Hoare Logic and Verification Condition Generation

David Pereira José Proença Eduardo Tovar

FVOCA 2021/2022

Formal Verification of Critical Applications

CISTER – ISEP
Porto, Portugal

https://cister-labs.github.io/fvoca2122

Recap Hoare Logic

Recall the target language's syntax

The syntax of the language that we will be using is an extension of the basic Imperative Language with logical assertions, plus the concept of **Hoare Triple**.

$$
\begin{array}{lcll}
x & \in & \text{Identifiers} & \\
n & \in & \mathbb{Z} & \\
B & ::= & \text{true} \mid \text{false} \mid B \,\&\&\, B \mid B \,||\, B \mid !B \mid E < E \mid E = E & \text{(boolean-expr)} \\
E & ::= & n \mid x \mid E + E \mid E * E \mid E - E & \text{(int-expr)} \\
C & ::= & \text{skip} \mid C; C \mid x := E \mid \text{if } B \text{ then } C \text{ else } C \mid \text{while } B \text{ do } C & \text{(command)} \\
A & ::= & \text{true} \mid \text{false} \mid E < E \mid E = E \mid A \wedge A \mid A \vee A \mid \neg A \mid A \rightarrow A & \\
 & & \mid \forall x.A \mid \exists x.A & \text{(assertions)} \\
S & ::= & \{A\}\ C\ \{A\} & \text{(specification/Hoare Triple)}
\end{array}
$$

Hoare Triples - review

Core concept

The atomic concept that we will be using in order to reason about the partial correctness of programs is that known as *Hoare Triple*. It is a specification of the form

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

that reads as follows: for all states satisfying the *precondition* P, if the program C executes and terminates in a state satisfying the *postcondition* Q, then the triple is valid.

Design by Contract

These preconditions and postconditions, as well as other types of logical specifications that we will see ahead, are also known as **contracts**, that is, they bound the developer to implement code that satisfies the prescribed logical specifications to ensure partial correctness.

Hoare Logic

Validity of Hoare Triples

How to check the validity of Hoare Triples?

- the usual method for assuring the validity of Hoare Triples is to use a **sound** program-proof system.
- by sound one means that it should not allow to conclude specifications that are not valid.

Program-Proof System

A program-proof system is a set of inference rules that can be seen as fundamental laws about programs. In the next slides, we will present such rules.

The case of the skip command

The skip command does not change the state of the program. Hence, the precondition and postconditions must be the same.

$$\frac{}{\{P\}\ \text{skip}\ \{P\}}$$

The Program-Proof System Rules

The case of the assignment command

The assignment rule states that a postcondition Q can be granted for an assignment command if the condition that results from substituting E for x in Q holds as precondition.

$$\frac{}{\{Q[x \mapsto E]\}\ x := E\ \{Q\}}$$

Instances of the above axiom

- $\{Y = 2\}\ x := 2\ \{Y = X\}$
- $\{x + 1 = n + 1\}\ x := x + 1\ \{x = n + 1\}$
- $\{E = E\}\ x := E\ \{x = E\}$

The case of command sequence

When in the presence of a sequence of commands, we must prove that if the first command C_1 terminates in a postcondition R then, the second command C_2, satisfying R at start, if terminates, must do so in the prescribed postcondition Q.

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

An example of the sequence rule

A short example

By the assignment axiom we have:

(i) $\{X = x \wedge Y = y\}\ R := X\ \{R = x \wedge Y = y\}$

(ii) $\{R = x \wedge Y = y\}\ X := Y\ \{R = x \wedge X = y\}$

(iii) $\{R = x \wedge X = y\}\ Y := R\ \{Y = x \wedge X = y\}$

By (i) and (ii) and the sequence rule we obtain

$$(iv)\ \{X = x \wedge Y = y\}\ R := X; X := Y\ \{R = x \wedge Y = y\}$$

Now, by (iv) and (iii) we finish the deduction

$$\{X = x \wedge Y = y\}\ R := X; X := Y; Y := R\ \{Y = x \wedge X = y\}$$

The case of conditionals

In the case of conditionals, one must prove that independently of the value of the Boolean B that serves as a guarda for the conditional.

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

The case fo while loops

In the case of loops, we need to prove an invariant I (i.e., a condition that must hold at the entry and exit of the loop) throughout all the iterations of the loop (we don't know beforehand how many will be).

$$\frac{\{B \wedge I\}\ C_1\ \{I\}}{\{I\}\ \text{while } B \text{ do } C_1 \{I \wedge \neg B\}}$$

The case of logical consequence

This one is a special rule that deals only with logical assertions. It establishes a connection with first-order logic by means of side conditions that are assertions rather than specifications.

$$\frac{\{P_1\}\, C\, \{Q_1\}}{\{P\}\, C\, \{Q\}}$$

if $P \to P_1$ and $Q_1 \to Q$ hold (these are called side-conditions of the rule). Note also that this rule implies that Hoare logic is not meant to be used by itself; it must always be accompanied by some device for establishing the validity of side conditions, such as a decision procedure based on satisfiability, or an inference system for first-order logic.

A simple example using the logical consequence rule

The small derivation below shows the applicability of the logical consequence rule, followed by the assignment rule. The logical consequence rule generates two side effects which can be trivially proved, and enables the application of the assignment rule.

$$\dfrac{\dfrac{}{\{x+1>10\}\ x := x+1\ \{x>10\}}\ \text{(assignment rule)}}{\{x>10\}\ x := x+1\ \{x>10\}}\ \text{if } x>10 \rightarrow x+1>10 \text{ and } x>10 \rightarrow x>10$$

How to deal with Hoare Logic proofs

Two ways

- directly encoding the inference system in the logic of the proof tool and reasoning simultaneously with rules of both Hoare logic and first-order logic as required: reasoning starts with the former but switches to the latter logic when side conditions are introduced by the consequence rule.
- The alternative approach consists in two steps:
  - A **proof tree** is constructed for the desired specification, assuming that the side conditions generated by the consequence rule are valid
  - An external proof tool is used (such as a theorem prover or a proof assistant) to actually establish the validity of the side conditions
  - several proof trees can be created for the same specification (e.g., in the previous example we could weaken the precondition $x > 10$ to $x + 2 > 10$ instead of $x + 1 > 10$, meaning that we would obtain two different proof trees that would give the same conclusion.

On Verification Conditions Generation

From proof trees to algorithms

We have stated that a proof tree must be constructed to perform verification using Hoare Logic. When we close all the trees via the application of one of the rules that represents an axiom (e.g., skip rule or assignment rule), then we can state that a valid proof is obtained. Still, it is a manual process, which we would really like to avoid.

Verification Conditions Generation

The process of constructing a proof tree can, fortunately, be replaced by an algorithm. This algorithm, called **Verification Conditions Generator**, or **VCGen** for short, applies a particular strategy for producing verification conditions that correspond to the side conditions of a particular derivation

How do these VCGen work?

A VCGen algorithm takes as input a Hoare Triple $\{P\}\ C\ \{Q\}$ and returns a set of first-order logic proof obligations (recall that assertions/specifications are first-order logic formulas). As mentioned already, these proof obligations represent side conditions of the form $F_1 \rightarrow F_2$, but where program variables need to be scoped by a universal quantifier ($\forall x, F_1 \rightarrow F_2$ in this case) to ensure validity.

Example

Again looking at the example presented for the logic consequence rule of Hoare logic, the side condition $x > 10 \rightarrow x + 1 > 10$ in reality should be $\forall x, x > 10 \rightarrow x + 1 > 10$. That is, we prove that this side condition is a theorem!!!

Important: there are two possible sources of erros that may cause verification of a Hoare triple to fail: (1) program errors and (2) specification errors!

Alternative Formulation of Hoare Logic

Towards a new formulation

Desirable properties

- *subformula property:* the premises of a rule should not contain occurrences of assertions that do not appear in the rule's conclusion (otherwise we have to invent formulas).
- *unambiguity:* a unique rule should be applicable in a backward fashion for the goal at hand, so that the construction of the proof tree is syntax-directed (and thus algorithmic).

We start by addressing the second desirable property, i.e., syntax-directed rules!

Syntax directed Hoare logic

Undesired properties of original formulation

- the skip rule can only be applied if the precondition and post condition are the same;
- the assign rule can only be applied if the precondition results from the postcondition by performing the corresponding substitution;
- the while rule can only be applied if the precondition is an invariant of the loop, and the postcondition is the same invariant "strenghtened" with the negation of the loop condition.

Important:

the application of the above rules to goals with arbitrary preconditions and postconditions may very well require the application of the consequence rule! So this rule has to be embedded somehow in a new set of rules to be syntax-directed!!!

The case of the skip command

The skip command does not change the state of the program. Hence, the precondition and postconditions must be the same.

$$\frac{}{\{P\}\ \text{skip}\ \{P\}}$$

By incorporating the consequence rule, we obtain a new rule:

$$\frac{}{\{P\}\ \text{skip}\ \{Q\}}\ \text{if } P \to Q$$

The case of the assignment command

The assignment rule states that a postcondition Q can be granted for an assignment command if the condition that results from substituting E for x in Q holds as precondition.

$$\frac{}{\{Q[x \mapsto E]\}\ x := E\ \{Q\}}$$

By incorporating the consequence rule, we obtain a new rule:

$$\frac{}{\{P\}\ x := E\ \{Q\}}\ \text{if } P \to Q[x \mapsto E]$$

The case fo while loops

In the case of loops, we need to prove an invariant I (i.e., a condition that must hold at the entry and exit of the loop) throughout all the iterations of the loop (we don't know beforehand how many will be).

$$\frac{\{B \wedge I\}\, C_1\, \{I\}}{\{I\}\, \text{while } B \text{ do } C_1 \{I \wedge \neg B\}}$$

By incorporating the consequence rule, we obtain a new rule:

$$\frac{\{B \wedge I\}\, C\, \{I\}}{\{P\}\, \text{while } B \text{ do } C \{Q\}} \text{ if } P \to I \text{ and } I \wedge \neg B \to Q$$

Almost there to define a VCGen

Goal directed but no sub-formula property

We can definitely state that the new rules provide Hoare logic with a syntax-directed approach: it consists of **exactly one rule for each program construct**, and moreover, for a given specification $\{P\}\ C\ \{Q\}$, the rule matching the program C can always be applied (granted that side conditions are met!), since the precondition and postcondition are now arbitrary.

No sub-formula property

- the while rule lost the property since the invariant assertion no longer occurs in the conclusion
- the rule for the sequence construct does not enjoy the property either, since the intermediate assertion R does not occur in the conclusion

Program Annotations

The role of program annotations

Restoring the subformula property

One way to restore the subformula property in our current system for Hoare logic is precisely to introduce **human-provided annotations** in the programs. An annotated program is a **program with assertions embedded within it**.

Extending the language of commands

$$C \ ::= \ \text{skip} \ \mid \ C; \{A\}\, C \ \mid \ x := E \ \mid \ \text{if } B \text{ then } C \text{ else } C \ \mid \ \text{while } B \text{ do } \{A\}\, C$$

Some examples

Example 1

For instance, the command

$$\text{while } B \text{ do } \{I\}\ C$$

denotes a loop with condition B and user-provided invariant I.

Example 2

For instance, the command $C_1;\ \{A\}\ C_2$ has the same meaning ot $C; C$ but the assertion A must be true when the execution of C_1 terminates.

Updating the sequence rule

The new case of command sequence

When in the presence of a sequence of commands, we must prove that if the first command C_1 terminates in a postcondition R then, the second command C_2, satisfying R at start, if terminates, must do so in the prescribed postcondition Q.

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

Example of annotated Fibonnaci program

a) Program $C_{\texttt{Fib}}$

$x := 1\,;$
$y := 0\,;$
$i := 1\,;$
while $i < n$ **do**
{
  $aux := y\,;$
  $y := x\,;$
  $x := x + aux\,;$
  $i := i + 1$
}

b) Annotated program $C^{\mathrm{A}}_{\texttt{Fib}}$

$x := 1\,;$
$\{x == 1\}$
$y := 0\,;$
$\{x == 1 \;\&\&\; y == 0\}$
$i := 1\,;$
$\{x == 1 \;\&\&\; y == 0 \;\&\&\; i == 1\}$
while $i < n$ **do** $\{i \leq n \;\&\&\; x == \texttt{Fib}(i) \;\&\&\; y == \texttt{Fib}(i-1)\}$
{
  $aux := y\,;$
$\{i \leq n \;\&\&\; x == \texttt{Fib}(i) \;\&\&\; aux == \texttt{Fib}(i-1)\}$
  $y := x\,;$
$\{i < n \;\&\&\; x == \texttt{Fib}(i) \;\&\&\; y == \texttt{Fib}(i) \;\&\&\; aux == \texttt{Fib}(i-1)\}$
  $x := x + aux\,;$
$\{i \leq n \;\&\&\; x == \texttt{Fib}(i) + \texttt{Fib}(i-1) \;\&\&\; y == \texttt{Fib}(i)\}$
  $i := i + 1\,;$
}

Extending the annotation to while loops

The case fo while loops

The new rule for while loops now considers the invariant as an annotation:

$$\frac{\{B \wedge I\}\ C\ \{I\}}{\{P\}\ \text{while}\ B\ \text{do}\ \{I\}\ C\ \{Q\}}\ \text{if}\ P \to I\ \text{and}\ I \wedge \neg B \to Q$$

Propagation of annotations in code

Annotating a program is definitely a tedious and error prone task. Nevertheless, there are two ways to achieve it: vai forward propagation or via backward propagation. Lets understand these via examples. Consider the following code:

$$
\begin{aligned}
&\{x == 5 \;\&\&\; y == 10\} \\
&\quad aux := y\,; \\
&\quad y := x\,; \\
&\quad x := x + aux\,; \\
&\{x > 10 \;\&\&\; y == 5\}
\end{aligned}
$$

Results from propagation

$$
\begin{array}{l}
\{x == 5 \;\&\&\; y == 10\} \\
\quad aux := y\,; \\
\{x + aux > 10 \;\&\&\; x == 5\} \\
\quad y := x\,; \\
\{x + aux > 10 \;\&\&\; y == 5\} \\
\quad x := x + aux\,; \\
\{x > 10 \;\&\&\; y == 5\}
\end{array}
$$

a) Backward propagation of the postcondition

$$
\begin{array}{l}
\{x == 5 \;\&\&\; y == 10\} \\
\quad aux := y\,; \\
\{x == 5 \;\&\&\; y == 10 \;\&\&\; aux == 10\} \\
\quad y := x\,; \\
\{x == 5 \;\&\&\; y == 5 \;\&\&\; aux == 10\} \\
\quad x := x + aux\,; \\
\{x > 10 \;\&\&\; y == 5\}
\end{array}
$$

b) Forward propagation of the precondition

Weakest Preconditions

Weakest Preconditions

A new look at back propagation of annotations

- we have seen that back propagating annotations from postconditions works
- it can be realised as an algorithm
- such algorithm generates the so-called **weakest preconditions** of an annotated code starting from its postcondition, that is, the weakes conditions that ensure that a program C satisfies the postcondition Q if it terminates, that is

$$\{\mathrm{wprec}(C, Q)\}\ C\ \{Q\}$$

- if we are able to prove that a precondition $p \rightarrow \mathrm{wprec}(C, Q)$ holds, then we can use this weakest precondition generation algorithm to help building the proof tree!

Algorithm for weakest precondition generation

$$
\begin{aligned}
wprec(\text{skip}, Q) &= Q \\
wprec(x := E, Q) &= Q[x \mapsto E] \\
wprec(C_1; C_2, Q) &= wprec(C_1, wprec(C_2, Q)) \\
wprec(\text{if } B \text{ then } C_1 \text{ else } C_2, Q) &= (B \to wprec(C_1, Q)) \wedge (\neg B \to wprec(C_2, Q)) \\
wprec(\text{while } B \text{ do } \{I\}\ C, Q) &= I
\end{aligned}
$$

Algorithm for generating Verification Conditions

$$
\begin{aligned}
VC(\{P\}\,\mathsf{skip}\{Q\}) &= \{P \to Q\} \\
VC(\{P\}\,x := E\,\{Q\}) &= \{P \to Q[x \mapsto E]\} \\
VC(\{P\}\,C_1; C_2\,\{Q\}) &= VC(\{P\}\,C_1\,\{wprec(C_2, Q)\}) \\
&\quad \cup \\
&\quad VC(\{wprec(C_2, Q)\}\,C_2\,\{Q\})
\end{aligned}
$$

Algorithm for generating Verification Conditions

$$
\begin{aligned}
VC(\{P\}\ \text{if}\ B\ \text{then}\ C_1\ \text{else}\ C_2\ \{Q\}) &= VC(\{P \wedge B\}\ C_1\{Q\}) \\
&\quad \cup \\
&\quad VC(\{P \wedge \neg B\}\ C_2\ \{Q\}) \\
VC(\{P\}\ \text{while}\ B\ \text{do}\ \{I\}\ C\ \{Q\}) &= \{P \rightarrow I, I \wedge \neg B \rightarrow Q\} \\
&\quad \cup \\
&\quad VC(\{P \wedge \neg B\}\ C\ \{Q\})
\end{aligned}
$$

Improved Verification Condition Generation

$$
\begin{aligned}
VC(\mathit{skip}, Q) &= \emptyset \\
VC(x := e, Q) &= \emptyset \\
VC(C_1; C_2, Q) &= VC(C_1, \mathit{wprec}(C_2, Q)) \cup VC(C_2, Q) \\
VC(\text{if } B \text{ then } C_1 \text{ else } C_2, Q) &= VC(C_1, Q) \cup VC(C_2, Q) \\
VC(\text{while } B \text{ do } \{I\}\ C) &= \{(I \wedge B) \rightarrow \mathit{wprec}(C, I)\} \cup VC(C, I) \\
&\quad\ \{(I \wedge \neg B) \rightarrow Q\}
\end{aligned}
$$

$$
VCG(\{P\}\ C\ \{Q\}) = \{P \rightarrow \mathit{wprec}(C, Q)\} \cup VC(C, Q)
$$