Hoare logic

Lecture 4: A verifier for Hoare logic

Jean Pichon-Pharabod
University of Cambridge

CST Part II – 2017/18

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.

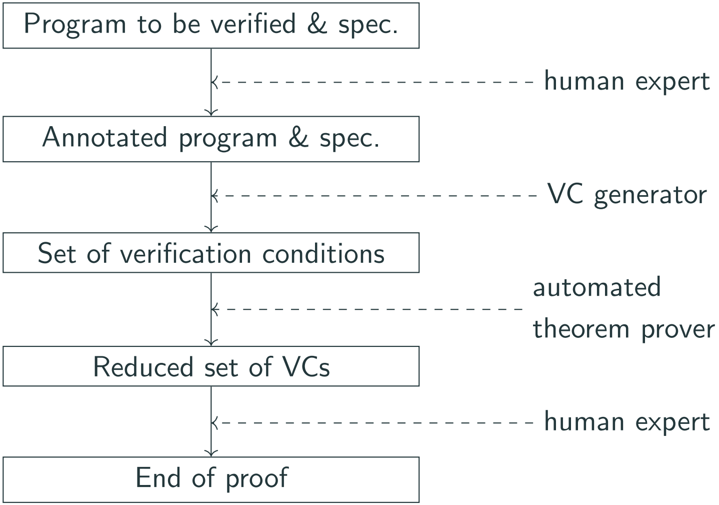
This does not mean that one cannot have procedures that will prove many useful statements.
For example, SMT solvers work quite well.

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):

```
$\{\top\}$
   $R := X;$
   $Q := 0;$
   **while** $Y \leq R$ **do**
      $(R := R - Y; Q := Q + 1)$
$\{X = R + Y \times Q \wedge R < Y\}$
```

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:

```
$R := X;$
$Q := 0;$
$\{R = X \wedge Q = 0\}$
**while** $Y \leq R$ **do** $\{X = R + Y \times Q\}$
   $(R := R - Y; Q := Q + 1)$
```

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:

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

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

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

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

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:

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

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)

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

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

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

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

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,

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

We will write

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

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

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

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

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

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

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

Example of VCs for conditionals

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

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

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

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

$$\boxed{\begin{aligned} VC(P,\ \mathcal{C}_1; \{R\}\ \mathcal{C}_2,\ Q) &\overset{def}{=} VC(P, \mathcal{C}_1, R) \cup VC(R, \mathcal{C}_2, Q) \\ VC(P,\ \mathcal{C}; V := E,\ Q) &\overset{def}{=} VC(P, \mathcal{C}, Q[E/V]) \end{aligned}}$$

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

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

This suggests defining

$$\boxed{\begin{array}{l} VC(P, \textbf{while}\ B\ \textbf{do}\ \{I\}\ \mathcal{C}, Q) \overset{def}{=} \\ \quad \{P \Rightarrow I\} \cup VC(I \wedge B, \mathcal{C}, I) \cup \{I \wedge \neg B \Rightarrow Q\} \end{array}}$$

Soundness of VCs for loops

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

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

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

Therefore, from the definition of $VC(P, \textbf{while } B \textbf{ do } \{I\}\ \mathcal{C}, Q)$, 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

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

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

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')

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:

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

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

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

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

There is no general finite formula for $\mathit{wlp}(\textbf{while } B \textbf{ do } C, Q)$ 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

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

(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:

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

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".