Hoare Logic and Model Checking

Kasper Svendsen
University of Cambridge

CST Part II – 2016/17

Acknowledgement: slides heavily based on previous versions by Mike Gordon and Alan Mycroft

Mechanised Program Verification

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

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

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

Mechanisation

Unfortunately, logicians have shown that it is impossible in principle to design a decision procedure to decide automatically the truth or falsehood of an arbitrary mathematical statement.

This does not mean that one cannot have procedures that will prove many useful theorems:

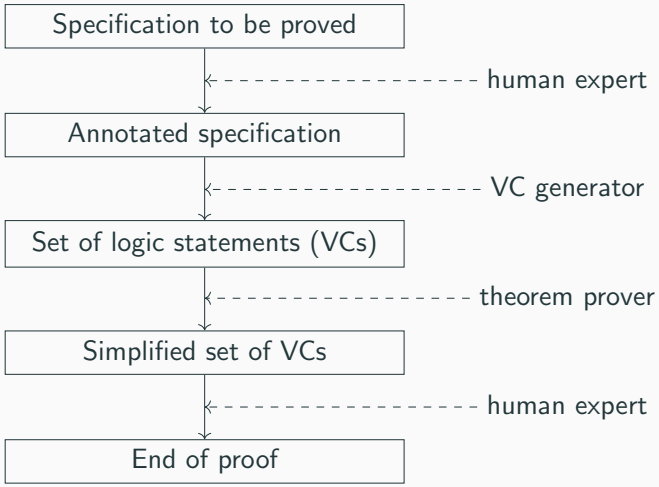
- the non-existence of a general decision procedure merely shows that one cannot hope to prove everything automatically
- in practice, it is quite possible to build a system that will mechanise the boring and routine aspects of verification

Mechanisation

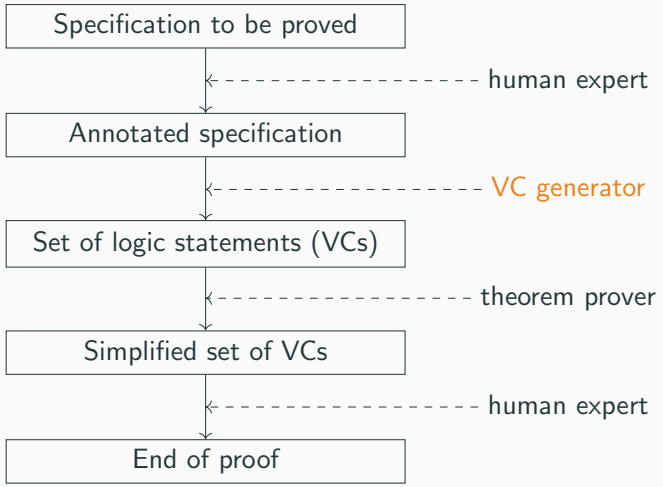
The standard approach to this will be described in the course

- ideas very old (JC King's 1969 CMU PhD, Stanford verifier in 1970s)
- used by program verifiers (e.g. Gypsy and SPARK verifier)
- provides a verification front end to different provers (see Why system)

Architecture of a verifier



Architecture of a verifier



VC generator

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

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

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

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

Using a verifier

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

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

Using a verifier

Verifiers are not a silver bullet!

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

Example

Before diving into the details, lets look at an example.

We will illustrate the process with the following example

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

Example

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

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

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

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

Example

Step 2 will generate the following four VCs for our example

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

Notice that these are statements of arithmetic; the constructs of our programming language have been 'compiled away'

Step 3 uses a standard theorem prover to automatically discharge as many VCs as possible and let the user prove the rest manually

Annotation of Commands

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

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

- before C2 in C1;C2 if C2 is not an assignment command
- after the word DO in WHILE commands

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

Backwards-reasoning proof rules

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

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

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

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

Annotations of Specifications

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

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

$$\begin{array}{l}\{X = n\} \\ \quad Y := 1; \longleftarrow l_1 \\ \quad \textbf{while}\ X = 0\ \textbf{do} \longleftarrow l_2 \\ \qquad (Y := Y * X; X := X - 1) \\ \{X = 0 \wedge Y = n!\}\end{array}$$

Generating VCs

Next we need to specify the VC generator

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

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

Backwards-reasoning proof rules

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

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

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

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

Justification of VCs

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

Formally,

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

This will be proven by induction on C

- we have to show the result holds for all primitive commands
- and that it holds for all compound commands C, assuming it holds for the constituent commands of C

VC for assignments

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

Example: The verification condition for

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

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

VC for assignments

To justify the VC generated for assignment we need to show

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

which holds by the backwards-reasoning assignment rule

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

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

VCs for conditionals

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

Example: The verification conditions for

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

are

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

VCs for conditionals

To justify the VC generated for assignment we need to show that

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

where

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

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

VCs for conditions

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

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

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

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

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

VCs for sequences

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

- have either been annotated with an intermediate assertion, or
- C_2 is an assignment

We define VC for each of these two cases

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

VCs for sequences

Example

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

VCs for sequences

To justify the VCs we have to prove that

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

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

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

VCs for loops

A properly annotated loop has the form

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

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

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

VCs for loops

To justify the VCs for loops we have to prove that

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

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

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

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

It follows by the backwards-reasoning rule for loops that

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

Summary

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

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

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

The soundness of the verifier used justified using a simple inductive argument and use many of the derived rules for backwards reasoning from the last lecture

Other uses for Hoare triples

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

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

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

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

Weakest preconditions

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

- if P and Q are assertions then P is 'weaker' than Q if $Q \Rightarrow P$
- thus, $\{P\}\ C\ \{Q\} \Leftrightarrow P \Rightarrow wlp(C, Q)$

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

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

Weakest preconditions

While the following property holds for loops

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

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

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

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

Program refinement

We have focused on proving programs meet specifications

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

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

For more: "Programming From Specifications" by Carroll Morgan

Conclusion

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

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

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