CS245: Program Verification

David Duan

August 2, 2018

Contents

1 Introduction

1.1 Hoare Triples

- A **Hoare triple** consists of a **precondition** $(\!|\ P\ |\!)$, a program C, and a **postcondition** $(\!|\ Q\ |\!)$.
- The Hoare triple $(\!|\ P\ |\!)\ C\ (\!|\ Q\ |\!)$ claims that *if the state of program C before execution satisfies P, then the ending state of C after execution will satisfy Q.*
- A Hoare triple with C as the second component is called a **specification** of the program C.

1.2 Partial and Total Correctness

- A triple $(\!|\ P\ |\!)\ C\ (\!|\ Q\ |\!)$ is satisfied under **partial correctness**, denoted $\models_{\text{par}} (\!|\ P\ |\!)\ C\ (\!|\ Q\ |\!)$, if and only if for every state s_1 that satisfies condition P, if execution of C starting from state s_1 terminates in a state s_2, then state s_2 satisfies condition Q.
- A triple $(\!|\ P\ |\!)\ C\ (\!|\ Q\ |\!)$ is satisfied under **total correctness**, denoted $\models_{\text{tot}} (\!|\ P\ |\!)\ C\ (\!|\ Q\ |\!)$, if and only if it satisfies partial correctness and the program **terminates**, i.e. for every state s_1 that satisfies condition P, execution of C stating from state s_1 terminates in a state s_2, and s_2 satisfies condition Q.

Remarks

1. **Specification is not behavior**: A Hoare triple specifies neither a unique program nor a unique behavior.
2. **Logical variables**: additional variables that help formulating pre- and postconditions that do not appear in the program.

Examples

- The Hoare triples $(\!|\ (x_0 = x)\ |\!)$ `x = 3;` $(\!|\ (x = 3)\ |\!)$ and $(\!|\ (x = x_0)\ |\!)$ `x = 3;` $(\!|\ (x = 3)\ |\!)$ are satisfied under partial correctness since after the assignment statement the claim $x = 3$ holds true.
- The Hoare triple $(\!|\ (x_0 = x)\ |\!)$ `x = 2;` $(\!|\ (x = 3)\ |\!)$ is not satisfied under partial correctness.

1.3 Proving Partial Correctness

- A proof of partial correctness has one or more conditions before and after each code statement.
- Each statement makes a Hoare triple with the preceding and following conditions.
- Each triple has a justification next to its postcondition that explains its correctness.

2 Assignment

Theorem 2.1: Assignment

$$\frac{}{\llparenthesis\ Q[E/x]\ \rrparenthesis\ (x = E)\ \llparenthesis\ Q\ \rrparenthesis} \quad \text{(assignment)}$$

Example

To determine the precondition, we need to identify Q and E then perform substitution, replacing every instance of the assigned variable by its assigned value in the postcondition.

$$\frac{}{\llparenthesis\ ?\ \rrparenthesis\ (x = y + 1)\ \llparenthesis\ (x = 7)\ \rrparenthesis}$$

- $Q : (x = 7)$
- $E : y + 1$
- $Q[E/x] \implies (x = 7)[x \mapsto y + 1] \implies$ Precondition is $((y + 1) = 7)$

Intuition

- $Q(x)$ will hold after assignment the value of E to x if Q was true of that value beforehand.

Remark

- The conditions P and Q are well-formed formulas thus must be fully parenthesized.
- Always work bottom-up from the postcondition when dealing with assignment statements.
- Do not simplify $Q[E/x]$ in any way and do not switch sides of the (in)equality.

3 Implication and Composition

3.1 Implication

Theorem 3.1: Implication: Precondition Strengthening

$$\frac{P \to P' \qquad \llparenthesis\, P' \,\rrparenthesis\; C \;\llparenthesis\, Q \,\rrparenthesis}{\llparenthesis\, P \,\rrparenthesis\; C \;\llparenthesis\, Q \,\rrparenthesis} \quad \text{(implied)}$$

Theorem 3.2: Implication: Postcondition Weakening

$$\frac{\llparenthesis\, P \,\rrparenthesis\; C \;\llparenthesis\, Q' \,\rrparenthesis \qquad Q' \to Q}{\llparenthesis\, P \,\rrparenthesis\; C \;\llparenthesis\, Q \,\rrparenthesis} \quad \text{(implied)}$$

Example: Precondition Strengthening

$\llparenthesis\, (y = 6) \,\rrparenthesis$	
$\llparenthesis\, ((y + 1) = 7) \,\rrparenthesis$	implied
`x = y + 1;`	
$\llparenthesis\, (x = 7) \,\rrparenthesis$	assignment

Example: Postcondition Weakening

$\llparenthesis\, (y < 0) \,\rrparenthesis$	
`x = y;`	
$\llparenthesis\, (x < 0) \,\rrparenthesis$	assignment
$\llparenthesis\, (x < 5) \,\rrparenthesis$	implied

Remark

- Do not simplify the assertions in the annotated program.
- Do the simplification in the separate justification for implications.

3.2 Composition

Theorem 3.3: Composition (implicit)

$$\frac{\llparenthesis\, P \,\rrparenthesis\; C_1 \;\llparenthesis\, Q \,\rrparenthesis \qquad \llparenthesis\, Q \,\rrparenthesis\; C_2 \;\llparenthesis\, R \,\rrparenthesis}{\llparenthesis\, P \,\rrparenthesis\; C_1; C_2 \;\llparenthesis\, R \,\rrparenthesis} \quad \text{(composition)}$$

Remark

- Q is called a **midcondition** that help us prove $\llparenthesis\, P \,\rrparenthesis\; C_1; C_2 \;\llparenthesis\, R \,\rrparenthesis$.
- The composition rule is implicit since any program that has more than 1 line requires it.

4 Conditionals

4.1 If-Then-Else

Theorem 4.1: If-Then-Else

$$\frac{(\!|\, (P \wedge B) \,|\!)\ C_1\ (\!|\, Q \,|\!) \qquad (\!|\, (P \wedge (\neg B)) \,|\!)\ C_2\ (\!|\, Q \,|\!)}{(\!|\, P \,|\!)\ \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2\ (\!|\, Q \,|\!)} \quad \text{(if-then-else)}$$

Template 4.1: If-Then-Else

```
(| $P$ |)
if ( $B$ ) {
    (| $(P \wedge B)$ |)                      if-then-else
    $C_1$
    (| $Q$ |)                                 justification based on $C_1$
} else {
    (| $(P \wedge (\neg B))$ |)               if-then-else
    $C_2$
    (| $Q$ |)                                 justification based on $C_2$
}
(| $Q$ |)                                     if-then-else
```

Example: If-Then-Else

```
(| true |)
if ( x > y ) {
    (| $(\text{true} \wedge (x > y))$ |)                                                      if-then-else
    (| $\big(((x > y) \wedge (x = x)) \vee ((x \leq y) \wedge (x = y))\big)$ |)               implied (A)
    max = x;
    (| $\big(((x > y) \wedge (\text{max} = x)) \vee ((x \leq y) \wedge (\text{max} = y))\big)$ |)   assignment
} else {
    (| $(\text{true} \wedge (\neg\ (x > y)))$ |)                                              if-then-else
    (| $\big(((x > y) \wedge (y = x)) \vee ((x \leq y) \wedge (y = y))\big)$ |)               implied (B)
    max = y;
    (| $\big(((x > y) \wedge (\text{max} = x)) \vee ((x \leq y) \wedge (\text{max} = y))\big)$ |)   assignment
}
(| $\big(((x > y) \wedge (\text{max} = x)) \vee ((x \leq y) \wedge (\text{max} = y))\big)$ |)       if-then-else
```

Proof of implied (A) and (B) are omitted due to space constraint.

4.2 If-Then

Theorem 4.2: If-Then

$$\frac{\llparenthesis\, (P \wedge B) \,\rrparenthesis\; C \;\llparenthesis\, Q \,\rrparenthesis \qquad \big((P \wedge (\neg B)) \to Q\big)}{\llparenthesis\, P \,\rrparenthesis\; \texttt{if}\; B\; \texttt{then}\; C\; \llparenthesis\, Q \,\rrparenthesis} \quad \text{(if-then)}$$

Template 4.2: If-Then

```
⦇ $P$ ⦈
if ( $B$ ) {
    ⦇ $(P \wedge B)$ ⦈                          if-then
    $C$
    ⦇ $Q$ ⦈                          justification based on $C$
}
⦇ $Q$ ⦈                                          if-then
                          implied: $\big((P \wedge (\neg B)) \to Q\big)$
```

Example: If-Then

```
⦇ true ⦈
if ( max < x ) {
    ⦇ (max < $x$) ⦈                                  if-then
    ⦇ ($x \geq x$) ⦈                                 implied (A)
    max = x;
    ⦇ (max $\geq x$; ) ⦈                             assignment
}
⦇ (max $\geq x$) ⦈                                   if-then
              implied (B): $\big((\neg(\text{max} < x)) \to (\text{max} \geq x)\big)$
```

Proof of implied (A) Assume (max $< x$) is true. ($x \geq x$) is true by reflexivity of $=$.

Proof of implied (B) Assume ($\neg$ (max $< x$)). By definition of $\neg$, we have (max $\geq x$).

5 While Loops

Recall that total correctness of a while loop requires partial correctness and termination.

5.1 Partial While

Theorem 5.1: Partial While

$$\frac{(\!|\ (I \wedge B)\ |\!)\ C\ (\!|\ I\ |\!)}{(\!|\ I\ |\!)\ \texttt{while}\ (B)\ C\ (\!|\ (I \wedge (\neg B))\ |\!)} \quad \text{(partial-while)}$$

Template 5.1: Partial While

$(\!\mid P \mid\!)$	
$(\!\mid I \mid\!)$	implied (A): $P \to I$
`while` (B) {	
$(\!\mid (I \wedge B) \mid\!)$	partial-while
C	
$(\!\mid I \mid\!)$	justification based on C
}	
$(\!\mid (I \wedge (\neg B)) \mid\!)$	partial-while
$(\!\mid Q \mid\!)$	implied (B): $((I \wedge (\neg B)) \to Q)$

Intuition

- In words, if the code C satisfies the triple $(\!|\ (I \wedge B)\ |\!)\ C\ (\!|\ I\ |\!)$ and I is true at the start of the `while` loop, then no matter how many times we execute C, condition I will still be true.

Loop Invariant

- The condition I in the theorem above is called a **loop invariant**, an assertion (condition) that is true both *before* and *after* each execution of the body of a loop.
- The loop invariant expresses a relationship among the variables used within the body of the loop. Some of thes variables will have their values changed within the loop.
- Recall that the postcondition is the ultimate goal of our while loop and we are making progress towards it at every iteration. The loop invariant describes our progress.

Finding a Loop Invariant

1. Write down the values of all the variables every time the loop guard is reached.
2. Find relationships (that can be expressed using propositional or predicate logic) among the variables that are true for every while test; these are our candidate invariants.
3. Try each candidate invariant until we find one that works for our proof. To check whether an invariant leads to a valid proof, we need to check whether all of the implied's can be proved.

Example: Partial-While

```
$\llparenthesis\ (x \geq 0)\ \rrparenthesis$
$\llparenthesis\ (1 = 0!)\ \rrparenthesis$                                   implied (A)
y = 1;
$\llparenthesis\ (y = 0!)\ \rrparenthesis$                                   assignment
z = 0;
$\llparenthesis\ (y = z!)\ \rrparenthesis$                                   assignment
while (z != x) {
    $\llparenthesis\ \big((y = z!) \land (\neg\,(z = x))\big)\ \rrparenthesis$      partial-while
    $\llparenthesis\ \big((y \cdot (z+1)) = (z+1)!\big)\ \rrparenthesis$          implied (B)
    z = z + 1;
    $\llparenthesis\ \big((y \cdot z) = z!\big)\ \rrparenthesis$                    assignment
    y = y * z;
    $\llparenthesis\ (y = z!)\ \rrparenthesis$                               assignment
}
$\llparenthesis\ \big((y = z!) \land (z = x)\big)\ \rrparenthesis$                partial-while
$\llparenthesis\ (y = x!)\ \rrparenthesis$                                   implied (C)
```

Proof A By definition of factorial, $(x \geq 0) \vdash (1 = 0!)$.

Proof B Since $y = z!$, multiplying both sides by $(z+1)$ and by definition of factorial, line B holds.

Proof C Since $y = z!$ and $z = x$, a simple substitution completes the proof.

5.2 Termination

Motivation

- Since while loops are responsible for any non-termination (other than infinite recursion) in our program, we must prove it terminates in order to prove its total correctness.
- To prove termination, we want to identify an *integer* expression that is
  1. *always* non-negative, and
  2. the value *decreases* every time the loop body runs
- This integer expression is called a **loop variant**, which usually corresponds with the loop condition called the **loop guard**.

Example: Loop Variant

- The loop guard $(z \neq x)$ in the example above implies $\big((x - z) \neq 0\big)$.
- Moreover, in the beginning we have $(x \geq 0)$ and $(z = 0)$, thus $(x - z) \geq 0$.
- Choosing the variant $(x - z)$ works as $(x - z)$ eventually reaches 0 and the loop terminates.

Example: More on Partial-While

Original Problem

```
⦇ $((n \geq 0) \wedge (a \geq 0))$ ⦈
s = 1;
i = 0;
while (i < n) {
    s = s * a;
    i = i + 1;
}
⦇ $(s = a^n)$ ⦈
```

Proving the program is satisfied under partial correctness.

Trace the program

i	n	a	s
0	n	a	1
1	n	a	a
2	n	a	a^2
3	n	a	a^3
$\cdots$	$\cdots$	$\cdots$	$\cdots$
k	n	a	a^k

Observe a and n are not mutated throughout the program. From the table above, we conclude that $s = a^i$; this shall be our first loop invariant.

First Try

```
⦇ $((n \geq 0) \wedge (a \geq 0))$ ⦈
⦇ ... ⦈
s = 1;
⦇ ... ⦈
i = 0;
⦇ $(s = a^i)$ ⦈
while (i < n) {
    ⦇ $((s = a^i) \wedge (i < n))$ ⦈        partial-while
    ⦇ ... ⦈
    s = s * a;
    ⦇ ... ⦈
    i = i + 1;
    ⦇ $(s = a^i)$ ⦈
}
⦇ $((s = a^i) \wedge (i \geq n))$ ⦈          partial-while
⦇ $(s = a^n)$ ⦈
```

We see a problem: using only $(I \wedge (\neg B))$, we cannot conclude the postcondition Q. Indeed, $i \leq n$ can be concluded from the table, so why don't we add it as another part of the loop invariant, and in the end $((i \leq n) \wedge (i \geq n)) \implies (i = n)$.

Fix

```
⦇ $((n \geq 0) \wedge (a \geq 0))$ ⦈
⦇ $((1 = a^0) \wedge (0 \leq n))$ ⦈                              implied (A)
s = 1;
⦇ $((s = a^0) \wedge (0 \leq n))$ ⦈                              assignment
i = 0;
⦇ $((s = a^i) \wedge (i \leq n))$ ⦈                              assignment
while (i < n) {
    ⦇ $(((s = a^i) \wedge (i \leq n)) \wedge (i < n))$ ⦈          partial-while
    ⦇ $(((s \cdot a) = a^{i+1}) \wedge ((i + 1) \leq n)))$ ⦈      implied (B)
    s = s * a;
    ⦇ $((s = a^{i+1}) \wedge ((i + 1) \leq n)))$ ⦈               assignment
    i = i + 1;
    ⦇ $((s = a^i) \wedge (i \leq n))$ ⦈
}
⦇ $(((s = a^i) \wedge (i \leq n)) \wedge (\neg(i < n)))$ ⦈        partial-while
⦇ $(s = a^n)$ ⦈                                                  implied (C)
```

6 Array Assignments

6.1 Array Assignment

Theorem 6.1: Array Assignment

$$\frac{}{\llparenthesis\ Q[A\{e_1 \leftarrow e_2\}/A]\ \rrparenthesis\ \texttt{A[e1] = e2;}\ \llparenthesis\ Q\ \rrparenthesis} \quad \text{(Array Assignment)}$$

where

$$A\{e_1 \leftarrow e_2\}[i] = \begin{cases} e_2 & \text{if } i = e_1 \\ A[i] & \text{if } i \neq e_1 \end{cases}$$

Intuition

- $A\{e_1 \leftarrow e_2\}$ is a new array which is identical to array A except that the e_1th element is e_2.

Example: Basic Array Assignment

$\llparenthesis\ \big((A[x] = x_0) \wedge (A[y] = y_0)\big)\ \rrparenthesis$

$\llparenthesis\ \big((A\{x \leftarrow A[y]\}\{y \leftarrow A[x]\}[x] = y_0) \wedge (A\{x \leftarrow A[y]\}\{y \leftarrow A[x]\}[y] = x_0)\big)\ \rrparenthesis$ implied (A)

`t = A[x];`

$\llparenthesis\ \big((A\{x \leftarrow A[y]\}\{y \leftarrow t\}[x] = y_0) \wedge (A\{x \leftarrow A[y]\}\{y \leftarrow t\}[y] = x_0)\big)\ \rrparenthesis$ assignment

`A[x] = A[y];`

$\llparenthesis\ \big((A\{y \leftarrow t\}[x] = y_0) \wedge (A\{y \leftarrow t\}[y] = x_0)\big)\ \rrparenthesis$ array assignment

`A[y] = t;`

$\llparenthesis\ \big((A[x] = y_0) \wedge (A[y] = x_0)\big)\ \rrparenthesis$ array assignment

Proof A Show the following lemma:

$$A\{x \leftarrow A[y]\}\{y \leftarrow A[x]\}[x] = A[y]$$
$$A\{x \leftarrow A[y]\}\{y \leftarrow A[x]\}[y] = A[x]$$

6.2 Reversing an Array

Given an array R with n elements, we want to reverse the elements, i.e. exchange $R[j]$ with $R[n+1-j]$ for each $i \leq j \leq \lfloor n/2 \rfloor$. A possible solution is shown on the right. The pre- and postcondition are provided; we want to find a loop invariant.

$P: \big(\forall x\ ((1 \leq x \leq n) \rightarrow (R[x] = r_x))\big)$

$Q: \big(\forall x\ ((1 \leq x \leq n) \rightarrow (R[x] = r_{n+1-x}))\big)$

```
j = 1;
while (2 * j <= n) {
    t = R[j];
    R[j] = R[n+1-j];
    R[n+1-j] = t;
    j = j + 1;
}
```

Observation

At iteration j, consider an index x with restriction $1 \leq x \leq n$:

- If $1 \leq x < j$, $R[x]$ and $R[n+1-x]$ have already been exchanged.
- If $j \leq x \leq \frac{n+1}{2}$, then $R[x]$ and $R[n+1-x]$ are not swapped yet.

In other words, one pair is swapped in each iteration, and at iteration j, every item before jth has been swapped with its counterpart, but the middle section is left unchanged.

Loop Invariant

$$\begin{aligned}\mathrm{Inv}'(j) = \Big(\forall x \,\Big(& (((1 \leq x) \wedge (x < j)) \to ((R[x] = r_{n+1-x}) \wedge (R[n+1-x] = r_x))) \\ & \wedge \big(((j \leq x) \wedge (x \leq \frac{n+1}{2})) \to ((R[x] = r_x) \wedge (R[n+1-x] = r_{n+1-x}))\big)\Big)\Big)\end{aligned}$$

Further, let $R' = R\{j \leftarrow R[((n+1)-j)]\}\{((n+1)-j) \leftarrow R[j]\}$.

Annotation

```
$(\!| \big(\forall x(((1 \leq x) \wedge (x \leq n)) \to (R[x] = r_x))\big) |\!)$
$(\!| \big(\mathrm{Inv}'(1) \wedge (1 \leq (\frac{n}{2} + 1))\big) |\!)$                implied A
j = 1;
$(\!| \big(\mathrm{Inv}'(j) \wedge (j \leq (\frac{n}{2} + 1))\big) |\!)$                assignment
while (2 * j <= n) {
    $(\!| \big((\mathrm{Inv}'(j) \wedge (j \leq \frac{n}{2} + 1))) \wedge ((2 \cdot j) \leq n)\big) |\!)$      partial-while
    $(\!| \big(\mathrm{Inv}'(j+1)[R'/R] \wedge ((j+1) \leq (\frac{n}{2} + 1))\big) |\!)$      implied B
    t = R[j];
    R[j] = R[n+1-j];
    R[n+1-j] = t;
    $(\!| \big(\mathrm{Inv}'(j+1) \wedge ((j+1) \leq (\frac{n}{2} + 1))\big) |\!)$      Omitted as proof has been provided above.
    j = j + 1;
    $(\!| \big(\mathrm{Inv}'(j) \wedge (j \leq \frac{n}{2} + 1))\big) |\!)$      assignment
}
$(\!| \big((\mathrm{Inv}'(j) \wedge (j \leq (\frac{n}{2} + 1))) \wedge ((2 \cdot j) > n)\big) |\!)$      partial-while
$(\!| \big(\forall x(((1 \leq x) \wedge (x \leq n)) \to (R[x] = r_{n+1-x}))\big) |\!)$      implied C
```

Termination

Consider the loop variant $V = \lfloor \frac{n}{2} \rfloor + 1 - j$.

- Start: when $j = 1$, $V \geq 0$ as $n \geq 0$; loop variant satisfies being non-negative.
- Mutate: j is incremented by 1 each iteration, thus V would eventually reach 0.
- Break: if $V = 0$, we have $j = \frac{n}{2} + 1 > \frac{n}{2}$, loop guard fails and loop terminates.

Proof A

Recall that the loop invariant states that elements from 1 to $j-1$ have been swapped and elements from j to $\frac{n+1}{2}$ have not been swapped, plus the fact that $j \leq \lfloor \frac{n}{2} \rfloor$. With j being 1, no swap has occurred. The second part follows from the fact that $n \geq 0$.

Proof B

First, keep in mind this implication happens before the swap, meaning the array should not been modified yet. $\text{Inv}'(j)$ in the hypothesis claims the first and last $j - 1$ elements have been taken care of; $\text{Inv}'(j + 1)$ in the conclusion differs only on entries j and $(n - j + 1)$ (they have been swapped). By definition of R', it is the new array with j-th and $(n - j + 1)$-th terms swapped. In other words, $\text{Inv}'(j + 1)$ swaps the first j elements, then $[R'/R]$ swaps the last element again — the $\{j, n - j + 1\}$ pair gets swapped twice so nothing happened; the array remains the same. The conclusion should be obvious as nothing is changed.

Proof C

Case 1. n is even. Then $n/2$ gives an integer; $((j \leq (\frac{n}{2} + 1)) \wedge (2 \cdot j) > n))$ implies $j = \frac{n}{2} + 1$. Plug in $j = \frac{n}{2} + 1$ in $\text{Inv}'(j)$, we get

$$\begin{aligned}\text{Inv}'(\frac{n}{2}+1) = \Big(\forall x \, \Big(& (((1 \leq x) \wedge (x < (\frac{n}{2}+1))) \rightarrow ((R[x] = r_{n+1-x}) \wedge (R[n+1-x] = r_x))) \\ & \wedge \big((((\frac{n}{2}+1) \leq x) \wedge (x \leq \frac{n+1}{2})) \rightarrow ((R[x] = r_x) \wedge (R[n+1-x] = r_{n+1-x}))\big)\Big)\Big)\end{aligned}$$

Obviously the hypothesis for the second implication is non-satisfiable, so we can reduce the statement into its first part. Given $1 \leq x \leq \frac{n}{2}$, we get the conclusion that all elements have been swapped. This is the desired result.

Case 2. n is odd. Then $\frac{n+1}{2}$ is an integer, so $((j \leq \frac{n}{2} + 1) \wedge (2 \cdot j > n))$, or equivalently $((j \leq \frac{n+1}{2} + \frac{1}{2}) \wedge (2 \cdot j > n))$ gives $j = \frac{n+1}{2}$. Then

$$\begin{aligned}\text{Inv}'(\frac{n+1}{2}) = \Big(\forall x \, \Big(& (((1 \leq x) \wedge (x < (\frac{n+1}{2}))) \rightarrow ((R[x] = r_{n+1-x}) \wedge (R[n+1-x] = r_x))) \\ & \wedge \big((((\frac{n+1}{2}) \leq x) \wedge (x \leq \frac{n+1}{2})) \rightarrow ((R[x] = r_x) \wedge (R[n+1-x] = r_{n+1-x}))\big)\Big)\Big)\end{aligned}$$

First thing to notice is $n + 1 - \frac{n+1}{2} = \frac{n+1}{2}$ so that the middle element is not swapped (which is good). Reducing the formula to $(1 \leq x \leq \frac{n-1}{2} \rightarrow (R[x] = r_{n+1-x} \wedge R[n+1-x] = r_x))$, this asserts that all the required swaps have been performed and the postcondition is thus true.