

Class 07: More Verification Examples

*Changed
Oct 26*

A. Why?

Studying more examples will help with understanding the proof rules and with using proof outlines.

B. Outcomes

By the end of the class you should

- Have an understanding of how loop initialization, loop invariants, and loop bodies are all interconnected.
- Be able to read and generate proof outlines involving loops, given a loop invariant.

C. Return Quiz 2

D. Modified Versions of Summation Loop (Changes to Initialization)

- Last time we saw a particular example of summing the integers from 0 through n :

$$\{n \geq 0\} \{p[s:=0][i:=0]\} i:=0; \{p[s:=0]\} s:=0;$$

$$\{\text{inv } p \equiv 0 \leq i \leq n \wedge s = \text{sum}(0, i)\}$$

while $i < n$ **do**

$$\{p \wedge i < n\} \{p[s:=s+i][i:=i+1]\} i:=i+1;$$

$$\{p[s:=s+i]\} s:=s+i \{p\}$$

od

$$\{p \wedge i \geq n\}$$

$$\{s = \text{sum}(0, n)\}$$

where

- $p[s:=0] \equiv 0 \leq i \leq n \wedge 0 = \text{sum}(0, i)$
- $p[s:=0][i:=0] \equiv 0 \leq 0 \leq n \wedge 0 = \text{sum}(0, 0)$
- $p[s:=s+i] \equiv 0 \leq i \leq n \wedge s+i = \text{sum}(0, i)$
- $p[s:=s+i][i:=i+1] \equiv 0 \leq i+1 \leq n \wedge s+(i+1) \leq n \wedge s = \text{sum}(0, i+1)$
- One modification: change the initialization of i from $i:=0$ to $i:=1$. Then we also need to change the initialization of s and the initial precondition:

$$\{n \geq 1\} \{p[s:=1][i:=1]\} i:=1; \{p[s:=1]\} s:=1 \{p\}$$

So here, $p[s:=1][i:=1] \equiv (0 \leq i \leq n \wedge s = \text{sum}(0, i))[s:=1][i:=1]$

*Fixed
Oct 26*

$$\equiv (0 \leq 1 \leq n \wedge 1 = \text{sum}(0, 1))$$

The rest of the proof doesn't have to change (we could change the $0 \leq i$ clause of p to $1 \leq i$ if you like tighter bounds).

- Alternatively, we can initialize $i := 1$ and keep $s := 0$, but change the invariant, to $p_1 \equiv 0 \leq i-1 \leq n \wedge s = \text{sum}(0, i-1)$. We must change the loop test: We can loop while $i-1 < n$ (or $i \leq n$ or $i-1 \neq n$, etc). The loop postcondition becomes $p_1 \wedge i-1 \geq n$, which implies $s = \text{sum}(0, n)$.

- Another alternative has us initialize $s := 1$ and keep $i := 0$:

```

{ n ≥ 0 } { p2[s := 1][i := 0] } i := 0; { p[s := 1] } s := 1;
{ inv p2 ≡ 0 ≤ i+1 ≤ n ∧ s = sum(0, i+1) }
while i+1 < n do ... od /* loop body doesn't change */
{ p2 ∧ i+1 ≥ n }
{ s = sum(0, n) }

```

E. Partial Proof Outlines

- A proof outline is pretty much determined by the outer pre- and post-conditions and the loop invariants.
- To cut down on the writing, we can use **partial proof outlines**: proof outlines that are incomplete but can be expanded to the full proof outlines we've been using.
- Example:

```

{ n ≥ 0 } i := 0; s := 0;
{ inv p ≡ 0 ≤ i ≤ n ∧ s = sum(0, i) }
while i < n do i := i+1; s := s+i od
{ s = sum(0, n) }

```

- Example:

```

{ n ≥ 1 } i := 1; s := 1;
{ inv p ≡ 0 ≤ i ≤ n ∧ s = sum(0, i) }
while i < n do i := i+1; s := s+i od
{ s = sum(0, n) }

```

- Example:

```

{ n ≥ 0 } i := 1; s := 0;
{ inv p1 ≡ 0 ≤ i-1 ≤ n ∧ s = sum(0, i-1) }
while i-1 < n do i := i+1; s := s+i od
{ s = sum(0, n) }

```

- Example

```

{ $n \geq 0$ }  $i := 0; s := 1;$ 
{inv  $p_2 \equiv 0 \leq i+1 \leq n \wedge s = \text{sum}(0, i+1)$ }
while  $i+1 < n$  do  $i := i+1; s := s+i$  od
{ $s = \text{sum}(0, n)$ }

```

F. Modified Summation Loop (Changes To The Loop Body)

- **Definition:** A **progress step** is a part of a loop that brings us closer to termination. **Making progress** (toward termination) means executing a progress step.
 - For our example, incrementing i is the progress step.
 - Many loop bodies have the form
 - $\{invariant \wedge loop\ test\} S; \{p'\}$ *progress step* $\{invariant\}$
 - p' ensures that making progress reestablishes the invariant.
 - For us, $\{p \wedge i < n\} s := s+e; \{p'\} i := i+1 \{p\}$ where $p' \equiv p[i := i+1]$ and e needs to be figured out.
 - To make the loop work, we need $(p \wedge i < n) \rightarrow p'[s := s+e]$
 - $p'[s := s+e] \equiv 0 \leq i+1 \leq n \wedge s+e = \text{sum}(0, i+1)$
 - p tells us that $s = \text{sum}(0, i)$
 - So $e = \text{sum}(0, i+1) - s = \text{sum}(0, i+1) - \text{sum}(0, i) = i+1$.

G. Using Invariants Simplifies Loop Writing

- A loop invariant lets you worry about initialization, termination, and progress separately.
- With $\{r\} S_1; \{\mathbf{inv} p\} \mathbf{while} B \mathbf{do} S_2 \mathbf{od} \{p \wedge \neg B\} \{q\}$
 - The initialization S_1 depends only on r and p .
 - The loop test B depends only on p and q .
 - The loop body S_2 depends only on p and B .

 (This is as far as we got.)

H. Derived Assignment Rules

- The regular assignment rule is “backward”: $\{p[u := t]\} u := t \{p\}$
- It’s possible to derive assignment rules that work “forward”.

Forward Assignment Rule 1

- $\{p \wedge u = c\} u := t \{p[u := c] \wedge u = t[u := c]\}$ where c is a constant
 - (Let’s make substitution have high precedence.)
 - c stands for the value of u before the assignment.
 - Probably a named constant.
 - Probably doesn’t appear in the program, just the proof.
- Derivation from backward assignment rule:

$$\begin{array}{l} \{p \wedge u = c\} \\ \{p[u := c] \wedge t = t[u := c]\} \\ \{p[u := c][u := t] \wedge u[u := t] = t[u := c][u := t]\} \\ \{(p[u := c] \wedge u = t[u := c])[u := t]\} \\ u := t \\ \{p[u := c] \wedge u = t[u := c]\} \end{array}$$

Forward Assignment Rule 2

- If u is a new variable, we can simplify even more:
 - $\{p\} u := t \{p \wedge u = t\}$ where u not free in p or t .
- Derivation from forward assignment rule 1:
 - u not free in p or t , so $p[u := c] \equiv p$ and $t[u := c] \equiv t$.
 - So postcondition $(p[u := c] \wedge u = t[u := c]) \equiv (p \wedge u = t)$.
 - So $\{p \wedge u = c\} u := t \{p \wedge u = t\}$
 - For precondition, we can start with p and introduce c as a new named constant: $p \rightarrow p \wedge u = c$ is the definition of c .
 - So $\{p\} u := t \{p \wedge u = t\}$ if u not free in p or t .

I. Binary Search Example (Version 1)

- An example that uses a forward assignment rule is binary search: calculating the midpoint introduces it as a new variable.
- Let $Sorted(b, n) \equiv \forall i: \forall j: 0 \leq i < j \leq n \rightarrow b[i] \leq b[j]$.
- Specification:

$$\{Sorted(b, n) \wedge n \geq 1 \wedge b[0] \leq x < b[n]\}$$

$$binsearch(b, x, n)$$

$$\{0 \leq lt < n \wedge b[lt] \leq x < b[lt+1]\}$$

- (It's handy to have $b[n]$ be a sentinel value.)
- Let invariant $p \equiv 0 \leq lt < rt \leq n \wedge b[lt] \leq x < b[rt]$
 - Treat b, n as named constants, so $Sorted(b, n)$ can be used anywhere, so it doesn't have to be part of p .
- Initialization $\{p[lt:=0][rt:=n]\} \quad lt:=0; \quad rt:=n \quad \{p\}$
 - $p[lt:=0][rt:=n] \equiv 0 \leq 0 < n \leq n \wedge b[0] \leq x < b[n]$
 - In particular, we need $n > 0$ and $b[0] \leq x < b[n]$.
- Loop test: Loop until $rt = lt+1$; loop while $rt \neq lt+1$
- Loop so far:
 - $\{q_0\} \quad lt:=0; \quad rt:=n$
 - **{inv p } while $rt \neq lt+1$ do ... od**
 - $\{p \wedge rt = lt+1\}$
 - $\{q_1\}$
 - where $q_0 \equiv Sorted(b, n) \wedge n \geq 1 \wedge b[0] \leq x < b[n]$
 - $p \equiv 0 \leq lt < rt \leq n \wedge b[lt] \leq x < b[rt]$
 - $q_1 \equiv 0 \leq lt < n \wedge b[lt] \leq x < b[lt+1]$
- For the loop body, we calculate a midpoint $m := (lt+rt)/2$ (with truncating division) and make progress via $lt := m$ or $rt := m$.
- What needs to be true before making progress?
 - Check $\{p[lt:=m]\} \quad lt:=m \quad \{p\}$
 - $p[lt:=m] \equiv (0 \leq lt < rt \leq n \wedge b[lt] \leq x < b[rt])[lt:=m]$
 - $\equiv 0 \leq m < rt \leq n \wedge b[m] \leq x < b[rt]$
 - Check $\{p[rt:=m]\} \quad lt:=m \quad \{p\}$
 - $p[rt:=m] \equiv (0 \leq lt < rt \leq n \wedge b[lt] \leq x < b[rt])[rt:=m]$
 - $\equiv 0 \leq lt < m \leq n \wedge b[lt] \leq x < b[m]$
- Establishing whether $b[m] \leq x$ or $x < b[m]$ can be done with an if test.
- What about $lt < m < rt$?
 - Together $lt < rt$ and $rt \neq lt+1$ imply $rt - lt \geq 2$.
 - Let $rt - lt = 2 + d$ where $d \geq 0$
 - $lt < m = (lt+rt)/2 = (lt+(lt+2+d))/2 = lt+1+(d/2)$
 - $rt > m = (lt+rt)/2 = ((rt-2-d)+rt)/2 = rt-1-(d/2)$
- So the loop body is

```

{p ∧ rt ≠ lt+1}
m := (lt+rt)/2;
{p ∧ rt ≠ lt+1 ∧ m = (lt+rt)/2} /* call this p1 */
if b[m] ≤ x then
  {p1 ∧ b[m] ≤ x} {p[lt := m]} lt := m {p}
else
  {p1 ∧ b[m] > x} {p[rt := m]} rt := m {p}
fi {p}

```

J. Binary Search (Version 2)

- The search program above is a bit inefficient because it doesn't stop early if it finds x (partly because it finds the rightmost occurrence of x).
- Use new postcondition q_2 and invariant p_2 :
 - $q_2 \equiv 0 \leq lt < n \wedge (b[lt] = x \vee b[lt] < x < b[lt+1])$
 - $p_2 \equiv 0 \leq lt < rt \leq n \wedge (b[lt] = x \vee b[lt] < x < b[rt])$
 - Now if $b[m] = x$, we can set $lt = rt - 1 = m$ and halt immediately.
- The initialization code and loop test are unchanged, but its correctness proof is different:

```

{q0 ≡ Sorted(b, n) ∧ n ≥ 1 ∧ b[0] ≤ x < b[n]} /* Same q0 */
{p2[rt := n][lt := 0]} lt := 0; {p2[rt := n]} rt := n;
{inv p2} while rt ≠ lt+1 do ... od
{p2 ∧ rt = lt+1}
{q2 ≡ 0 ≤ lt < n ∧ (b[lt] = x ∨ b[lt] < x < b[lt+1])}

```

- The loop body changes because we test for $b[m] <$, $=$, or $> x$:

```

{p2 ∧ rt ≠ lt+1}
m := (lt+rt)/2;
{p2 ∧ rt ≠ lt+1 ∧ m = (lt+rt)/2} /* call this p3 */
if b[m] < x then
  {p3 ∧ b[m] < x} {p2[lt := m]} lt := m; {p2}
else {p3 ∧ b[m] ≥ x} if b[m] > x then
  {p3 ∧ ... ∧ b[m] > x} {p2[rt := m]} rt := m {p2}
else {p3 ∧ b[m] ≥ x ∧ b[m] ≤ x}
  {p2[rt := m+1][lt := m]} lt := m; {p2[rt := m+1]}
  {p2[rt := m+1]} rt := m+1; {p2}
fi {p2} fi {p2}

```

- With just the code and outer conditions, we have

```
{ $p_2 \wedge rt \neq lt+1$ }  
 $m := (lt+rt)/2$ ;  
if  $b[m] < x$  then  
     $lt := m$ ;  
else if  $b[m] > x$  then  
     $rt := m$   
else  
     $lt := m$ ;  $rt := m+1$ ;  
fi { $p_2$ } fi { $p_2$ }
```