertions):

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

Note: this is a "bad" specification; it should probably talk about the initial state of $X$ instead.



## Annotating the example

Step 1 is to annotate the program with two assertions:

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



## VCs for the 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.

Here, all of them can be discharged.



# The VC generator

## Design of the VC generator

If we have enough annotations to not have to guess how to apply them, looking at the backwards reasoning rules from a logic programming perspective suggests an algorithm to collect first-order logic constraints on derivability:

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

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

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

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



## Annotation of commands

A properly annotated command is a command with extra assertions embedded within it as follows:

$$\begin{array}{rcl} \mathcal{C} & ::= & \textbf{skip} \\ & | & \mathcal{C}_1; \{R\}\ \mathcal{C}_2 \\ & | & \mathcal{C}; V := E \\ & | & V := E \\ & | & \textbf{if}\ B\ \textbf{then}\ \mathcal{C}_1\ \textbf{else}\ \mathcal{C}_2 \\ & | & \textbf{while}\ B\ \textbf{do}\ \{I\}\ \mathcal{C} \end{array}$$

(We overload command constructors.)
These are the places where one had to guess an assertion in our backwards reasoning rules.

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



## Erasure function

To use the verifier to verify a command $C$, a human expert has to propose an annotated version of the command to be verified, that is, an annotated command $\mathcal{C}$ such that $|\mathcal{C}| = C$, where $|-|$ is the following erasure function:

$$\begin{array}{rcl} |\textbf{skip}| & \stackrel{def}{=} & \textbf{skip} \\ |\mathcal{C}_1; \{R\}\ \mathcal{C}_2| & \stackrel{def}{=} & |\mathcal{C}_1|; |\mathcal{C}_2| \\ |\mathcal{C}; V := E| & \stackrel{def}{=} & |\mathcal{C}|; V := E \\ |V := E| & \stackrel{def}{=} & V := E \\ |\textbf{if}\ B\ \textbf{then}\ \mathcal{C}_1\ \textbf{else}\ \mathcal{C}_2| & \stackrel{def}{=} & \textbf{if}\ B\ \textbf{then}\ |\mathcal{C}_1|\ \textbf{else}\ |\mathcal{C}_2| \\ |\textbf{while}\ B\ \textbf{do}\ \{I\}\ \mathcal{C}| & \stackrel{def}{=} & \textbf{while}\ B\ \textbf{do}\ |\mathcal{C}| \end{array}$$



## Example of annotated command

The following annotated command is an annotated version of a variant of) our factorial program from the previous lecture, suitably annotated to establish the specification
$\{X = x \wedge X \geq 0\} \ldots \{Y = x!\}$:

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

## Generating VCs

We can now define the VC generator.

We will define it as a function $VC(P, \mathcal{C}, Q)$ that gives a set of verification conditions for a properly annotated command $\mathcal{C}$ and pre- and postconditions $P$ and $Q$.

The function will be defined by recursion on $\mathcal{C}$, and is easily implementable.



## Backwards reasoning proof rules (recap)

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

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

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

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



## Backwards reasoning proof rules, given annotations

$$\frac{\vdash P \Rightarrow Q}{\vdash \{P\}\ |\mathbf{skip}|\ \{Q\}} \qquad \frac{\vdash \{P\}\ |\mathcal{C}_1|\ \{R\} \qquad \vdash \{R\}\ |\mathcal{C}_2|\ \{Q\}}{\vdash \{P\}\ |\mathcal{C}_1; \{R\}\ \mathcal{C}_2|\ \{Q\}}$$

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

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

$$\frac{\vdash \{P \wedge B\}\ |\mathcal{C}_1|\ \{Q\} \qquad \vdash \{P \wedge \neg B\}\ |\mathcal{C}_2|\ \{Q\}}{\vdash \{P\}\ |\mathbf{if}\ B\ \mathbf{then}\ \mathcal{C}_1\ \mathbf{else}\ \mathcal{C}_2|\ \{Q\}}$$

All the guessing has been pushed into the annotations.



## Soundness of VCs

We want our VC generator to be sound, in the sense that if all the VCs generated for $P$, $\mathcal{C}$, and $Q$ are derivable, then $\{P\}\ |\mathcal{C}|\ \{Q\}$ is derivable in Hoare Logic.

Formally,

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

We will write

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

which intuitively means "we generate sufficient VCs for $\mathcal{C}$", and will prove $\forall \mathcal{C}.\, \psi(\mathcal{C})$ by induction on $\mathcal{C}$.

## VCs for assignments

Recall

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

This suggests defining

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

Example:

$$\begin{aligned} & VC(X = 0,\ X := X + 1,\ X = 1) \\ & \quad = \{X = 0 \Rightarrow (X = 1)[X + 1/X]\} \\ & \quad = \{X = 0 \Rightarrow X + 1 = 1\} \end{aligned}$$



## Soundness of VCs for assignments

How can we show that we generate sufficient VCs for assignments, that is, formally, $\psi(V := E)$?

Recall

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

Fix $P$ and $Q$. Assume $\forall \phi \in VC(P, V := E, Q).\vdash \phi$.

Therefore, from the definition of $VC(P, V := E, Q)$, we have $\vdash P \Rightarrow Q[E/V]$.

Therefore, by the backwards reasoning assignment rule, we have $\vdash \{P\}\ |V := E|\ \{Q\}$.



## VCs for conditionals

Recall

$$\frac{\vdash \{P \wedge B\}\ |\mathcal{C}_1|\ \{Q\} \qquad \vdash \{P \wedge \neg B\}\ |\mathcal{C}_2|\ \{Q\}}{\vdash \{P\}\ |\textbf{if } B \textbf{ then } \mathcal{C}_1 \textbf{ else } \mathcal{C}_2|\ \{Q\}}$$

This suggests defining

$$\begin{aligned} & VC(P, \textbf{if } B \textbf{ then } \mathcal{C}_1 \textbf{ else } \mathcal{C}_2, Q) \stackrel{def}{=} \\ & \quad VC(P \wedge B, \mathcal{C}_1, Q) \cup VC(P \wedge \neg B, \mathcal{C}_2, Q) \end{aligned}$$



## Example of VCs for conditionals

Example: The verification conditions for the earlier "bad" specification of the earlier maximum program are

$$\begin{aligned} & VC(\top,\ \textbf{if } X \geq Y \textbf{ then } Z := X \textbf{ else } Z := Y,\ Z = max(X, Y)) \\ & = VC(\top \wedge X \geq Y,\ Z := X,\ Z = max(X, Y)) \cup \\ & \quad VC(\top \wedge \neg(X \geq Y),\ Z := Y,\ Z = max(X, Y)) \\ & = \{(\top \wedge X \geq Y) \Rightarrow (Z = max(X, Y))[X/Z], \\ & \quad (\top \wedge \neg(X \geq Y)) \Rightarrow (Z = max(X, Y))[Y/Z]\} \\ & = \{(\top \wedge X \geq Y) \Rightarrow (X = max(X, Y)), \\ & \quad (\top \wedge \neg(X \geq Y)) \Rightarrow (Y = max(X, Y))\} \end{aligned}$$

These are easily shown to be true arithmetic statements.

## Soundness of VCs for conditionals

How can we show that we generate sufficient VCs for a conditional, that is, formally, $\psi(\textbf{if } B \textbf{ then } \mathcal{C}_1 \textbf{ else } \mathcal{C}_2)$, assuming that we generate sufficient VCs for the "then" and the "else" branch, that is, formally (IH1) $\psi(\mathcal{C}_1)$ and (IH2) $\psi(\mathcal{C}_2)$?

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

Fix $P$ and $Q$. Assume $\forall \phi \in VC(P, \textbf{if } B \textbf{ then } \mathcal{C}_1 \textbf{ else } \mathcal{C}_2, Q).\vdash \phi$.

Therefore, from the definition of $VC(P, \textbf{if } B \textbf{ then } \mathcal{C}_1 \textbf{ else } \mathcal{C}_2, Q)$, we have $\forall \phi \in VC(P \wedge B, \mathcal{C}_1, Q).\vdash \phi$ and $\forall \phi \in VC(P \wedge \neg B, \mathcal{C}_2, Q).\vdash \phi$.
Therefore, by the induction hypotheses $\psi(\mathcal{C}_1)$ and $\psi(\mathcal{C}_2)$, we have $\vdash \{P \wedge B\}\ |\mathcal{C}_1|\ \{Q\}$ and $\vdash \{P \wedge \neg B\}\ |\mathcal{C}_2|\ \{Q\}$.
Therefore, by the backwards reasoning conditional rule, we have $\vdash \{P\}\ |\textbf{if } B \textbf{ then } \mathcal{C}_1 \textbf{ else } \mathcal{C}_2|\ \{Q\}$.



## VCs for sequences

Recall

$$\frac{\vdash \{P\}\ |\mathcal{C}_1|\ \{R\} \qquad \vdash \{R\}\ |\mathcal{C}_2|\ \{Q\}}{\vdash \{P\}\ |\mathcal{C}_1; \{R\}\ \mathcal{C}_2|\ \{Q\}} \qquad \frac{\vdash \{P\}\ |\mathcal{C}|\ \{Q[E/V]\}}{\vdash \{P\}\ |\mathcal{C}; V := E|\ \{Q\}}$$

This suggests defining

$$VC(P,\ \mathcal{C}_1; \{R\}\ \mathcal{C}_2,\ Q) \stackrel{def}{=} VC(P, \mathcal{C}_1, R) \cup VC(R, \mathcal{C}_2, Q)$$
$$VC(P,\ \mathcal{C}; V := E,\ Q) \stackrel{def}{=} VC(P, \mathcal{C}, Q[E/V])$$



## Example of VCs for sequences

We can compute the VCs for a command swapping the values of $X$ and $Y$ using an intermediate variable $R$, with the specification we saw using auxiliary variables:

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



## Soundness of VCs for sequences

To justify the VCs generated for sequences, it suffices to prove that

$$\psi(\mathcal{C}_1) \wedge \psi(\mathcal{C}_2) \Rightarrow \psi(\mathcal{C}_1; \{R\}\ \mathcal{C}_2), \qquad \text{and}$$
$$\psi(\mathcal{C}) \Rightarrow \psi(\mathcal{C}; V := E)$$

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

## VCs for skip

Recall

$$\frac{\vdash P \Rightarrow Q}{\vdash \{P\}\ |\mathbf{skip}|\ \{Q\}}$$

Exercise: What does this suggest defining $VC(P, \mathbf{skip}, Q)$ as?

Proving soundness is also left as an exercise.



## VCs for loops

Recall

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

This suggests defining

$$VC(P, \mathbf{while}\ B\ \mathbf{do}\ \{I\}\ \mathcal{C}, Q) \stackrel{def}{=} \{P \Rightarrow I\} \cup VC(I \wedge B, \mathcal{C}, I) \cup \{I \wedge \neg B \Rightarrow Q\}$$



## Soundness of VCs for loops

How can we show that we generate sufficient VCs for a loop, that is, formally, $\psi(\mathbf{while}\ B\ \mathbf{do}\ \{I\}\ \mathcal{C})$, assuming that we generate sufficient VCs for the body, that is, formally, (IH) $\psi(C)$?

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

Fix $P$ and $Q$. Assume $\forall \phi \in VC(P, \mathbf{while}\ B\ \mathbf{do}\ \{I\}\ \mathcal{C}, Q).\vdash \phi$.

Therefore, from the definition of $VC(P, \mathbf{while}\ B\ \mathbf{do}\ \{I\}\ \mathcal{C}, Q)$, we have $\vdash P \Rightarrow I$, $\forall \phi \in VC(I \wedge B, \mathcal{C}, I).\vdash \phi$, and $\vdash I \wedge \neg B \Rightarrow Q$.

Therefore, by the induction hypothesis, we have $\vdash \{I \wedge B\}\ |\mathcal{C}|\ \{I\}$.

Therefore, by the backwards reasoning rule for loops, we have

$$\vdash \{P\}\ |\mathbf{while}\ B\ \mathbf{do}\ \{I\}\ \mathcal{C}|\ \{Q\}$$



## Summary of VCs

We have outlined the design of a semi-automated program verifier.

It takes an annotated program and a specification, and generates a set of first-order logic statements that, if derivable, ensure that the specification is derivable. It tries to discharge the easy statements by using automated theorem provers.

Intelligence is still required to provide the annotations, in particular loop invariants, to write them so as to help the automated theorem provers, and to discharge the difficult statements.

Soundness of the verifier is justified by the derived Hoare logic rules for backwards reasoning from the last lecture.

## VCs in practice

The ideas are very old, dating back to JC King's PhD in 1969, and the Stanford verifier in the 1970s.

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);
- Why3 (state of the art, used by SPARK): `http://why3.lri.fr/`.

These tools do much more work that our sketch of a verifier, supporting much more complex languages, including data structures, and interface with many automated theorem provers, providing them well-phrased statements.



## Other perspectives on 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 $Q$, $sp(P, C)$? ('strongest postcondition')

Symmetrically, if we are given $C$ and $Q$, can we infer a $P$?
Is there a best such $P$, $wlp(C, Q)$? ('weakest liberal precondition')

We are looking for functions $wlp$ and $sp$ such that

$$(\vdash P \Rightarrow wlp(C, Q)) \Leftrightarrow \vdash \{P\}\ C\ \{Q\} \Leftrightarrow (\vdash sp(P, C) \Rightarrow Q)$$

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



## Other perspectives on Hoare triples

## Terminology

Recall, if $P$ and $Q$ are assertions, $P$ is stronger than $Q$, and $Q$ is weaker than $P$, when $P \Rightarrow Q$.

We write *wlp* and talk about weakest **liberal** precondition because we only consider partial correctness.
For historical reasons, we do not say strongest liberal precondition because people only considered strongest postconditions for partial correctness.

This has no relevance here because, as we will see, there is no effective general finite formula for weakest preconditions, liberal or not, or strongest postconditions, for commands containing loops, so we will not consider weakest preconditions, liberal or not, for loops, so there is no difference between partial and total correctness.

## Computing weakest liberal preconditions (except for loops)

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

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

These rules are suggested by the relative completeness of the Hoare logic proof rules from the first lecture.



## Example of weakest liberal precondition computation

$$\begin{aligned} & wlp(X := X + 1; Y := Y + X,\ \exists m, n.\, X = 2 \times m \wedge Y = 2 \times n) \\ = {} & wlp(X := X + 1,\ wlp(Y := Y + X,\ \exists m, n.\, X = 2 \times m \wedge Y = 2 \times n)) \\ = {} & wlp(X := X + 1,\ (\exists m, n.\, X = 2 \times m \wedge Y = 2 \times n)[Y + X/Y]) \\ = {} & wlp(X := X + 1,\ \exists m, n.\, X = 2 \times m \wedge Y + X = 2 \times n) \\ = {} & (\exists m, n.\, X = 2 \times m \wedge Y + X = 2 \times n)[X + 1/X] \\ = {} & \exists m, n.\, X + 1 = 2 \times m \wedge Y + (X + 1) = 2 \times n \\ \Leftrightarrow {} & \exists m, n.\, X = 2 \times m + 1 \wedge Y = 2 \times n \end{aligned}$$



## Weakest preconditions for loops

While the following property holds for loops

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

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

There is no general finite formula for $wlp(\textbf{while } B \textbf{ do } C, Q)$ in first-order logic. (Otherwise, it would be easy to find invariants!)



## Relaxing annotations required for the VC generator

We can relax the syntax of annotated commands to include commands $C$ when $C$ is loop-free, and take

$$VC(P, C, Q) \stackrel{def}{=} \{P \Rightarrow wlp(C, Q)\}$$

(or $sp(P, C) \Rightarrow Q$).

Actual verifiers like Why3 include this and many other tricks to reduce how many assertions the user has to provide.

## Computing strongest postconditions (except for loops)

Strongest postconditions work symmetrically:

$$\begin{aligned} sp(P, \textbf{skip}) &= P \\ sp(P, V := E) &= \exists n.\, (V = E[n/V]) \wedge P[n/V] \\ sp(P, C_1; C_2) &= sp(sp(P, C_1), C_2) \\ sp(P, \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2) &= sp(P \wedge B, C_1) \vee sp(P \wedge \neg B, C_2) \end{aligned}$$

and suffer from the same problem with loops: there is no general finite formula for $sp(P, \textbf{while } B \textbf{ do } C)$ in first-order logic.

The strongest postcondition for assignments corresponds to the premise of Floyd's rule for assignment.



## Example of strongest postcondition

$$\begin{aligned} & sp(\exists m.\, X = 2 \times m,\ X := X + 1) \\ & \quad = \exists n.\, X = (X + 1)[n/X] \wedge (\exists m.\, X = 2 \times m)[n/X] \\ & \quad = \exists n.\, X = n + 1 \wedge (\exists m.\, n = 2 \times m) \\ & \quad \Leftrightarrow \exists m.\, X = 2 \times m + 1 \end{aligned}$$



## Symbolic execution

Determining the strongest postconditions $sp(P, C)$ corresponds to symbolically executing command $C$ under assumption $P$.

Symmetrically, determining the weakest liberal precondition $wlp(C, Q)$ corresponds to symbolically executing command $C$ backwards assuming the final state satisfies $Q$.



## Automatically finding loop invariants

Fully automated verification techniques need to circumvent the lack of a general finite formula for loops in first-order logic, rather than putting the onus on the human expert. There are several approaches:

- considering only programs with a finite number of states, as in traditional **model checking**;
- considering only executions of bounded length, as in **bounded model checking**;
- trying to soundly approximate the strongest invariants, as in abstract interpretation;
- ...

## Program refinement

We have focused on proving that a given program meets a given specification.

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 and the Vienna Development Method (VDM) are based on this idea.

Used for the automated Paris Metro Lines 14 and 1, and the Charles de Gaulle airport shuttle:
`http://rodin.cs.ncl.ac.uk/Publications/fm_sc_rs_v2.pdf`

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



## Summary

We have sketched the design a simple verifier, and justified its soundness using Hoare logic.

Weakest liberal preconditions (or strongest postconditions) can be used to reduce the number of annotations required in loop-free code.

In the next lecture, we will look at how to reason about programs with pointers.

Today, Tony Hoare is giving a talk at 14:00 in FW26:
"Logic for Program Development, Verification and Implementation".



# Hoare logic

**Lecture 5: Introduction to separation logic**

**Jean Pichon-Pharabod**
University of Cambridge

CST Part II – 2017/18

## Introduction

In the previous lectures, we have considered a language, $\mathrm{WHILE}$, where mutability only concerned program variables.

In this lecture, we will extend the $\mathrm{WHILE}$ language with pointer operations on a heap, and introduce an extension of Hoare logic, called separation logic, to enable practical reasoning about pointers.

## WHILE$_p$, a language with pointers

## Syntax of WHILE$_p$

We introduce new commands to manipulate the heap:

$$
\begin{array}{rcll}
E & ::= & N \mid V \mid E_1 + E_2 & \textit{arithmetic expressions} \\
 & \mid & E_1 - E_2 \mid E_1 \times E_2 \mid \cdots & \\
\textbf{null} & \stackrel{def}{=} & 0 &
\end{array}
$$

$$
\begin{array}{rcll}
B & ::= & \textbf{T} \mid \textbf{F} \mid E_1 = E_2 & \textit{boolean expressions} \\
 & \mid & E_1 \leq E_2 \mid E_1 \geq E_2 \mid \cdots &
\end{array}
$$

$$
\begin{array}{rcll}
C & ::= & \textbf{skip} \mid C_1; C_2 \mid V := E & \textit{commands} \\
 & \mid & \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2 & \\
 & \mid & \textbf{while } B \textbf{ do } C & \\
 & \mid & V := [E] \mid [E_1] := E_2 & \\
 & \mid & V := \textbf{alloc}(E_0, ..., E_n) & \\
 & \mid & \textbf{dispose}(E) &
\end{array}
$$



## The heap

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

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

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

It allows us to consider faults, and how separation logic can be used to prevent faults, and it also makes things clearer.



## Heap usage commands

Heap assignment command $[E_1] := E_2$

- evaluates $E_1$ to a location $\ell$ and $E_2$ to a value $N$, and updates the heap to map $\ell$ to $N$; faults if $\ell$ is not currently allocated.

Heap dereferencing command $V := [E]$

- evaluates $E$ to a location $\ell$, and assigns the value that $\ell$ maps to to $V$; faults if $\ell$ is not currently allocated.

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

## Heap management commands

Allocation assignment command: $V := \textbf{alloc}(E_0, ..., E_n)$

- chooses $n+1$ **consecutive** unallocated locations starting at location $\ell$, evaluates $E_0, ..., E_n$ to values $N_0, ..., N_n$, updates the heap to map $\ell + i$ to $N_i$ for each $i$, and assigns $\ell$ to $V$.

In $\mathrm{WHILE}_p$, allocation never faults.
A real machine would run out of memory at some point.

Deallocation command $\textbf{dispose}(E)$

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



## Pointers

$\mathrm{WHILE}_p$ has proper pointer operations, as opposed for example to references:

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

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.

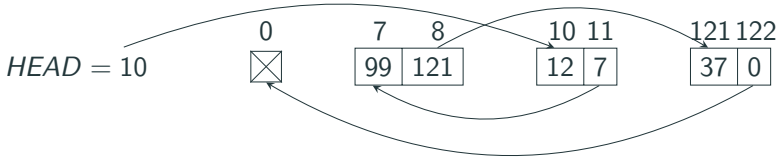


## Pointers and data structures

In $\mathrm{WHILE}_p$, 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:

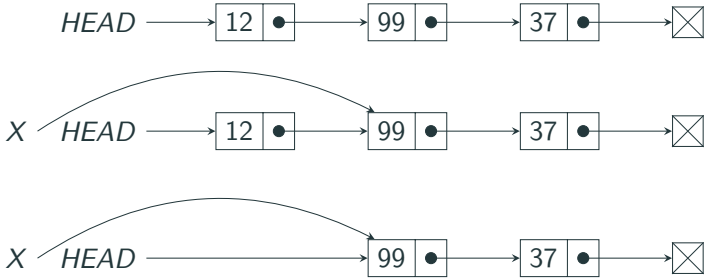

In $\mathrm{WHILE}$, we would have had to encode that in integers, for example as $HEAD = 2^{12} \times 3^{99} \times 5^{37}$ (as in Part IB Computation theory).

More concretely:






## Operations on mutable data structures




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

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

## Dynamic semantics of WHILE$_p$

## States of WHILE$_p$

For the WHILE language, we modelled the state as a function mapping program variables to values (integers):

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

For WHILE$_p$, we extend the state to be composed of a **stack** and a **heap**, where

- the stack maps program variables to values (as before), and
- the heap maps allocated locations to values.

We have

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



## Heaps

We elect for locations to be non-negative integers:

$$\ell \in \mathit{Loc} \stackrel{\mathit{def}}{=} \{\ell \in \mathbb{Z} \mid 0 \leq \ell\}$$

**null** is a location, but a "bad" one, that is never allocated.

To model the fact that only a finite number of locations is allocated at any given time, we model the heap as a **finite** function, that is, a partial function with a finite domain:

$$h \in \mathit{Heap} \stackrel{\mathit{def}}{=} (\mathit{Loc} \setminus \{\mathbf{null}\}) \stackrel{\mathit{fin}}{\to} \mathbb{Z}$$



## Failure of commands

WHILE$_p$ commands can fail by:

- dereferencing an invalid pointer,
- assigning to an invalid pointer, or
- deallocating an invalid pointer.

because the location expression we provided does not evaluate to a location, or evaluates to a location that is not allocated (which includes **null**).

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

$$\Downarrow : \mathcal{P}(\mathit{Cmd} \times \mathit{State} \times (\{\lightning\} + \mathit{State}))$$

We could instead just leave the configuration stuck, but explicit failure makes things clearer and easier to state.

## Adapting the base constructs to handle the heap

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

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

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

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

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



## Adapting the base constructs to handle failure

They can also be adapted to handle failure in the expected way:

$$\frac{\langle C_1, (s,h)\rangle \Downarrow \lightning}{\langle C_1; C_2, (s,h)\rangle \Downarrow \lightning} \qquad \frac{\langle C_1, s\rangle \Downarrow (s',h') \qquad \langle C_2, (s',h')\rangle \Downarrow \lightning}{\langle C_1; C_2, (s,h)\rangle \Downarrow \lightning}$$

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

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

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



## Heap dereferencing

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

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

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

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



## Heap assignment

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

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

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

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

## For reference: deallocation

Deallocating an allocated location removes that location from the heap:

$$\frac{\mathcal{E}[\![E]\!](s) = \ell \qquad \ell \in dom(h)}{\langle \textbf{dispose}(E), (s,h)\rangle \Downarrow (s, h \setminus \{(\ell, h(\ell))\})}$$

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

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



## For reference: 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:

$$\frac{\begin{array}{c}\mathcal{E}[\![E_0]\!](s) = N_0 \qquad \ldots \qquad \mathcal{E}[\![E_n]\!](s) = N_n \\ \forall i \in \{0, \ldots, n\}.\, \ell + i \notin dom(h) \\ \ell \neq \textbf{null}\end{array}}{\langle V := \textbf{alloc}(E_0, \ldots, E_n), (s,h)\rangle \Downarrow (s[V \mapsto \ell], h[\ell \mapsto N_1, \ldots, \ell + n \mapsto N_n])}$$

Because the heap has a finite domain, it is always possible to pick a suitable $\ell$, so allocation never faults.



# 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 (without 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 $C$:

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

$$\begin{aligned} mod([E_1] := E_2) &= \emptyset \\ mod(V := [E]) &= \{V\} \\ mod(V := \textbf{alloc}(E_0, \ldots, E_n)) &= \{V\} \\ mod(\textbf{dispose}(E)) &= \emptyset \end{aligned}$$



## For reference: free variables

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

$$\begin{aligned} FV(-) &: Term \to \mathcal{P}(Var) \\ FV(\nu) &\stackrel{def}{=} \{\nu\} \\ FV(f(t_1, \ldots, t_n)) &\stackrel{def}{=} FV(t_1) \cup \ldots \cup FV(t_n) \end{aligned}$$

and

$$\begin{aligned} FV(-) &: Assertion \to \mathcal{P}(Var) \\ FV(\top) = FV(\bot) &\stackrel{def}{=} \emptyset \\ FV(P \wedge Q) = FV(P \vee Q) = FV(P \Rightarrow Q) &\stackrel{def}{=} FV(P) \cup FV(Q) \\ FV(\forall v.\, P) = FV(\exists v.\, P) &\stackrel{def}{=} FV(P) \setminus \{v\} \\ FV(t_1 = t_2) &\stackrel{def}{=} FV(t_1) \cup FV(t_2) \\ FV(p(t_1, \ldots, t_n)) &\stackrel{def}{=} FV(t_1) \cup \ldots FV(t_n) \end{aligned}$$

respectively.



## 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:

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

This rule is admissible in standard Hoare logic.



## 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:

$$\vdash \{P\}\ C_1\ \{Q\} \qquad\qquad \vdash \{R\}\ C_2\ \{S\}$$

and then, as long as they use different program variables, we can compose them.
For example, if $mod(C_1) \cap FV(R) = \emptyset$ and $mod(C_2) \cap FV(Q) = \emptyset$, we can compose them sequentially:

$$\dfrac{\dfrac{\vdash \{P\}\ C_1\ \{Q\} \qquad mod(C_1) \cap FV(R) = \emptyset}{\vdash \{P \wedge R\}\ C_1\ \{Q \wedge R\}} \qquad \dfrac{\vdash R \wedge Q \Rightarrow Q \wedge R \qquad \dfrac{\vdash \{R\}\ C_2\ \{S\} \qquad mod(C_2) \cap FV(Q) = \emptyset}{\vdash \{R \wedge Q\}\ C_2\ \{S \wedge Q\}} \qquad \vdash S \wedge Q \Rightarrow Q \wedge S}{\vdash \{Q \wedge R\}\ C_2\ \{Q \wedge S\}}}{\vdash \{P \wedge R\}\ C_1; C_2\ \{Q \wedge S\}}$$

## A bad rule for reasoning about pointers

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

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

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

$$\frac{\vdash \{\top\}\ [37] := 42\ \{37 \hookrightarrow 42\} \qquad \mathit{mod}([37] := 42) \cap \mathit{FV}(Y \hookrightarrow 0) = \emptyset}{\vdash \{\top \wedge Y \hookrightarrow 0\}\ [37] := 42\ \{37 \hookrightarrow 42 \wedge Y \hookrightarrow 0\}}$$

even if $Y = 37$, in which case the postcondition would require 0 to be equal to 42.

There is a problem!



# Separation logic

## 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:

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

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.



## Separation logic

Separation logic is an extension of Hoare logic that simplifies reasoning about pointers by using new connectives to control aliasing.

The variant of separation logic that we are going to consider, which is suited to reason about an explicitly managed heap (as opposed to a heap with garbage collection), is called classical separation logic (as opposed to intuitionistic separation logic).

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.

## Concepts of separation logic

Separation logic introduces two new concepts for reasoning about pointers:

- **ownership**: separation logic assertions not only describe properties of the current state (as Hoare logic assertions did), but also assert ownership of part of the heap.
- **separation**: separation logic introduces a new connective for reasoning about the combination of **disjoint** parts of the heap.



## The points-to assertion

Separation logic introduces a new assertion, written $t_1 \mapsto t_2$, and read "$t_1$ points to $t_2$", for reasoning about individual heap cells.

The points-to assertion $t_1 \mapsto t_2$

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

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



## The separating conjunction

Separation logic introduces a new connective, the separating conjunction $*$, for reasoning about disjointedness.

The assertion $P * Q$ asserts that $P$ and $Q$ hold (like $P \wedge Q$), and that moreover the parts of the heap owned by $P$ and $Q$ are **disjoint**.

The separating conjunction has a neutral element, *emp*, which describes the empty heap: $emp * P \Leftrightarrow P \Leftrightarrow P * emp$.



## Examples of separation logic assertions

1. $(X \mapsto t_1) * (Y \mapsto t_2)$

   This assertion is unsatisfiable in a state where $X$ and $Y$ refer to the same location, since $X \mapsto t_1$ and $Y \mapsto t_2$ would both assert ownership of the same location.

   The following heap satisfies the assertion:

2. $(X \mapsto t) * (X \mapsto t)$

   This assertion is not satisfiable, as $X$ is not disjoint from itself.

## Examples of separation logic assertions

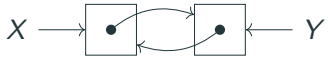
3. $X \mapsto t_1 \wedge Y \mapsto t_2$

   This asserts that $X$ and $Y$ alias each other and $t_1 = t_2$:

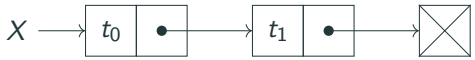



## Examples of separation logic assertions

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




5. $(X \mapsto t_0, Y) * (Y \mapsto t_1, \textbf{null})$




   Here, $X \mapsto t_0, ..., t_n$ is shorthand for

$$(X \mapsto t_0) * ((X + 1) \mapsto t_1) * \cdots * ((X + n) \mapsto t_n)$$



## Example use of the separating conjunction

6. $\exists x, y.\ \ (HEAD \mapsto 12, x) * (x \mapsto 99, y) * (y \mapsto 37, \textbf{null})$

   This describes our singly linked list from earlier:




# Semantics of separation logic assertions

## 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 store $s$, 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 $dom(h)$.

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

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

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



## 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 $t_2$:

$$
[\![t_1 \mapsto t_2]\!](s) \stackrel{def}{=} \left\{ h \in \mathit{Heap} \;\middle|\; \exists \ell, N. \; \begin{array}{l} [\![t_1]\!](s) = \ell \wedge \\ \ell \neq \textbf{null} \wedge \\ [\![t_2]\!](s) = N \wedge \\ dom(h) = \{\ell\} \wedge \\ h(\ell) = N \end{array} \right\}
$$

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



## Semantics of separation logic assertions: $*$

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

$$
[\![P * Q]\!](s) \stackrel{def}{=} \left\{ h \in \mathit{Heap} \;\middle|\; \exists h_1, h_2. \; \begin{array}{l} h_1 \in [\![P]\!](s) \wedge \\ h_2 \in [\![Q]\!](s) \wedge \\ h = h_1 \uplus h_2 \end{array} \right\}
$$

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

## Semantics of separation logic assertions: *emp*

The empty heap assertion only holds for the empty heap:

$$[\![emp]\!](s) \stackrel{def}{=} \{h \in Heap \mid dom(h) = \emptyset\}$$

*emp* does not assert ownership of any location, so to capture ownership, $dom(h) = \emptyset$.



## 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



## 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:

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

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

$$\nvdash \{100 \mapsto 1\}\ [37] := 42\ \{100 \mapsto 1\}$$

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, $*$, 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:

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

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 $mod(C) \cap FV(R) = \emptyset$.



## Examples of 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:

$$\{100 \mapsto 1\}\ [37] := 42\ \{100 \mapsto 1\}$$

In particular, it would preserve the frame $37 \mapsto 1$:

$$\{100 \mapsto 1 * 37 \mapsto 1\}\ [37] := 42\ \{100 \mapsto 1 * 37 \mapsto 1\}$$

This triple definitely does not hold, since location 37 contains 42 in the terminal state.

## 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 states satisfies $37 \mapsto 42 * 37 \mapsto 1$:

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



## 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 $h_1 \uplus h_F$, $C$ terminates, then the terminal heap 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 heaps $h_F$.



## Formal semantics of separation logic triples

Written formally, the semantics is:

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

We then have the semantic version of the frame rule baked in:
If $\models \{P\}\ C\ \{Q\}$ and $mod(C) \cap FV(R) = \emptyset$, then
$\models \{P * R\}\ C\ \{Q * R\}$.



## Summary

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

Separation logic extends Hoare logic with notions of **ownership** and **separation** to control aliasing and reason about 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:

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

## For reference: failure of expressions

We can also allow failure in expressions:

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

$$\mathcal{E}[\![E_1 + E_2]\!](s) \stackrel{\text{def}}{=} \begin{cases} \text{if } \exists N_1, N_2.\ \mathcal{E}[\![E_1]\!](s) = N_1 \wedge \mathcal{E}[\![E_2]\!](s) = N_2 & , \quad N_1 + N_2 \\ \text{otherwise,} & \quad \lightning \end{cases}$$

$$\mathcal{E}[\![E_1 / E_2]\!](s) \stackrel{\text{def}}{=} \begin{cases} \text{if } \exists N_1, N_2.\ \mathcal{E}[\![E_1]\!](s) = N_1 \wedge \mathcal{E}[\![E_2]\!](s) = N_2 \wedge N_2 \neq 0 & , \quad N_1 / N_2 \\ \text{otherwise,} & \quad \lightning \end{cases}$$

$$\vdots$$

$$\mathcal{B}[\![-]\!] : \mathit{BExp} \times \mathit{Store} \to \{\lightning\} + \mathbb{B}$$

$$\vdots$$



## For reference: handling failures of expressions

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

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

$$\frac{\mathcal{B}[\![B]\!](s) = \lightning}{\langle \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2, (s,h)\rangle \Downarrow \lightning} \qquad \frac{\mathcal{B}[\![B]\!](s) = \lightning}{\langle \textbf{while } B \textbf{ do } C, (s,h)\rangle \Downarrow \lightning}$$

$$\frac{\mathcal{E}[\![E]\!](s) = \lightning}{\langle \textbf{dispose}(E), (s,h)\rangle \Downarrow \lightning}$$



## For reference: semantics with failure of expressions

The definitions we give work without modifications, because implicitly, by writing $N$ and $\ell$, we assume $N \neq \lightning$ and $\ell \neq \lightning$.

However, the separation logic rules have to be modified to prevent faulting of expressions (see next lecture).



# Hoare logic

**Lecture 6: Examples in separation logic**

**Jean Pichon-Pharabod**
University of Cambridge

CST Part II – 2017/18

## Introduction

In the previous lecture, we saw how reasoning about pointers in Hoare logic was problematic, which motivated introducing separation logic. We looked at the concepts separation logic is based on, the new assertions that embody them, and the semantics of assertions and partial correctness triples in separation logic.

In this lecture, we will

- introduce a syntactic proof system for separation logic;
- use it to verify example programs, thereby illustrating the power of separation logic.

The lecture will be focused on partial correctness.



## Separation logic

Separation logic inherits all the partial correctness rules from Hoare logic from the first lecture, and extends them with

- structural rules, including the frame rule;
- rules for each new heap-manipulating command.

As we saw last time, some of the rules that were admissible for plain Hoare logic, for example the rule of constancy, are no longer sound for separation logic.

We now want the rule of consequence to be able manipulate our extended assertion language, with our new assertions $P * Q$, $t_1 \mapsto t_2$, and $emp$, and not just first-order logic anymore.



# A proof system for separation logic

## The frame rule

The frame rule expresses that separation logic triples always preserve any assertion 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 $C$.

## Why the frame rule matters

The frame rule is the core of separation logic.
As we saw last time, it builds in modularity and compositionality.

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

It is so central to separation logic that its soundness is built in the definition of the semantics of separation logic triples, making it sound by construction.



## Other structural rules

Given the rules that we are going to consider for the heap-manipulating commands, we are going to need to include structural rules like the following:

$$\frac{\vdash \{P\}\ C\ \{Q\}}{\vdash \{\exists v.\, P\}\ C\ \{\exists v.\, Q\}}$$

$$\vdots$$

Rules like these were admissible in Hoare logic.



## 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:

$$\frac{}{\vdash \{E_1 \mapsto t\}\ [E_1] := E_2\ \{E_1 \mapsto E_2\}}$$

If expressions are allowed to fault, we need a more complex rule.



## 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:

$$\frac{}{\vdash \{E \mapsto u \wedge V = v\}\ V := [E]\ \{E[v/V] \mapsto u \wedge V = u\}}$$

Here, $u$ and $v$ are auxiliary variables, and $v$ is used to refer to the initial value of program variable $V$ in the postcondition.

## Allocation and deallocation

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

$$\frac{}{\vdash \{V = v\}\ V := \mathbf{alloc}(E_0, ..., E_n)\ \{V \mapsto E_0[v/V], ..., E_n[v/V]\}}$$

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

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



## Specification of swap

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:

$$\{X \mapsto n_1 * Y \mapsto n_2\}\ C_{swap}\ \{X \mapsto n_2 * Y \mapsto n_1\}$$



## Swap example

## Proof outline for swap

$\{X \mapsto n_1 * Y \mapsto n_2\}$
$A := [X];$
$\{(X \mapsto n_1 * Y \mapsto n_2) \wedge A = n_1\}$
$B := [Y];$
$\{(X \mapsto n_1 * Y \mapsto n_2) \wedge A = n_1 \wedge B = n_2\}$
$[X] := B;$
$\{(X \mapsto B * Y \mapsto n_2) \wedge A = n_1 \wedge B = n_2\}$
$[Y] := A;$
$\{(X \mapsto B * Y \mapsto A) \wedge A = n_1 \wedge B = n_2\}$
$\{X \mapsto n_2 * Y \mapsto n_1\}$

Justifying these individual steps is now considerably more involved than in Hoare logic.

## Detailed proof outline for the first triple of swap

$\{X \mapsto n_1 * Y \mapsto n_2\}$

$\{\exists a.\, ((X \mapsto n_1 * Y \mapsto n_2) \wedge A = a)\}$

$\{(X \mapsto n_1 * Y \mapsto n_2) \wedge A = a\}$

$\{(X \mapsto n_1 \wedge A = a) * Y \mapsto n_2\}$

$\{X \mapsto n_1 \wedge A = a\}$

$A := [X]$

$\{X[a/A] \mapsto n_1 \wedge A = n_1\}$

$\{X \mapsto n_1 \wedge A = n_1\}$

$\{(X \mapsto n_1 \wedge A = n_1) * Y \mapsto n_2\}$

$\{(X \mapsto n_1 * Y \mapsto n_2) \wedge A = n_1\}$

$\{\exists a.\, ((X \mapsto n_1 * Y \mapsto n_2) \wedge A = n_1)\}$

$\{(X \mapsto n_1 * Y \mapsto n_2) \wedge A = n_1\}$



## For reference: proof of the first triple of swap

Put another way:

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

$$\{X \mapsto n_1 \wedge A = a\}\ A := [X]\ \{X[a/A] \mapsto n_1 \wedge A = n_1\}$$

Then we existentially quantify the auxiliary variable $a$:

$$\{\exists a.\, X \mapsto n_1 \wedge A = a\}\ A := [X]\ \{\exists a.\, X[a/A] \mapsto n_1 \wedge A = n_1\}$$

Applying the rule of consequence, we obtain:

$$\{X \mapsto n_1\}\ A := [X]\ \{X \mapsto n_1 \wedge A = n_1\}$$

Since $A := [X]$ does not modify $Y$, we can frame on $Y \mapsto n_2$:

$$\{X \mapsto n_1 * Y \mapsto n_2\}\ A := [X]\ \{(X \mapsto n_1 \wedge A = n_1) * Y \mapsto n_2\}$$

Lastly, by the rule of consequence, we obtain:

$$\{X \mapsto n_1 * Y \mapsto n_2\}\ A := [X]\ \{(X \mapsto n_1 * Y \mapsto n_2) \wedge A = n_1\}$$



## Proof of the first triple of swap (continued)

We relied on many properties of our assertion logic.

For example, to justify the first application of consequence, we need to show that

$$\vdash P \Rightarrow \exists a.\, (P \wedge A = a)$$

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

$$\vdash ((X \mapsto n_1 \wedge A = n_1) * Y \mapsto n_2) \Rightarrow ((X \mapsto n_1 * Y \mapsto n_2) \wedge A = n_1)$$



## Properties of separation logic assertions

## Syntax of assertions in separation logic

We now have an extended language of assertions, with a new connective, the separating conjunction $*$:

$$\begin{array}{rcl} P, Q & ::= & \bot \mid \top \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q \\ & \mid & P * Q \mid \mathit{emp} \\ & \mid & \forall v.\, P \mid \exists v.\, P \mid t_1 = t_2 \mid p(t_1, ..., t_n) \quad n \geq 0 \end{array}$$

$\mapsto$ is a predicate symbol of arity 2.

This is not just usual first-order logic anymore: this is an instance of the classical first-order logic of bunched implication (which is related to linear logic).
We will also require inductive predicates later.

We will take an informal look at what kind of properties hold and do not hold in this logic. Using the semantics, we can prove the properties we need as we go.



## Properties of separating conjunction

Separating conjunction is a commutative and associative operator with *emp* as a neutral element (like $\wedge$ was with $\top$):

$$\vdash P * Q \Leftrightarrow Q * P$$
$$\vdash (P * Q) * R \Leftrightarrow P * (Q * R)$$
$$\vdash P * \mathit{emp} \Leftrightarrow P$$

Separating conjunction is monotone with respect to implication:

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

Separating conjunction distributes over disjunction:

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



## Properties of separating conjunction (continued)

Separating conjunction semi-distributes over conjunction (but not the other direction in general):

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

In classical separation logic, $\top$ is not a neutral element for the separating conjunction: we only have

$$\vdash P \Rightarrow P * \top$$

but not the other direction in general. This means that we cannot "forget" about allocated locations (this is where classical separation logic differs from intuitionistic separation logic): we have $\vdash P * Q \Rightarrow P * \top$, but not $\vdash P * Q \Rightarrow P$ in general. To actually get rid of $Q$, we have to deallocate the corresponding locations.



## Properties of pure assertions

An assertion is **pure** when it does not talk about the heap. Syntactically, this means it does not contain *emp* or $\mapsto$.

Separating conjunction and conjunction become more similar when they involve 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 |

## Axioms for the points-to assertion

**null** cannot point to anything:

$\vdash \forall t_1, t_2.\, t_1 \mapsto t_2 \Rightarrow (t_1 \mapsto t_2 * t_1 \neq \textbf{null})$

locations combined by $*$ are disjoint:

$\vdash \forall t_1, t_2, t_3, t_4.\, (t_1 \mapsto t_2 * t_3 \mapsto t_4) \Rightarrow (t_1 \mapsto t_2 * t_3 \mapsto t_4 * t_1 \neq t_3)$

$\vdots$

Assertions in separation logic are not freely duplicable in general: we do not have $\vdash P \Rightarrow P * P$ for all $P$. Therefore, in general, we need to repeat the assertions on the right-hand side of the implication to not "lose" them.



# 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 this, we will specify and verify a library for working with lists, implemented using null-terminated singly-linked lists, in separation logic.



## A list library implemented using singly-linked lists

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

We will use null-terminated singly-linked list, starting from some designated *HEAD* program variable that refers to the first element of the linked list.

For instance, we will represent the mathematical list $[12, 99, 37]$ as we did in the previous lecture:

## 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 $\mathit{list}(t, \alpha)$ that relates a mathematical list, $\alpha$, with its memory representation starting at location $t$ (here, $\alpha, \beta, \ldots$ are just terms, but we write them differently to clarify the fact that they refer to mathematical lists).

To define such a predicate formally, we need to extend the assertion logic to reason about inductively defined predicates. We probably also want to extend it to reason about mathematical lists directly rather than through encodings. We will elide these details.



## Representation predicates

We are going to define the $\mathit{list}(t, \alpha)$ predicate by induction on the list $\alpha$:

- The empty list [] is represented as a **null** pointer:

$$\mathit{list}(t, []) \stackrel{\mathit{def}}{=} t = \mathbf{null}$$

- The list $h :: \alpha$ (again, $h$ is just a term) is represented by a pointer to two consecutive heap cells that contain the head $h$ of the list and the location of the representation of the tail $\alpha$ of the list, respectively:

$$\mathit{list}(t, h :: \alpha) \stackrel{\mathit{def}}{=} \exists y.\ t \mapsto h * (t+1) \mapsto y * \mathit{list}(y, \alpha)$$

(recall that $t \mapsto h \Rightarrow (t \mapsto h * t \neq \mathbf{null})$)



## 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.

For example, assuming that we represent the mathematical list $\alpha$ at location $\mathit{HEAD}$, we can specify a push operation $C_{\mathit{push}}$ that pushes the value of program variable $X$ onto the list in terms of its behaviour on the abstract state of the list as follows:

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



## Representation predicates

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

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

$$\left\{ \begin{array}{l} \mathit{list}(\mathit{HEAD}, \alpha) \wedge \\ X = x \end{array} \right\}\ C_{\mathit{push}}\ \{\mathit{list}(\mathit{HEAD}, x :: \alpha)\}$$

$$\{\mathit{list}(\mathit{HEAD}, \alpha)\}\ \ C_{\mathit{pop}}\ \left\{ \begin{array}{l} \left( \begin{array}{l} \mathit{list}(\mathit{HEAD}, []) \wedge \\ \alpha = [] \wedge \mathit{ERR} = 1 \end{array} \right) \vee \\ \left( \begin{array}{r} \alpha = h :: \beta \wedge \\ \exists h, \beta.\ \ \mathit{list}(\mathit{HEAD}, \beta) \wedge \\ \mathit{RET} = h \wedge \mathit{ERR} = 0 \end{array} \right) \end{array} \right\}$$

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

$$\vdots$$

The $\mathit{emp}$ in the postcondition of $C_{\mathit{delete}}$ ensures that the locations of the precondition have been deallocated.

## Implementation of *push*

The *push* operation stores the *HEAD* pointer into a temporary variable $Y$ before allocating two consecutive locations for the new list element, storing the start-of-block location to *HEAD*:

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

We wish to prove that $C_{push}$ satisfies its intended specification:

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

(We could use $HEAD := \mathbf{alloc}(X, HEAD)$ instead.)



## Proof outline for *push*

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

$$\{list(HEAD, \alpha) \wedge X = x\}$$
$$Y := HEAD$$
$$\{list(Y, \alpha) \wedge X = x\}$$
$$HEAD := \mathbf{alloc}(X, Y)$$
$$\{(list(Y, \alpha) * HEAD \mapsto X, Y) \wedge X = x\}$$
$$\{list(HEAD, X :: \alpha) \wedge X = x\}$$
$$\{list(HEAD, x :: \alpha)\}$$

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



## For reference: detailed proof outline for the allocation

$$\{list(Y, \alpha) \wedge X = x\}$$
$$\{\exists z.\, (list(Y, \alpha) \wedge X = x) \wedge HEAD = z\}$$
$$\{(list(Y, \alpha) \wedge X = x) \wedge HEAD = z\}$$
$$\{(list(Y, \alpha) \wedge X = x) * HEAD = z\}$$
$$\{HEAD = z\}$$
$$HEAD := \mathbf{alloc}(X, Y)$$
$$\{HEAD \mapsto X[z/HEAD], Y[z/HEAD]\}$$
$$\{HEAD \mapsto X, Y\}$$
$$\{(list(Y, \alpha) \wedge X = x) * HEAD \mapsto X, Y\}$$
$$\{(list(Y, \alpha) * HEAD \mapsto X, Y) \wedge X = x)\}$$
$$\{\exists z.\, (list(Y, \alpha) * HEAD \mapsto X, Y) \wedge X = x)\}$$
$$\{(list(Y, \alpha) * HEAD \mapsto X, Y) \wedge X = x\}$$



## Implementation of *delete*

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

$$C_{delete} \equiv\ X := HEAD;$$
$$\quad \mathbf{while}\ X \neq \mathbf{null}\ \mathbf{do}$$
$$\quad\quad (Y := [X + 1]; \mathbf{dispose}(X); \mathbf{dispose}(X + 1); X := Y)$$

We wish to prove that $C_{delete}$ satisfies its intended specification:

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

For that, we need a suitable loop invariant.
To execute safely, $X$ effectively needs to point to **a** list (which is $\alpha$ only at the start).

## Proof outline for *delete*

We can pick the invariant that we own the rest of the list:

$\{list(HEAD, \alpha)\}$
$X := HEAD;$
$\{list(X, \alpha)\}$
$\{\exists \beta.\, list(X, \beta)\}$
**while** $X \neq$ **null do**
  $\{\exists \beta.\, list(X, \beta) \wedge X \neq \textbf{null}\}$
  $(Y := [X + 1]; \textbf{dispose}(X); \textbf{dispose}(X + 1); X := Y)$
  $\{\exists \beta.\, list(X, \beta)\}$
$\{\exists \beta.\, list(X, \beta) \wedge \neg(X \neq \textbf{null})\}$
$\{emp\}$

We need to complete the proof outline for the body of the loop.



## 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:

$\{\exists \beta.\, list(X, \beta) \wedge X \neq \textbf{null}\}$
$\{\exists h, y, \gamma.\, X \mapsto h, y * list(y, \gamma)\}$
$Y := [X + 1];$
$\{\exists h, \gamma.\, X \mapsto h, Y * list(Y, \gamma)\}$
$\textbf{dispose}(X); \textbf{dispose}(X + 1);$
$\{\exists \gamma.\, list(Y, \gamma)\}$
$X := Y$
$\{\exists \gamma.\, list(X, \gamma)\}$
$\{\exists \beta.\, list(X, \beta)\}$



## Classical separation logic and deallocation

Classical separation logic forces us to deallocate.

If we did not have the two deallocations in the body of the loop, we would have to do something with

$$X \mapsto h * X + 1 \mapsto Y$$

We can at best weaken that assertion to $\top$, but not fully eliminate it.

We could weaken our loop invariant to $\exists \beta.\, list(X, \beta) * \top$: the $\top$ would indicate the memory leak.



## Implementation of *max*

The *max* operation iterates over a non-empty list, computing its maximum element:

$C_{max} \equiv$
  $X := [HEAD + 1]; M := [HEAD];$
  **while** $X \neq$ **null do**
    $(E := [X]; (\textbf{if } E > M \textbf{ then } M := E \textbf{ else skip}); X := [X + 1])$

We wish to prove that $C_{max}$ satisfies its intended specification:

$\{list(HEAD, h :: \alpha)\}\ C_{max}\ \{list(HEAD, h :: \alpha) * M = max(h :: \alpha)\}$

For that, we need a suitable loop invariant.
The lists represented starting at *HEAD* and $X$ are not disjoint.

## Proof outline for *max*

We can define an auxiliary predicate $\mathit{plist}(t_1, \alpha, t_2)$ inductively:

$$\mathit{plist}(t_1, [], t_2) \stackrel{\mathit{def}}{=} (t_1 = t_2)$$

$$\mathit{plist}(t_1, h :: \alpha, t_2) \stackrel{\mathit{def}}{=} (\exists y.\, t_1 \mapsto h, y * \mathit{plist}(y, \alpha, t_2))$$

such that $\mathit{list}(t, \alpha +\!\!+ \beta) \Leftrightarrow \exists y.\, \mathit{plist}(t, \alpha, y) * \mathit{list}(y, \beta)$, and use it to express our invariant:

$\{\mathit{list}(\mathit{HEAD}, h :: \alpha)\}$

$X := [\mathit{HEAD} + 1]; M := [\mathit{HEAD}];$

$\{\mathit{plist}(\mathit{HEAD}, [h], X) * \mathit{list}(X, \alpha) * M = \mathit{max}([h])\}$

$\{\exists \beta, \gamma.\, h :: \alpha = \beta +\!\!+ \gamma * \mathit{plist}(\mathit{HEAD}, \beta, X) * \mathit{list}(X, \gamma) * M = \mathit{max}(\beta)\}$

**while** $X \neq$ **null do**

$(E := [X]; (\textbf{if } E > M \textbf{ then } M := E \textbf{ else skip}); X := [X + 1])$

$\{\mathit{list}(\mathit{HEAD}, h :: \alpha) * M = \mathit{max}(h :: \alpha)\}$



## Concurrency (not examinable)

## Summary of examples in separation logic

We can specify abstract data types using representation predicates which relate an abstract model of the state of the data structure with a concrete memory representation.

Justification of individual steps has to be made quite carefully given the unfamiliar interaction of connectives in separation logic, but proof outlines remain very readable.



## Concurrent composition

Imagine extending our $\mathrm{WHILE}_p$ language with a concurrent composition construct (also "parallel composition"), $C_1 || C_2$, which executes the two statements $C_1$ and $C_2$ concurrently.

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

For instance, $(X := 0 || X := 1); \mathit{print}(X)$ is allowed to print 0 or 1.

## Concurrency disciplines

Adding concurrency 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, and
- well-synchronised shared state.



# Disjoint concurrency

## Disjoint concurrency

Disjoint concurrency refers to multiple commands potentially executing concurrently, 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 assertions into two disjoint parts, $P_1$ and $P_2$, and give each parallel command ownership of one of them:

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

The third hypothesis ensures that $C_1$ does not modify any program variables used in the specification of $C_2$, the fourth hypothesis ensures the symmetric.

## Disjoint concurrency example

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

$$\{X \mapsto 3 * Y \mapsto 4\}$$

| | |
|---|---|
| $\{X \mapsto 3\}$ | $\{Y \mapsto 4\}$ |
| $A := [X]; [X] := A + 1$ | $B := [Y]; [Y] := B + 1$ |
| $\{X \mapsto 4\}$ | $\{Y \mapsto 5\}$ |

$$\{X \mapsto 4 * Y \mapsto 5\}$$



## Well-synchronised concurrency

## 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 $I$. We have the following rules:

$$\frac{FV(I) = \emptyset}{I \vdash \{emp\}\ \textbf{lock}\ \{I * locked\}} \qquad \frac{FV(I) = \emptyset}{I \vdash \{I * locked\}\ \textbf{unlock}\ \{emp\}}$$

The *locked* resource ensures the lock can only be unlocked 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 $X$ concurrently.

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 that location $X$ initially contains an even number, we wish to prove that the contents of location $X$ is still even after the two concurrent threads have terminated.

A non-synchronised interleaving would allow $X$ to end up being odd.



## 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:

$$I \;\equiv\; \exists n.\, x \mapsto 2 \times n$$

We have to use an indirection through $X = x$ because $I$ is not allowed to mention program variables.



## Well-synchronised shared state example

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

| | |
|---|---|
| $\{X = x \wedge emp\}$ | $\{X = x \wedge emp\}$ |
| **lock**; | **lock**; |
| $\{X = x \wedge I * locked\}$ | $\{X = x \wedge I * locked\}$ |
| $\{X = x \wedge (\exists n.\, x \mapsto 2 \times n) * locked\}$ | |
| $A := [X]; [X] := A + 1;$ | |
| $\{X = x \wedge (\exists n.\, x \mapsto 2 \times n + 1) * locked\}$ | $C := [X]; [X] := C + 2;$ |
| $B := [X]; [X] := B + 1;$ | |
| $\{X = x \wedge (\exists n.\, x \mapsto 2 \times n) * locked\}$ | |
| $\{X = x \wedge I * locked\}$ | $\{X = x \wedge I * locked\}$ |
| **unlock**; | **unlock**; |
| $\{X = x \wedge emp\}$ | $\{X = x \wedge emp\}$ |

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

We can temporarily violate the invariant when holding the lock.



## Summary of concurrent separation logic

Concurrent separation logic supports disjoint concurrency by splitting resources into disjoint parts and distributing them to non-interacting commands.

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

Concurrent separation logic can also be extended to support reasoning about general concurrency interference.

Papers of historical interest:

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

## Perspectives

- Verification of the seL4 microkernel assembly:
  `https://entropy2018.sciencesconf.org/data/myreen.pdf`
- The RustBelt project:
  `https://plv.mpi-sws.org/rustbelt/`
- The iGPS logic for relaxed memory concurrency:
  `http://plv.mpi-sws.org/igps/`
- The Iris higher-order concurrent separation logic framework, implemented and verified in a proof assistant:
  `http://iris-project.org/`
- Facebook's bug-finding Infer tool:
  `http://fbinfer.com/`



## Overall summary

We have seen that Hoare logic (separation logic, when we have pointers) enables specifying and reasoning about programs.

Reasoning remains close to the syntax, and captures the intuitions we have about why programs are correct.

It's all about **invariants**!*

*Or recursive function pre- and postconditions, which amount to the same thing.



## The heap assignment rule for when expressions can fault

$$\vdash \{E_1 \mapsto t_1 * E_2 = t_2\}\ [E_1] := E_2\ \{E_1 \mapsto E_2\}$$

It also requires that evaluating $E_2$ does not fault.
Exercise: Why is $E_1 = t_1$ not necessary in the precondition?