CSE 331
Software Design & Implementation

Kevin Zatloukal

Summer 2016

Lecture 3 – Reasoning About Loops

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

Review: Straight-line Code

Forward & Backward Reasoning

Forward reasoning

{{ P }}
`S`
{{ ? }}

- P is what we know initially
- Work downward
- Determine what holds after `S` executes

Backward reasoning

{{ ? }}
`S`
{{ Q }}

- Q is what we want at the end
- Work upward
- Determine what must hold initially before `S` executes

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	Backward reasoning
{{ P }}	{{ ? }}
`S1`	`S1`
{{ P1 }}	{{ ? }}
`S2`	`S2`
{{ P2 }}	{{ 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

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

Backward reasoning

```
{{ ? }}
if (cond)
    {{ Q1 }}
    S1
    {{ Q }}
else
    {{ Q2 }}
    S2
    {{ Q }}
{{ 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

```
{{ Q1 and cond or
   Q2 and not cond }}
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 }}
{{ (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

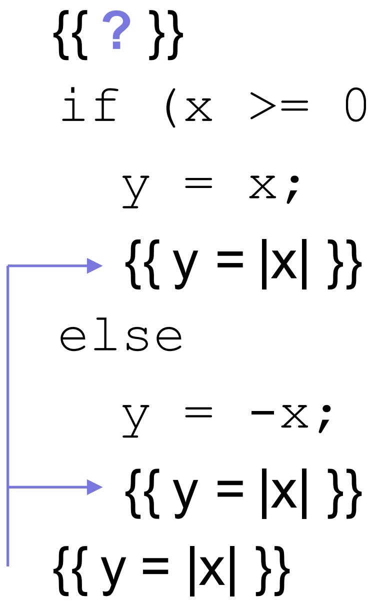
```
{{ ? }}
if (x >= 0)
   y = x;
else
   y = -x;
{{ y = |x| }}
```

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

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.

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 so 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 so 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 so 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 so 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: should almost always document your loop invariants
  - 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

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

Example: sum of array (attempt 3)

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) {
  i = i + 1;
  s = s + b[i];
}
```

{{ s = b[0] + … + b[n-1] }}

Suppose we use i != n instead of i != n-1…

We can spot this bug because the postcondition no longer follows.

When i = n, we get:

s = b[0] + … + b[n]

which is wrong

Example: sum of array (attempt 4)

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

Suppose we misorder the assignments to `i` and `s`…

We can spot this bug because the invariant does not hold:

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

First assertion is not I.

Example: max of array

Consider the following code to compute `max(b[0], …, b[n-1])`:

{{ b.length >= n and n > 0 }}

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

{{ m = max(b[0], …, b[n-1]) }}

Example: max of array

Consider the following code to compute `max(b[0], …, b[n-1])`:

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

Example: max of array

Consider the following code to compute `max(b[0], …, b[n-1])`:

{{ b.length >= n and n > 0 }}

```
m = b[0];
i = 1;
```

{{ Inv: m = max(b[0], …, b[i-1]) }}

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

{{ m = max(b[0], …, b[n-1]) }}

- I holds initially: m = max(b[0])
- Postcondition follows from invariant and i = n.
- Remains to check loop body…

Example: max of array

Consider the following code to compute `max(b[0], …, b[n-1])`:

```
{{ Inv: m = max(b[0], …, b[i-1]) }}
while (i != n) {
   if (b[i] > m)
      m = b[i];
  {{ m = max(b[0], …, b[i]) }}
   i = i + 1;
  {{ m = max(b[0], …, b[i-1]) }}
}
```

Example: max of array

Consider the following code to compute `max(b[0], …, b[n-1])`:

```
{{ Inv: m = max(b[0], …, b[i-1]) }}
while (i != n) {
  if (b[i] > m)
    m = b[i];
  else
    ;
  {{ m = max(b[0], …, b[i]) }}
  i = i + 1;
}
```

Example: max of array

Consider the following code to compute `max(b[0], …, b[n-1])`:

```
{{ Inv: m = max(b[0], ..., b[i-1]) }}
while (i != n) {
  if (b[i] > m)
    m = b[i];
    {{ m = max(b[0], ..., b[i]) }}
  else
    ;
    {{ m = max(b[0], ..., b[i]) }}
  i = i + 1;
}
```

Example: max of array

Consider the following code to compute `max(b[0], …, b[n-1])`:

```
{{ Inv: m = max(b[0], …, b[i-1]) }}
while (i != n) {
  if (b[i] > m)
    {{ b[i] = max(b[0], …, b[i]) }}
    m = b[i];
    {{ m = max(b[0], …, b[i]) }}
  else
    {{ m = max(b[0], …, b[i]) }}
    ;
    {{ m = max(b[0], …, b[i]) }}
  i = i + 1;
}
```

Example: max of array

Consider the following code to compute `max(b[0], …, b[n-1])`:

```
{{ Inv: m = max(b[0], ..., b[i-1]) }}
while (i != n) {
  {{ (b[i] > m and b[i] = max(b[0], ..., b[i])) or
     (b[i] <= m and m = max(b[0], ..., b[i]) }}
  if (b[i] > m)
     {{ b[i] = max(b[0], ..., b[i]) }}
     m = b[i];
  else
     {{ m = max(b[0], ..., b[i]) }}
     ;
  i = i + 1;
}
```

check that I implies this…
(requires some thought)

Example: max of array

Consider the following code to compute `max(b[0], …, b[n-1])`:

{{ Inv: m = max(b[0], …, b[i-1]) }}

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

{{ m = max(b[0], …, b[i]) }}

```
  if (b[i] > m)
    m = b[i];
  else
    ;
  i = i + 1;
}
```

- invariant is preserved by the loop body

Example: max of array

Consider code to compute `indexOfMax(b[0], …, b[n-1])`:

```
{{ b.length >= n  and n > 0 }}
k = 0;
m = b[k];
i = 1;
while (i != n) {
  if (b[i] > m) {
    k = i;
    m = b[k];
  }
  i = i + 1;
}
{{ m = max(b[0], ..., b[n-1]) and m = b[k] }}
```

Example: max of array

Consider code to compute `indexOfMax(b[0], …, b[n-1])`:

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

Example: max of array

Consider code to compute `indexOfMax(b[0], …, b[n-1])`:

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

{{ m = b[0] and k = 0 and i = 1 }}
{{ m = max(b[0], …, b[0]) and m = b[k] }}

Example: max of array

Consider code to compute `indexOfMax(b[0], …, b[n-1])`:

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

- I holds initially

{{ m = max(b[0], …, b[i-1]) and m = b[k] and i = n }} implies

Example: max of array

Consider code to compute `indexOfMax(b[0], …, b[n-1])`:

{{ b.length >= n and n > 0 }}

```
k = 0;
m = b[k];
i = 1;
```

{{ Inv: m = max(b[0], ..., b[i-1]) and m = b[k] }}

```
while (i != n) {
  if (b[i] > m) {
    k = i;
    m = b[k];
  }
  i = i + 1;
}
```

{{ m = max(b[0], ..., b[n-1]) and m = b[k] }}

- I holds initially
- I and i = n implies postcondition

Example: max of array

Consider code to compute `indexOfMax(b[0], …, b[n-1])`:

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

- I holds initially
- I and i = n implies postcondition

Example: max of array

Consider code to compute `indexOfMax(b[0], …, b[n-1])`:

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

- I holds initially
- I and i = n implies postcondition

Example: max of array

Consider code to compute `indexOfMax(b[0], …, b[n-1])`:

```
{{ b.length >= n  and n > 0 }}
k = 0;
m = b[k];
i = 1;
{{ Inv: m = max(b[0], ..., b[i-1]) and m = b[k] }}
while (i != n) {
  if (b[i] > m) {
    k = i;
    m = b[k];
   {{ m = max(b[0], ..., b[i]) and m = b[k] }}
  } else {
   {{ m = max(b[0], ..., b[i]) and m = b[k] }}
  }
  i = i + 1;
}
```

- I holds initially
- I and i = n implies postcondition

Example: max of array

Consider code to compute `indexOfMax(b[0], …, b[n-1])`:

```
{{ b.length >= n  and n > 0 }}
k = 0;
m = b[k];
i = 1;
{{ Inv: m = max(b[0], ..., b[i-1]) and m = b[k] }}
while (i != n) {
  if (b[i] > m) {
    k = i;
   {{ b[k] = max(b[0], ..., b[i]) and b[k] = b[k] }}
    m = b[k];
  } else {
   {{ m = max(b[0], ..., b[i]) and m = b[k] }}
  }
  i = i + 1;
}
```

- I holds initially
- I and i = n implies postcondition

Example: max of array

Consider code to compute `indexOfMax(b[0], …, b[n-1])`:

```
{{ b.length >= n  and n > 0 }}
k = 0;
m = b[k];
i = 1;
{{ Inv: m = max(b[0], ..., b[i-1]) and m = b[k] }}
while (i != n) {
  if (b[i] > m) {
    {{ b[i] = max(b[0], ..., b[i]) }}
    k = i;
    m = b[k];
  } else {
    {{ m = max(b[0], ..., b[i]) and m = b[k] }}
  }
  i = i + 1;
}
```

- I holds initially
- I and i = n implies postcondition

Example: max of array

Consider code to compute `indexOfMax(b[0], …, b[n-1])`:

```
{{ b.length >= n  and n > 0 }}
k = 0;
m = b[k];
i = 1;
{{ Inv: m = max(b[0], ..., b[i-1]) and m = b[k] }}
while (i != n) {
  {{ (b[i] > m) and b[i] = max(b[0], ..., b[i]) or
     (b[i] <= m) and m = max(b[0], ..., b[i]) and m = b[k] }}
  if (b[i] > m) {
    k = i;
    m = b[k];
  }
  i = i + 1;
}
```

- I holds initially
- I and i = n implies postcondition

Remains to show that I is stronger than this (i.e., I implies this):

- if b[i] > m = max(b[0], …, b[i-1]),
  then b[i] = max(b[0], …, b[i])
- if b[i] <= m = max(b[0], …, b[i-1]),
  then m = max(b[0], ..., b[i])

Example: max of array

Consider code to compute `indexOfMax(b[0], …, b[n-1])`:

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

- I holds initially
- I and i = n implies postcondition
- I holds after loop body

Example: partition array

Consider the following code to put the negative values at the beginning of array `b`:

```
{{ }}
i = k = 0;
while (i != n) {
  if (b[i] < 0) {
    swap b[i], b[k];
    k = k + 1;
  }
  i = i + 1;
}
{{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[n-1] }}
```

(Also: b contains the same numbers since we use swaps.)

Example: partition array

Consider the following code to put the negative values at the beginning of array `b`:

```
{{ }}
i = k = 0;
{{ Inv: b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] }}
while (i != n) {
  if (b[i] < 0) {
    swap b[i], b[k];
    k = k + 1;
  }
  i = i + 1;
}
{{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[n-1] }}
```

Example: partition array

Consider the following code to put the negative values at the beginning of array `b`:

```
{{ }}
i = k = 0;
{{ Inv: b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] }}
while (i != n) {
  if (b[i] < 0) {
    swap b[i], b[k];
    k = k + 1;
  }
  i = i + 1;
}
{{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[n-1] }}
```

- I holds initially:
  - b[0], …, b[-1] is empty
- I and i = n implies postcondition

Example: partition array

Consider the following code to put the negative values at the beginning of array `b`:

```
{{ }}
i = k = 0;
{{ Inv: b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] }}
while (i != n) {
  if (b[i] < 0) {
    swap b[i], b[k];
    k = k + 1;
  }
  i = i + 1;
}
{{ b[0], ..., b[k-1] < 0<= b[k], ..., b[n-1] }}
```

- I holds initially
- I and i = n implies postcondition

Example: partition array

Consider the following code to put the negative values at the beginning of array `b`:

```
{{ }}
i = k = 0;
{{ Inv: b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] }}
while (i != n) {
  if (b[i] < 0) {
    swap b[i], b[k];
    k = k + 1;
  }
  i = i + 1;
  {{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] }}
}
{{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[n-1] }}
```

- I holds initially
- I and i = n implies postcondition

Example: partition array

Consider the following code to put the negative values at the beginning of array `b`:

```
{{ }}
i = k = 0;
{{ Inv: b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] }}
while (i != n) {
  if (b[i] < 0) {
    swap b[i], b[k];
    k = k + 1;
  }
  {{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[i] }}
  i = i + 1;
}
{{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[n-1] }}
```

- I holds initially
- I and i = n implies postcondition

Example: partition array

Consider the following code to put the negative values at the beginning of array `b`:

```
{{ }}
i = k = 0;
{{ Inv: b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] }}
while (i != n) {
  {{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] }}
  if (b[i] < 0) {
    swap b[i], b[k];
    k = k + 1;
  }
  {{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[i] }}
  i = i + 1;
}
{{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[n-1] }}
```

- I holds initially
- I and i = n implies postcondition

Example: partition array

Consider the following code to put the negative values at the beginning of array `b`:

```
{{ }}
i = k = 0;
{{ Inv: b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] }}
while (i != n) {
  if (b[i] < 0) {
    {{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] and b[i] < 0 }}
    swap b[i], b[k];
    k = k + 1;
    {{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[i] }}
  } else {
    {{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] and b[i] >= 0 }}
    {{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[i] }}
  }
  i = i + 1;
}
```

- I holds initially
- I and i = n implies postcondition

Example: partition array

Consider the following code to put the negative values at the beginning of array `b`:

```
{{ }}
i = k = 0;
{{ Inv: b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] }}
while (i != n) {
  if (b[i] < 0) {
    {{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] and b[i] < 0 }}
    swap b[i], b[k];
    k = k + 1;
    {{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[i] }}
  } else {
    {{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] and b[i] >= 0 }}
    {{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[i] }}
  }
  i = i + 1;
}
```

equivalent

- I holds initially
- I and i = n implies postcondition

Example: partition array

Consider the following code to put the negative values at the beginning of array `b`:

```
{{ }}
i = k = 0;
{{ Inv: b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] }}
while (i != n) {
  if (b[i] < 0) {
    {{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] and b[i] < 0 }}
    swap b[i], b[k];
    k = k + 1;
    {{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[i] }}
  }
  i = i + 1;
}
{{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[n-1] }}
```

- I holds initially
- I and i = n implies postcondition

Remain to check this…

Example: partition array

Consider the following code to put the negative values at the beginning of array `b`:

```
{{ }}
i = k = 0;
{{ Inv: b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] }}
while (i != n) {
  if (b[i] < 0) {
    {{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] and b[i] < 0 }}
    swap b[i], b[k];
    k = k + 1;
    {{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[i] }}
  }
  i = i + 1;
}
{{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[n-1] }}
```

- I holds initially
- I and i = n implies postcondition

Example: partition array

Consider the following code to put the negative values at the beginning of array `b`:

```
{{ }}
i = k = 0;
{{ Inv: b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] }}
while (i != n) {
  if (b[i] < 0) {
    {{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] and b[i] < 0 }}
    swap b[i], b[k];
    {{ b[0], ..., b[k] < 0 <= b[k+1], ..., b[i] }}
    k = k + 1;
  }
  i = i + 1;
}
{{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[n-1] }}
```

- I holds initially
- I and i = n implies postcondition

This is a valid triple.
(Takes some thought.)

Example: partition array

Consider the following code to put the negative values at the beginning of array `b`:

```
{{ }}
i = k = 0;
{{ Inv: b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] }}
while (i != n) {
  if (b[i] < 0) {
    swap b[i], b[k];
    k = k + 1;
  }
  i = i + 1;
}
{{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[n-1] }}
```

- I holds initially
- I and i = n implies postcondition
- I holds after loop body