CSE 331
Software Design & Implementation

Kevin Zatloukal

Fall 2017

Lecture 3 – Reasoning About Loops

(Based on slides by Mike Ernst, Dan Grossman, David Notkin, Hal Perkins, Zach Tatlock)

Reminders

- HW1 is due **tomorrow**

- HW2 will be posted on Wednesday
  - covers loops
  - it is harder, so more time given

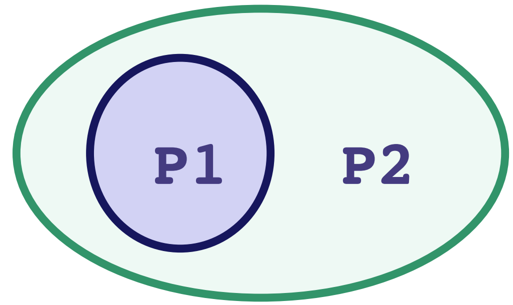
- Reading Quiz 1 is due Friday

- For those still trying to add the course:
  https://goo.gl/forms/Ws8LW8UtvBjH0RiA2

Weaker vs Stronger

If "whenever P1 holds, P2 also holds", then:

- P1 is called stronger than P2
- P2 is called weaker than P1



- It is more (or at least as) "difficult" to satisfy P1
  - the program states where P1 holds are a subset of the states where P2 holds
- P1 puts more constraints on program states
- P1 is a stronger set of requirements

- We do not always have P1 stronger than P2 or vice versa!
  - most assertions are incomparable

Applications to Hoare Logic

- Suppose:
  - **`{{ P }} S {{ Q }}`** is valid and
  - some **`P1`** is *stronger* than **`P`** and
  - some **`Q1`** is *weaker* than **`Q`**

- Then these are all valid too:
  - **`{{ P1 }} S {{ Q }}`**
    - a state where **`P1`** holds is one where **`P`** also holds
  - **`{{ P }} S {{ Q1 }}`**
    - a state where **`Q`** holds is one where **`Q1`** also holds
  - **`{{ P1 }} S {{ Q1 }}`**

Weakest preconditions

- Suppose we know **`Q`** and **`S`**
- There are potentially many **`P`** such that {{ **`P`** }} **`S`** {{ **`Q`** }} is valid

- Would be ideal if there were a *unique* weakest precondition **P**
  – most general assumptions under which **`S`** makes **`Q`** hold
  – get a valid triple for **`P1`** if and only if **`P1`** implies **`P`**

- Amazingly, without loops, for any **`S`** and **`Q`**, this exists!
  – we denote this by wp(**`S`**,**`Q`**)
  – can be found by general rules

- This is just **backward reasoning**!

Rules for weakest preconditions

- wp(**`x = e`**, **`Q`**) is **`Q[x=e]`**
  - Example: wp(**`x = 2*y`**, **`x > 4`**) = **`2*y > 4`**, i.e., **`y > 2`**
- wp(**`S1;S2`**, **`Q`**) is wp(**`S1`**, wp(**`S2`**,**`Q`**))
  - i.e., let **`R`** be wp(**`S2`**,**`Q`**) and overall wp is wp(**`S1`**,**`R`**)
  - Example: wp(**`y = x+1`**, wp(**`z = y+1`**, **`z > 2`**)) =
    wp(**`y = x+1`**, **`y+1 > 2`**) =
    `(x+1)+1 > 2` or equivalently **`x > 0`**
- wp(**`if b S1 else S2`**, **`Q`**) is this logic formula:

  `(b and` wp(**`S1`**,**`Q`**)**`) or (!b and`** wp(**`S2`**,**`Q`**)**`)`**

  - you need wp(**`S1`**,**`Q`**) if **`S1`** is executed and wp(**`S2`**,**`Q`**) if **`S2`** is
  - you can often simplify the result considerably

More Examples

- If **`S`** is **`x = y*y`** and **`Q`** is **`x > 4`**,
  then wp(**`S`**,**`Q`**) is **`y*y > 4`**, i.e., **`|y| > 2`**

- If **`S`** is **`y = x + 1; z = y – 3;`** and **`Q`** is **`z = 10`**,
  then wp(**`S`**,**`Q`**) …
  = wp(**`y = x + 1; z = y – 3`**, **`z = 10`**)
  = wp(**`y = x + 1`**, wp(**`z = y – 3`**, **`z = 10`**))
  = wp(**`y = x + 1`**, **`y-3 = 10`**)
  = wp(**`y = x + 1`**, **`y = 13`**)
  = **`x+1 = 13`**
  = **`x = 12`**

Bigger Example

`S` is `if (y < 5) { x = y*y; } else { x = y+1; }`

wp(`S`, `x >= 9`)

= (`y < 5 and` wp(`x = y*y`, `x >= 9`))

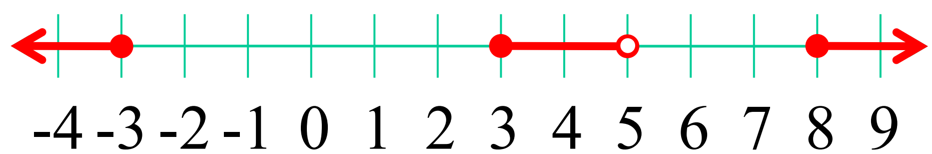
`or` (`y >= 5 and` wp(`x = y+1`, `x >= 9`))

= (`y < 5 and y*y >= 9`)

`or` (`y >= 5 and y+1 >= 9`)

= (`y <= -3`) `or` (`y >= 3 and y < 5`)

`or` (`y >= 8`)



Review: Straight-line Code

Forward & Backward Reasoning

Forward reasoning

{{ P }}
S
{{ ? }}

- P is what we know initially
- Work downward
- Determine strongest post-condition if P is true initially

Backward reasoning

{{ ? }}
S
{{ Q }}

- Q is what we want at the end
- Work upward
- Determine weakest precondition to make Q hold

Assignment Rule

Forward reasoning

{{ P }}
```
x = expr;
```
{{ ? }}

Assignment Rule

Forward reasoning

{{ P }}

```
x = expr;
```

{{ P and x = expr }}

- adds another known fact
- these tend to accumulate…
  - many are irrelevant

(above assumes `x` not used in P)

Assignment Rule

Forward reasoning

{{ P }}
`x = expr;`
{{ P and x = expr }}

- adds another known fact
- these tend to accumulate…
  - many are irrelevant

(above assumes `x` not used in P)

Backward reasoning

{{ ? }}
`x = expr;`
{{ Q }}

Assignment Rule

Forward reasoning

{{ P }}
`x = expr;`
{{ P and x = expr }}

- adds another known fact
- these tend to accumulate…
  - many are irrelevant

(above assumes `x` not used in P)

Backward reasoning

{{ Q[x=expr] }}
`x = expr;`
{{ Q }}

- just substitution
- most general conditions for getting Q after `x = expr;`

Assignment Example

Forward reasoning

{{ w = 3 }}

```
x = y - 5;
```

{{ ? }}

Assignment Example

Forward reasoning

{{ w = 3 }}

```
x = y - 5;
```

{{ w = 3 and x = y - 5 }}

Assignment Example

Forward reasoning

{{ w = 3 }}

```
x = y - 5;
```

{{ w = 3 and x = y - 5 }}

Backward reasoning

{{ ? }}

```
x = y - 5;
```

{{ w = x + 5 }}

Assignment Example

Forward reasoning

{{ w = 3 }}

`x = y - 5;`

{{ w = 3 and x = y - 5 }}

Backward reasoning

{{ w = y }}

`x = y - 5;`

{{ w = x + 5 }}

Sequence Rule

Forward reasoning

{{ P }}
```
S1
S2
```
{{ ? }}

Sequence Rule

Forward reasoning

{{ P }}
`S1`
{{ ? }}
`S2`
{{ ? }}

Sequence Rule

Forward reasoning

{{ P }}
`S1`
{{ P1 }}
`S2`
{{ ? }}

Sequence Rule

Forward reasoning

{{ P }}

`S1`

{{ P1 }}

`S2`

{{ P2 }}

Sequence Rule

Forward reasoning

{{ P }}
`S1`
{{ P1 }}
`S2`
{{ P2 }}

Backward reasoning

{{ ? }}
`S1`
`S2`
{{ Q }}

Sequence Rule

Forward reasoning

{{ P }}
`S1`
{{ P1 }}
`S2`
{{ P2 }}

Backward reasoning

{{ ? }}
`S1`
{{ ? }}
`S2`
{{ Q }}

Sequence Rule

Forward reasoning

{{ P }}
`S1`
{{ P1 }}
`S2`
{{ P2 }}

Backward reasoning

{{ ? }}
`S1`
{{ Q2 }}
`S2`
{{ Q }}

Sequence Rule

Forward reasoning

{{ P }}
`S1`
{{ P1 }}
`S2`
{{ P2 }}

Backward reasoning

{{ Q1 }}
`S1`
{{ Q2 }}
`S2`
{{ Q }}

If-Statement Rule

Forward reasoning

{{ P }}
```
if (cond)
  S1
else
  S2
```
{{ ? }}

If-Statement Rule

Forward reasoning

{{ P }}
```
if (cond)
```
{{ P and `cond` }}
```
S1
else
```
{{ P and not `cond` }}
```
S2
```
{{ ? }}

If-Statement Rule

Forward reasoning

{{ P }}
`if (cond)`
  {{ P and `cond` }}
  `S1`
  {{ P1 }}
`else`
  {{ P and not `cond` }}
  `S2`
  {{ P2 }}
{{ ? }}

If-Statement Rule

Forward reasoning

{{ P }}
`if (cond)`
    {{ P and `cond` }}
    `S1`
    {{ P1 }}
`else`
    {{ P and not `cond` }}
    `S2`
    {{ P2 }}
{{ P1 or P2 }}

If-Statement Rule

Forward reasoning

{{ P }}
`if (cond)`
  {{ P and `cond` }}
  `S1`
  {{ P1 }}
`else`
  {{ P and not `cond` }}
  `S2`
  {{ P2 }}
{{ P1 or P2 }}

Backward reasoning

{{ ? }}
`if (cond)`
  `S1`
`else`
  `S2`
{{ Q }}

If-Statement Rule

Forward reasoning

```
{{ P }}
if (cond)
   {{ P and cond }}
   S1
   {{ P1 }}
else
   {{ P and not cond }}
   S2
   {{ P2 }}
{{ P1 or P2 }}
```

Backward reasoning

```
{{ ? }}
if (cond)
   S1
   {{ Q }}
else
   S2
   {{ Q }}
{{ Q }}
```

If-Statement Rule

Forward reasoning	Backward reasoning
{{ P }}	{{ ? }}
`if (cond)`	`if (cond)`
{{ P and `cond` }}	{{ Q1 }}
`S1`	`S1`
{{ P1 }}	{{ Q }}
`else`	`else`
{{ P and not `cond` }}	{{ Q2 }}
`S2`	`S2`
{{ P2 }}	{{ Q }}
{{ P1 or P2 }}	{{ Q }}

If-Statement Rule

Forward reasoning

```
{{ P }}
if (cond)
    {{ P and cond }}
    S1
    {{ P1 }}
else
    {{ P and not cond }}
    S2
    {{ P2 }}
{{ P1 or P2 }}
```

Backward reasoning

```
{{ cond and Q1 or
    not cond and Q2 }}
if (cond)
    {{ Q1 }}
    S1
    {{ Q }}
else
    {{ Q2 }}
    S2
    {{ Q }}
{{ Q }}
```

If-Statement Example

Forward reasoning

```
{{ }}
if (x >= 0)
  y = x;
else
  y = -x;
{{ ? }}
```

If-Statement Example

Forward reasoning

{{ }}
```
if (x >= 0)
```
{{ x >= 0 }}
```
  y = x;
else
```
{{ x < 0 }}
```
  y = -x;
```
{{ ? }}

If-Statement Example

Forward reasoning

{{ }}
```
if (x >= 0)
```
  {{ x >= 0 }}
```
  y = x;
```
  {{ x >= 0 and y = x }}
```
else
```
  {{ x < 0 }}
```
  y = -x;
```
  {{ x < 0 and y = -x }}

{{ ? }}

If-Statement Example

Forward reasoning

{{ }}
```
if (x >= 0)
```
{{ x >= 0 }}
```
y = x;
```
{{ x >= 0 and y = x }}
```
else
```
{{ x < 0 }}
```
y = -x;
```
{{ x < 0 and y = -x }}

{{ ? }}

Warning: many write {{ y >= 0 }} here

That is true but it is *strictly* weaker.
(It includes cases where y != x)

If-Statement Example

Forward reasoning

```
{{ }}
if (x >= 0)
   {{ x >= 0 }}
   y = x;
   {{ x >= 0 and y = x }}
else
   {{ x < 0 }}
   y = -x;
   {{ x < 0 and y = -x }}
{{ (x >= 0 and y = x) or
   (x < 0 and y = -x) }}
```

If-Statement Example

Forward reasoning

{{ }}
```
if (x >= 0)
```
{{ x >= 0 }}
```
y = x;
```
{{ x >= 0 and y = x }}
```
else
```
{{ x < 0 }}
```
y = -x;
```
{{ x < 0 and y = -x }}

{{ y = |x| }}

If-Statement Example

Forward reasoning

{{ }}
```
if (x >= 0)
```
{{ x >= 0 }}
```
y = x;
```
{{ x >= 0 and y = x }}
```
else
```
{{ x < 0 }}
```
y = -x;
```
{{ x < 0 and y = -x }}

{{ y = |x| }}

Backward reasoning

{{ ? }}
```
if (x >= 0)
  y = x;
else
  y = -x;
```
{{ y = |x| }}

If-Statement Example

Forward reasoning

```
{{ }}
if (x >= 0)
   {{ x >= 0 }}
   y = x;
   {{ x >= 0 and y = x }}
else
   {{ x < 0 }}
   y = -x;
   {{ x < 0 and y = -x }}
{{ y = |x| }}
```

Backward reasoning

```
{{ ? }}
if (x >= 0)
   y = x;
   {{ y = |x| }}
else
   y = -x;
   {{ y = |x| }}
{{ y = |x| }}
```

If-Statement Example

Forward reasoning

```
{{ }}
if (x >= 0)
   {{ x >= 0 }}
   y = x;
   {{ x >= 0 and y = x }}
else
   {{ x < 0 }}
   y = -x;
   {{ x < 0 and y = -x }}
{{ y = |x| }}
```

Backward reasoning

```
{{ ? }}
if (x >= 0)
   {{ x = |x| }}
   y = x;
   {{ y = |x| }}
else
   {{ -x = |x| }}
   y = -x;
   {{ y = |x| }}
{{ y = |x| }}
```

If-Statement Example

Forward reasoning

{{ }}
```
if (x >= 0)
```
  {{ x >= 0 }}
```
  y = x;
```
  {{ x >= 0 and y = x }}
```
else
```
  {{ x < 0 }}
```
  y = -x;
```
  {{ x < 0 and y = -x }}

{{ y = |x| }}

Backward reasoning

{{ ? }}
```
if (x >= 0)
```
  {{ x >= 0 }}
```
  y = x;
```
  {{ y = |x| }}
```
else
```
  {{ x <= 0 }}
```
  y = -x;
```
  {{ y = |x| }}

{{ y = |x| }}

If-Statement Example

Forward reasoning

```
{{ }}
if (x >= 0)
    {{ x >= 0 }}
    y = x;
    {{ x >= 0 and y = x }}
else
    {{ x < 0 }}
    y = -x;
    {{ x < 0 and y = -x }}
{{ y = |x| }}
```

Backward reasoning

```
{{ (x >= 0 and x >= 0) or
    (x < 0 and x <= 0) }}
if (x >= 0)
    {{ x >= 0 }}
    y = x;
    {{ y = |x| }}
else
    {{ x <= 0 }}
    y = -x;
    {{ y = |x| }}
{{ y = |x| }}
```

If-Statement Example

Forward reasoning

{{ }}
```
if (x >= 0)
```
{{ x >= 0 }}
```
y = x;
```
{{ x >= 0 and y = x }}
```
else
```
{{ x < 0 }}
```
y = -x;
```
{{ x < 0 and y = -x }}

{{ y = |x| }}

Backward reasoning

{{ x >= 0 or x < 0 }}
```
if (x >= 0)
```
{{ x >= 0 }}
```
y = x;
```
{{ y = |x| }}
```
else
```
{{ x <= 0 }}
```
y = -x;
```
{{ y = |x| }}

{{ y = |x| }}

If-Statement Example

Forward reasoning

```
{{ }}
if (x >= 0)
    {{ x >= 0 }}
    y = x;
    {{ x >= 0 and y = x }}
else
    {{ x < 0 }}
    y = -x;
    {{ x < 0 and y = -x }}
{{ y = |x| }}
```

Backward reasoning

```
{{ }}
if (x >= 0)
    {{ x >= 0 }}
    y = x;
    {{ y = |x| }}
else
    {{ x <= 0 }}
    y = -x;
    {{ y = |x| }}
{{ y = |x| }}
```

Verifying Correctness (*Inspection*)

Two different ways of checking {{ P }} `S` {{ Q }}

Use forward reasoning:

{{ P }}
`S`
{{ Q' }}

- Find Q' assuming P.
- Check that Q' implies Q.
  - weaken postcondition

Use backward reasoning:

{{ P' }}
`S`
{{ Q }}

- Find P' that produces Q.
- Check that P implies P'.
  - strengthen precondition

You know how to verify correctness of straight-line code.

You will do this on HW1.

Using Both Forward & Backward

Also possible to check correctness by mixing forward & backward:

{{ }}
```
if (x >= 0)
  y = div(x,2);
else
  y = -div(-x+1,2);
```
{{ 2y = x or 2y = x - 1 }}

Assume that `div(a,b)` computes a/b rounded *toward zero*.

Code to compute x/2 rounded toward minus infinity (usual division).

Using Both Forward & Backward

Also possible to check correctness by mixing forward & backward:

```
{{ }}
if (x >= 0)
    {{ x >= 0 }}
    y = div(x,2);
else
    {{ x < 0 }}
    y = -div(-x+1,2);
{{ 2y = x or 2y = x - 1 }}
```

Using Both Forward & Backward

Also possible to check correctness by mixing forward & backward:

{{ }}
```
if (x >= 0)
```
{{ x >= 0 }}
```
y = div(x,2);
```
{{ 2y = x or 2y = x - 1 }}
```
else
```
{{ x < 0 }}
```
y = -div(-x+1,2);
```
{{ 2y = x or 2y = x - 1 }}

{{ 2y = x or 2y = x - 1 }}

Using Both Forward & Backward

Also possible to check correctness by mixing forward & backward:

```
{{ }}
if (x >= 0)
   {{ x >= 0 }}
   y = div(x,2);                 ?
   {{ 2y = x or 2y = x - 1 }}
else
   {{ x < 0 }}
   y = -div(-x+1,2);             ?
   {{ 2y = x or 2y = x - 1 }}
{{ 2y = x or 2y = x - 1 }}
```

Using Both Forward & Backward

Also possible to check correctness by mixing forward & backward:

{{ }}
```
if (x >= 0)
```
{{ x >= 0 }}
```
y = div(x,2);
```
{{ 2y = x or 2y = x - 1 }}
```
else
```
{{ x < 0 }}
```
y = -div(-x+1,2);
```
{{ 2y = x or 2y = x - 1 }}

{{ 2y = x or 2y = x - 1 }}

{{ 2 div(x,2) = x or 2 div(x,2) = x - 1 }}

true if x >= 0

{{ 2 div(-x+1,2) = (-x+1) -1 or 2 div(-x+1,2) = -x+1 }}

true if -x+1 >= 0

One caveat

- With forward reasoning, there is a problem with assignments:
  - changing a variable can affect other assumptions

{{ }}

```
w = x + y;
```

{{ w = x + y }}

```
x = 4;
```

{{ w = x + y and x = 4 }}

```
y = 3;
```

{{ w = x + y and x = 4 and y = 3 }}

- But clearly we do not know w = 7!
- The assertion w = x + y means the *original* values of x and y

One Fix

- Use different names for the values at different points
  - common to use subscripts to distinguish these
  - on every assignment, rename references to the old values

{{ }}

```
w = x + y;
```

{{ w = x + y }}

```
x = 4;
```

{{ w = x_0 + y and x = 4 }}

```
y = 3;
```

{{ w = x_0 + y_0 and x = 4 and y = 3 }}

Useful example: swap

- Consider code for a swapping x and y

{{ }}

```
tmp = x;
```

{{ tmp = x }}

```
x = y;
```

{{ tmp = x_0 and x = y }}

```
y = tmp;
```

{{ tmp = x_0 and x = y_0 and y = tmp }}

- Post condition implies x = y_0 and y = x_0
- I.e., their final values are equal to the original values swapped

Loops

Loop Invariant

A **loop invariant** is one that always holds at the top of the loop:

{{ Inv: I }}
```
while (cond)
  S
```

- It holds when we first get to the loop.
- It holds each time we execute `S` and come back to the top.

Notation: I'll use "Inv:" to indicate a loop invariant.

Lupin variants

While-Loop Rule

Consider a while-loop (other loop forms not too different):

{{ P }} `while (cond) S` {{ Q }}

This triple is valid iff: there is a loop invariant I such that

{{ P }}
{{ Inv: I }}
```
while (cond)
  S
```
{{ Q }}

- I holds initially
- I holds each time we execute `S`
- Q holds when I holds and `cond` is false

While-Loop Rule

Consider a while-loop (other loop forms not too different):

{{ P }} `while (cond) S` {{ Q }}

This triple is valid iff: there is a loop invariant I such that

{{ P }}
{{ Inv: I }}
```
while (cond)
  S
```
{{ Q }}

- P implies I
- I holds each time we execute `S`
- Q holds when I holds and `cond` is false

While-Loop Rule

Consider a while-loop (other loop forms not too different):

{{ P }} `while (cond) S` {{ Q }}

This triple is valid iff: there is a loop invariant I such that

{{ P }}
{{ Inv: I }}
`while (cond)`
`  S`
{{ Q }}

- P implies I
- {{ I and `cond` }} S {{ I }} is valid
- Q holds when I holds and `cond` is false

While-Loop Rule

Consider a while-loop (other loop forms not too different):

{{ P }} `while (cond) S` {{ Q }}

This triple is valid iff: there is a loop invariant I such that

{{ P }}
{{ Inv: I }}
```
while (cond)
   S
```
{{ Q }}

- P implies I
- {{ I and `cond` }} S {{ I }} is valid
- (I and not `cond`) implies Q

While-Loop Rule

Consider a while-loop (other loop forms not too different):

{{ P }} `while (cond) S` {{ Q }}

This triple is valid iff: there is a loop invariant I such that

```
{{ P }}
{{ Inv: I }}
while (cond)
  S
{{ Q }}
```

- P implies I
- {{ I and `cond` }} S {{ I }} is valid
- (I and not `cond`) implies Q

More on Loop Invariants

- We need a loop invariant to check validity of a while loop.
- There is no automatic way to generate these.
  - (A theory course will explain why…)

- For this lecture, all loop invariants will be given.
- Next lecture will discuss how to choose a loop invariant.

- Pro Tip: always document your invariants for non-trivial loops
  - as we just saw, much easier for others to check your code
  - possible exception for loops that are “obvious”
- Pro Tip: with a good loop invariant, the code is easy to write
  - we will see this next time

Example: sum of array

Consider the following code to compute `b[0] + … + b[n-1]`:

{{ b.length >= n }}

```
s = 0;
i = 0;
while (i != n) {
  s = s + b[i];
  i = i + 1;
}
```

{{ s = b[0] + ... + b[n-1] }}

Example: sum of array

Consider the following code to compute `b[0] + … + b[n-1]`:

{{ b.length >= n }}

```
s = 0;
i = 0;
```

{{ Inv: s = b[0] + ... + b[i-1] }}

```
while (i != n) {
  s = s + b[i];
  i = i + 1;
}
```

{{ s = b[0] + ... + b[n-1] }}

Example: sum of array

Consider the following code to compute `b[0] + … + b[n-1]`:

{{ b.length >= n }}

```
s = 0;
i = 0;
```

{{ s = 0 and i = 0 }}

{{ Inv: s = b[0] + ... + b[i-1] }}

```
while (i != n) {
  s = s + b[i];
  i = i + 1;
}
```

{{ s = b[0] + ... + b[n-1] }}

Example: sum of array

Consider the following code to compute `b[0] + … + b[n-1]`:

```
{{ b.length >= n }}
s = 0;
i = 0;
{{ s = 0 and i = 0 }}
{{ Inv: s = b[0] + ... + b[i-1] }}
while (i != n) {
  s = s + b[i];
  i = i + 1;
}
{{ s = b[0] + ... + b[n-1] }}
```

Example: sum of array

Consider the following code to compute `b[0] + … + b[n-1]`:

{{ b.length >= n }}

```
s = 0;
i = 0;
```

{{ s = 0 and i = 0 }}

{{ Inv: s = b[0] + ... + b[i-1] }}

```
while (i != n) {
  s = s + b[i];
  i = i + 1;
}
```

{{ s = b[0] + ... + b[n-1] }}

- (s = 0 and i = 0) implies
  s = b[0] + … + b[i-1] ?

  Yes. (An empty sum is zero.)

Example: sum of array

Consider the following code to compute `b[0] + … + b[n-1]`:

{{ b.length >= n }}

```
s = 0;
i = 0;
```

{{ s = 0 and i = 0 }}

{{ Inv: s = b[0] + ... + b[i-1] }}

```
while (i != n) {
  s = s + b[i];
  i = i + 1;
}
```

{{ s = b[0] + ... + b[n-1] }}

- (s = 0 and i = 0) implies I

Example: sum of array

Consider the following code to compute `b[0] + … + b[n-1]`:

{{ b.length >= n }}
```
s = 0;
i = 0;
```
{{ Inv: s = b[0] + ... + b[i-1] }}
```
while (i != n) {
```
  {{ s = b[0] + ... + b[i-1] and i != n }}
```
  s = s + b[i];
  i = i + 1;
```
  {{ s = b[0] + ... + b[i-1] }}
```
}
```
{{ s = b[0] + ... + b[n-1] }}

- (s = 0 and i = 0) implies I
- {{ I and i != n }} S {{ I }} ?

Example: sum of array

Consider the following code to compute `b[0] + … + b[n-1]`:

{{ b.length >= n }}

```
s = 0;
i = 0;
```

{{ Inv: s = b[0] + ... + b[i-1] }}

```
while (i != n) {
```

{{ s = b[0] + ... + b[i-1] and i != n }}

```
  s = s + b[i];
  i = i + 1;
```

{{ s = b[0] + ... + b[i-1] }}

```
}
```

{{ s = b[0] + ... + b[n-1] }}

- (s = 0 and i = 0) implies I
- {{ I and i != n }} S {{ I }} ?

Yes (e.g., by backward reasoning)

{{ s + b[i] = b[0] + … + b[i] }}
{{ s = b[0] + … + b[i] }}

Example: sum of array

Consider the following code to compute `b[0] + … + b[n-1]`:

{{ b.length >= n }}

```
s = 0;
i = 0;
```

{{ Inv: s = b[0] + ... + b[i-1] }}

```
while (i != n) {
  s = s + b[i];
  i = i + 1;
}
```

{{ s = b[0] + ... + b[n-1] }}

- (s = 0 and i = 0) implies I
- {{ I and i != n }} S {{ I }}
- {{ I and i == n }} implies
  s = b[0] + … + b[n-1] ?

  Yes. (I is the postcondition when we have i == n.)

Example: sum of array

Consider the following code to compute `b[0] + … + b[n-1]`:

{{ b.length >= n }}

```
s = 0;
i = 0;
```

{{ Inv: s = b[0] + ... + b[i-1] }}

```
while (i != n) {
  s = s + b[i];
  i = i + 1;
}
```

{{ s = b[0] + ... + b[n-1] }}

- (s = 0 and i = 0) implies I
- {{ I and i != n }} S {{ I }}
- {{ I and i == n }} implies Q

These three checks verify that the postcondition holds (i.e., the code is correct).

Termination

- Technically, this analysis does not check that the code **terminates**
  - it shows that the postcondition holds if the loop exits
  - but we never showed that the loop actually exits

- However, that follows from an analysis of the running time
  - e.g., if the code runs in $O(n^2)$ time, then it terminates
  - an infinite loop would be O(infinity)
  - any finite bound on the running time proves it terminates

- It is normal to also analyze the running time of code we write, so we get termination already from that analysis.

Example: sum of array (attempt 2)

Consider the following code to compute `b[0] + … + b[n-1]`:

{{ b.length >= n }}

```
s = 0;
i = -1;
while (i != n-1) {
  i = i + 1;
  s = s + b[i];
}
```

{{ s = b[0] + ... + b[n-1] }}

Example: sum of array (attempt 2)

Consider the following code to compute `b[0] + … + b[n-1]`:

{{ b.length >= n }}

```
s = 0;
i = -1;
```

{{ Inv: s = b[0] + ... + b[i] }}

```
while (i != n-1) {
  i = i + 1;
  s = s + b[i];
}
```

{{ s = b[0] + ... + b[n-1] }}

Example: sum of array (attempt 2)

Consider the following code to compute `b[0] + … + b[n-1]`:

{{ b.length >= n }}

```
s = 0;
i = -1;
```

{{ Inv: s = b[0] + ... + b[i] }}

```
while (i != n-1) {
   i = i + 1;
   s = s + b[i];
}
```

{{ s = b[0] + ... + b[n-1] }}

- (s = 0 and i = -1) implies I
  - as before
- {{ I and i != n-1 }} S {{ I }}
  - reason backward:

{{ s + b[i+1] = b[0] + ... + b[i+1] }}

{{ s + b[i] = b[0] + … + b[i] }}

- (I and i = n-1) implies Q
  - as before