COL750: Foundations of Automatic Verification (Jul-Dec 2024)

(Hoare Logic)

Kumar Madhukar

madhukar@cse.iitd.ac.in

Nov. 7th

Reasoning about code

- Assigning meanings to programs, Robert W. Floyd, 1967
- An Axiomatic Basis for Computer Programming, C. A. R. Hoare, 1969

A simple language

$$S ::= \quad x = E \mid S_1; S_2 \mid \mathit{if}\ (B)\ \mathit{then}\ \{S_1\}\ \mathit{else}\ \{S_2\} \mid \mathit{while}\ (B)\ \{S\}$$

$$B ::= \quad \mathit{true} \mid \mathit{false} \mid (\mathit{not}\ B) \mid (B_1\ \mathit{and}\ B_2) \mid (B_1\ \mathit{or}\ B_2) \mid (E_1 < E_2)$$

$$E ::= \quad n \mid x \mid (-E) \mid (E_1 + E_2) \mid (E_1 - E_2) \mid (E_1 * E_2)$$

where n denotes an integer, and x denotes a variable

Forward reasoning

```
x = 17

y = 42

z = x + y
```

Forward reasoning

$\{true\}$

```
x = 17
```

$\{x = 17\}$

```
y = 42
```

$\{x = 17 \land y = 42\}$

```
z = x + y
```

$\{x = 17 \land y = 42 \land z = 59\}$

Forward reasoning

$\{true\}$

```
x = 17
```

$\{x = 17\}$

```
y = 42
```

$\{x = 17 \land y = 42\}$

```
z = x + y
```

$\{x = 17 \land y = 42 \land z = 59\}$

- the assertions may accumulate a lot of irrelevant facts because we do not know what will actually be useful for proving the property

Backward reasoning

```
x = y

x = x + 1
```

$\{x > 0\}$

Backward reasoning

```
x = y
```

$\{x + 1 > 0\}$

```
x = x + 1
```

$\{x > 0\}$

Backward reasoning

$\{y + 1 > 0\}$

```
x = y
```

$\{x + 1 > 0\}$

```
x = x + 1
```

$\{x > 0\}$

Backward reasoning

$\{y + 1 > 0\}$
```
x = y
```
$\{x + 1 > 0\}$
```
x = x + 1
```
$\{x > 0\}$

- $(y + 1 > 0)$ at the beginning of the execution ensures that $(x > 0)$ holds at the end
- other *preconditions* also guarantee that the *postcondition* holds (e.g. $y > 50$ or $y > 3$)
- but $(y > -1)$ is the weakest precondition

Hoare triples

$$\{P\} \qquad S \qquad \{Q\}$$

Hoare triples

$\{P\}$	S	$\{Q\}$
precondition	*code*	*postcondition*

Hoare triples

$$\{P\} \qquad S \qquad \{Q\}$$

precondition *code* *postcondition*

- if P holds true, and S is executed, and Q is guaranteed to be true afterwards, then the Hoare triple $\{P\}\ S\ \{Q\}$ is said to be valid
- $\{x \neq 0\}\ \ y = x * x\ \ \{y > 0\}$ is a valid Hoare triple
- $\{x \geq 0\}\ \ y = 2 * x\ \ \{y > 0\}$ is an invalid Hoare triple

Partial and Total Correctness

- what if the code S does not terminate!
- $\{P\}\ S\ \{Q\}$ is valid under partial correctness if from all states in P, when S is executed, if S terminates then the resulting state will necessarily be in Q
- $\{P\}\ S\ \{Q\}$ is valid under total correctness if from all states in P, when S is executed, S is guaranteed to terminate and the resulting state will necessarily be in Q
- we will ignore the question of termination, and will restrict ourselves to partial correctness

Our agenda

is to prove correctness of programs, given their specification

```
y = 1;
z = 0;

while(z != x)
  z = z + 1;
  y = y * z;
```

we would like to prove that this implementation is partially correct wrt its specification (that the program computes the factorial of x and stores it in y)

Our agenda

is to prove correctness of programs, given their specification

```
y = 1;
z = 0;

while(z != x)
  z = z + 1;
  y = y * z;
```

$\{true\}$	$y = 1$	$\{y = 1\}$
$\{y = 1\}$	$z = 0$	$\{y = 1 \ \wedge \ z = 0\}$
$\{y = 1\}$	$z = 0$	$\{y = z!\}$

$\{y = z!\}$	$while(..)\{...\}$	$\{y = z! \ \wedge \ \neg(z \neq x)\}$
$\{y = z!\}$	$while(..)\{...\}$	$\{y = x!\}$

we would like to prove that this implementation is partially correct wrt its specification (that the program computes the factorial of x and stores it in y)

Strongest postcondition

$\mathtt{sp}(S, P)$ is the strongest Q such that $\{P\}\ S\ \{Q\}$ is valid

this means that if $\{P\}\ S\ \{Q\}$ is valid, $\mathtt{sp}(S, P) \Rightarrow Q$

Strongest postcondition

$\texttt{sp}(S, P)$ is the strongest Q such that $\{P\}\ S\ \{Q\}$ is valid

this means that if $\{P\}\ S\ \{Q\}$ is valid, $\texttt{sp}(S, P) \Rightarrow Q$

$$\texttt{sp}(x := E, P) \quad = \quad \exists x'.\ [x'/x]P \wedge x = [x'/x]E$$

$$\texttt{sp}(S_1; S_2,\ P) \quad = \quad \texttt{sp}(S_2,\ \texttt{sp}(S_1, P))$$

$$\texttt{sp}(\mathit{if}\,(B)\ \mathit{then}\ S_1\ \mathit{else}\ S_2, P) \quad = \quad \texttt{sp}(S_1, P \wedge B) \quad \vee \quad \texttt{sp}(S_2, P \wedge \neg B)$$

What about the loop?

the following holds, but doesn't help!

$$\mathtt{sp}(\mathit{while}(B)\ \{S\},\ P) \quad = \quad \mathtt{sp}(\mathit{while}(B)\ \{S\},\ \ \mathtt{sp}(S, P \wedge B)) \quad \vee \quad (P \wedge \neg B)$$

Weakest (liberal) precondition

$\texttt{wlp}(S, Q)$ is the weakest predicate P such that $\{P\}\ S\ \{Q\}$ is valid (for partial correctness)

$\texttt{wp}(S, Q)$ is the weakest predicate P such that $\{P\}\ S\ \{Q\}$ is valid (for total correctness)

this means that if $\{P\}\ S\ \{Q\}$ is valid, $P \Rightarrow \texttt{wlp}(S, Q)$

Weakest (liberal) precondition

$\texttt{wlp}(S, Q)$ is the weakest predicate P such that $\{P\}\ S\ \{Q\}$ is valid (for partial correctness)

$\texttt{wp}(S, Q)$ is the weakest predicate P such that $\{P\}\ S\ \{Q\}$ is valid (for total correctness)

this means that if $\{P\}\ S\ \{Q\}$ is valid, $P \Rightarrow \texttt{wlp}(S, Q)$

$$\texttt{wlp}(x := E, Q) \quad = \quad Q[E/x]$$

$$\texttt{wlp}(S_1; S_2,\ Q) \quad = \quad \texttt{wlp}(S_1,\ \texttt{wlp}(S_2, Q))$$

$$\texttt{wlp}(\mathit{if}(B)\ \mathit{then}\ S_1\ \mathit{else}\ S_2, Q) \quad = \quad (B \Rightarrow \texttt{wlp}(S_1, Q)) \quad \wedge \quad (\neg B \Rightarrow \texttt{wlp}(S_2, Q))$$

$$\texttt{wlp}(\mathit{if}(B)\ \mathit{then}\ S_1\ \mathit{else}\ S_2, Q) \quad = \quad (B \wedge \texttt{wlp}(S_1, Q)) \quad \vee \quad (\neg B \wedge \texttt{wlp}(S_2, Q))$$

What about the loop?

the following holds, but doesn't help!

$$\texttt{wlp}(\mathit{while}(B)\ \{S\},\ Q) \quad = \quad \mathit{if} \quad B \quad \mathit{then}\ \texttt{wlp}(S,\ \ \texttt{wlp}(\mathit{while}(B)\ \{S\},\ \ Q)\ ) \quad \mathit{else} \quad Q$$

sp vs. wlp

- computing sp is like symbolically executing a program
- computing wlp is like attempting a backward proof
- sp may make it possible to simplify the current state, and may also help resolve branches
- wlp focuses on relevant facts

Proof rules for partial correctness

$$\frac{\{\phi\}\ \ S_1\ \ \{\eta\} \qquad \{\eta\}\ \ S_2\ \ \{\psi\}}{\{\phi\}\ \ S_1; S_2\ \ \{\psi\}}\ \text{composition}$$

$$\frac{}{\{\psi\}[E/x]\quad x := E\quad \{\psi\}}\ \text{assignment}$$

$$\frac{\{\phi \wedge B\}\ \ S_1\ \ \{\psi\} \qquad \{\phi \wedge \neg B\}\ \ S_2\ \ \{\psi\}}{\{\phi\}\quad if\,(B)\ then\ S_1\ else\ S_2\quad \{\psi\}}\ \text{if} - \text{then} - \text{else}$$

$$\frac{\{\psi \wedge B\}\ \ S\ \ \{\psi\}}{\{\psi\}\quad while(B)\ \{S_1\}\quad \{\psi \wedge \neg B\}}\ \text{partial} - \text{while}$$

$$\frac{\phi' \Rightarrow \phi \qquad \{\phi\}\ \ S\ \ \{\psi\} \qquad \psi \Rightarrow \psi'}{\{\phi'\}\ \ S\ \ \{\psi'\}}\ \text{implied}$$

$$\frac{}{\{B \Rightarrow \psi\}\quad assume(B)\quad \{\psi\}}\ \text{assume} \qquad\qquad \frac{}{\{\psi\}\quad assume(B)\quad \{\psi \wedge B\}}\ \text{assume}$$

Examples

for the program P, below, suppose we would like to prove that $\{\top\}\ \ P\ \ \{y = x + 1\}$

```
a = x + 1;
if (a - 1 == 0)
    y = 1;
else
    y = a;
```

Example

in order to get $\{y = x + 1\}$ at the end, we must get $\{y = x + 1\}$ at the end of both the conditional branches, so that we can apply the if-then-else proof rule

```
a = x + 1;
if (a - 1 == 0)
    y = 1;
```
$\{y = x + 1\}$
```
else
    y = a;
```
$\{y = x + 1\}$

$\{y = x + 1\}$ $\quad$ $\text{if} - \text{then} - \text{else}$

Example

in order to get $\{y = x + 1\}$ at the end of both the conditional branches, we need to use the assignment rule in both the branches

```
a = x + 1;
if (a - 1 == 0)
```

$\{1 = x + 1\}$

```
     y = 1;
```

$\{y = x + 1\}$ $\quad$ assignment

```
else
```

$\{a = x + 1\}$

```
     y = a;
```

$\{y = x + 1\}$ $\quad$ assignment

$\{y = x + 1\}$ $\quad$ if – then – else

Example

we can now compute the precondition which gives us the desired postconditions at the beginning of both the branches

```
a = x + 1;
```
$\{(a - 1 = 0 \Rightarrow 1 = x + 1) \ \wedge \ (\neg(a - 1 = 0) \Rightarrow a = x + 1)\}$
```
if (a - 1 == 0)
```
$\{1 = x + 1\}$ assume
```
     y = 1;
```
$\{y = x + 1\}$ assignment
```
else
```
$\{a = x + 1\}$ assume
```
     y = a;
```
$\{y = x + 1\}$ assignment

$\{y = x + 1\}$ if – then – else

Example

the condition before 'if' must come from the assignment

$\{(x+1-1=0 \Rightarrow 1=x+1) \ \wedge \ (\neg(x+1-1=0) \Rightarrow x+1=x+1)\}$

```
a = x + 1;
```

$\{(a-1=0 \Rightarrow 1=x+1) \ \wedge \ (\neg(a-1=0) \Rightarrow a=x+1)\}$ — assignment

```
if (a - 1 == 0)
```

$\{1=x+1\}$ — assume

```
    y = 1;
```

$\{y=x+1\}$ — assignment

```
else
```

$\{a=x+1\}$ — assume

```
    y = a;
```

$\{y=x+1\}$ — assignment

$\{y=x+1\}$ — if − then − else

Example

the precondition that we got is a valid statement (is same as $\top$)

$\{\top\}$

$\{(x+1-1=0 \Rightarrow 1=x+1) \;\wedge\; (\neg(x+1-1=0) \Rightarrow x+1=x+1)\}$ implied

```
a = x + 1;
```

$\{(a-1=0 \Rightarrow 1=x+1) \;\wedge\; (\neg(a-1=0) \Rightarrow a=x+1)\}$ assignment

```
if (a - 1 == 0)
```

$\{1=x+1\}$ assume

```
    y = 1;
```

$\{y=x+1\}$ assignment

```
else
```

$\{a=x+1\}$ assume

```
    y = a;
```

$\{y=x+1\}$ assignment

$\{y=x+1\}$ if − then − else

Revisiting the factorial example

$\{\top\}$
$\{1 = 0!\}$ implied

```
y = 1;
```

$\{y = 0!\}$ assignment

```
z = 0;
```

$\{y = z!\}$ assignment

```
while(z != x)
```

$\{y = z! \wedge z \neq x\}$ assume
$\{y.(z+1) = (z+1)!\}$ implied

```
    z = z + 1;
```

$\{y.z = z!\}$ assignment

```
    y = y * z;
```

$\{y = z!\}$ assignment

$\{y = z! \wedge \neg(z \neq x)\}$ partial − while
$\{y = x!\}$ implied

Thank you!