Dealing with Loops

19CSE205 : PROGRAM REASONING

Dr. Swaminathan J

Assistant Professor

Department of Computer Science and Engineering

AMRITA VISHWA VIDYAPEETHAM

Jul - Dec 2020

Contents

A simple looping program

Computing sum of first n integers.

File: sigma-loop.c

```
int sigma(int n) {
    int s = 0;
    int i = 1;
    while (i <= n) {
        s = s + i;
        i = i + 1;
    }
    return s;
}
```

A simple looping program

Computing sum of first n integers.

File: sigma-loop.c

```
/*@ requires n > 0;
    ensures \result == n*(n+1)/2;
*/
int sigma(int n) {
    int s = 0;
    int i = 1;
    while (i <= n) {
        s = s + i;
        i = i + 1;
    }
    return s;
}
```

A simple looping program

Computing sum of first n integers.

File: sigma-loop.c

```
/*@ requires n > 0;
    ensures \result == n*(n+1)/2;
*/
int sigma(int n) {
    int s = 0;
    int i = 1;
    while (i <= n) {
        s = s + i;
        i = i + 1;
    }
    return s;
}
```

```
prompt> frama-c -wp sigma-loop.c
[kernel] Parsing sigma-loop.c (with preprocessing)
[wp] warning: Missing RTE guards
sigma-loop.c:7:[wp] warning: Missing assigns clause (assigns 'everything' instead)
[wp] 1 goal scheduled
[wp] [Alt-Ergo] Goal typed_sigma_post : Unknown (Qed:4ms) (906ms)
[wp] Proved goals: 0 / 1
    Alt-Ergo: 0 (unknown: 1)
```

A simple looping program

Computing sum of first n integers.

File: sigma-loop.c

```
/*@ requires n > 0;
    ensures \result == n*(n+1)/2;
*/
int sigma(int n) {
    int s = 0;
    int i = 1;
    while (i <= n) {
        s = s + i;
        i = i + 1;
    }
    return s;
}
```

- Frama-c fails to prove.
- But we don't know why?
- Could it be because of loop?
- Let's first confirm. **Test**!

```
prompt> frama-c -wp sigma-loop.c
[kernel] Parsing sigma-loop.c (with preprocessing)
[wp] warning: Missing RTE guards
sigma-loop.c:7:[wp] warning: Missing assigns clause (assigns 'everything' instead)
[wp] 1 goal scheduled
[wp] [Alt-Ergo] Goal typed_sigma_post : Unknown (Qed:4ms) (906ms)
[wp] Proved goals: 0 / 1
    Alt-Ergo: 0 (unknown: 1)
```

Let's break down the loop

Computing the sum of first 3 integers. i.e. fixed n.

File: sigma-fixedn.c

Let's break down the loop

Computing the sum of first 3 integers. i.e. fixed n.

File: sigma-fixedn.c

```
int sigma(int n) {
    int s = 0, i = 1;
    s = s + i; i = i + 1;
    s = s + i; i = i + 1;
    s = s + i;
    return s;
}
```

Let's break down the loop

Computing the sum of first 3 integers. i.e. fixed n.

File: sigma-fixedn.c

```
/*@ requires n == 3;
    ensures \result == n*(n+1)/2;
*/
int sigma(int n) {
    int s = 0, i = 1;
    s = s + i; i = i + 1;
    s = s + i; i = i + 1;
    s = s + i;
    return s;
}
```

- Loop is re-written for fixed n.
- In this case n = 3.
- The underlying logic is same.
- Frama-c is able to prove the correctness now.
- Note the postcondition remains the same.

Computing the sum of first 3 integers. i.e. fixed n.

File: sigma-fixedn.c

```
/*@ requires n == 3;
    ensures \result == n*(n+1)/2;
*/
int sigma(int n) {
    int s = 0, i = 1;
    s = s + i; i = i + 1;
    s = s + i; i = i + 1;
    s = s + i;
    return s;
}
```

```
prompt> frama-c -wp sigma-fixedn.c
[kernel] Parsing sigma-fixedn.c (with preprocessing)
[wp] warning: Missing RTE guards
[wp] 1 goal scheduled
[wp] Proved goals: 1 / 1
    Qed: 1
```

- Loop is re-written for fixed n.
- In this case n = 3.
- The underlying logic is same.
- Frama-c is able to prove the correctness now.
- Note the postcondition remains the same.

Computing the sum of upto 3 integers. i.e. bounded n.

File: sigma-boundedn.c

Simulating bounded loop using if's

Computing the sum of upto 3 integers. i.e. bounded n.

File: sigma-boundedn.c

```
/*@ requires 1 <= n <= 3;
    ensures \result == n*(n+1)/2;
*/
int sigma(int n) {
    int i = 1, s = 0;
    if (i <= n) { s = s + i; i = i + 1; }
    if (i <= n) { s = s + i; i = i + 1; }
    if (i <= n) { s = s + i; }
    return s;
}
```

Computing the sum of upto 3 integers. i.e. bounded n.

File: sigma-boundedn.c

```
/*@ requires 1 <= n <= 3;
    ensures \result == n*(n+1)/2;
*/
int sigma(int n) {
    int i = 1, s = 0;
    if (i <= n) { s = s + i; i = i + 1; }
    if (i <= n) { s = s + i; i = i + 1; }
    if (i <= n) { s = s + i; }
    return s;
}
```

- Loop is re-written for a bounded n.
- In this case n <= 3.
- The underlying logic is same.
- Frama-c is able to prove the correctness again.

Computing the sum of upto 3 integers. i.e. bounded n.

File: sigma-boundedn.c

```
/*@ requires 1 <= n <= 3;
    ensures \result == n*(n+1)/2;
*/
int sigma(int n) {
    int i = 1, s = 0;
    if (i <= n) { s = s + i; i = i + 1; }
    if (i <= n) { s = s + i; i = i + 1; }
    if (i <= n) { s = s + i; }
    return s;
}
```

```
prompt> frama-c -wp sigma-boundedn.c
[kernel] Parsing sigma-boundedn.c (with preprocessing)
[wp] warning: Missing RTE guards
[wp] 1 goal scheduled
[wp] Proved goals: 1 / 1
    Qed: 0 (20ms)
    Alt-Ergo: 1 (21ms) (16)
```

- Loop is re-written for a bounded n.
- In this case n <= 3.
- The underlying logic is same.
- Frama-c is able to prove the correctness again.

Computing the sum of upto 3 integers. i.e. bounded n.

File: sigma-boundedn.c

```
/*@ requires 1 <= n <= 3;
    ensures \result == n*(n+1)/2;
*/
int sigma(int n) {
    int i = 1, s = 0;
    if (i <= n) { s = s + i; i = i + 1; }
    if (i <= n) { s = s + i; i = i + 1; }
    if (i <= n) { s = s + i; }
    return s;
}
```

```
prompt> frama-c -wp sigma-boundedn.c
[kernel] Parsing sigma-boundedn.c (with preprocessing)
[wp] warning: Missing RTE guards
[wp] 1 goal scheduled
[wp] Proved goals: 1 / 1
    Qed: 0 (20ms)
    Alt-Ergo: 1 (21ms) (16)
```

- Loop is re-written for a bounded n.
- In this case n <= 3.
- The underlying logic is same.
- Frama-c is able to prove the correctness again.

Deduction seems to breakdown in the presence of loops.
Two problems are evident.

How can one be sure the contract will be satsified for any n?

AMRITA VISHWA VIDYAPEETHAM

How can one be sure the contract will be satsified for any n?

- Weakest precondition calculus works backward, statement-by-statement.

```
while ( x < n ) {
    x = x + 1;
}
```

1. Loops are unbounded

How can one be sure the contract will be satsified for any n?

- Weakest precondition calculus works backward, statement-by-statement.

```
while ( x < n ) {
    x = x + 1;
}
```

- During execution, the loop may be iterated zero or more times.
- The question is how many times must the backward deduction be pushed through the loop?

1. Loops are unbounded

How can one be sure the contract will be satsified for any n?

- Weakest precondition calculus works backward, statement-by-statement.

```
while ( x < n ) {
    x = x + 1;
}
```

- During execution, the loop may be iterated zero or more times.
- The question is how many times must the backward deduction be pushed through the loop?

P': x<2 Q': x<5

Does P' $\Rightarrow$ Q'?

How can one be sure the contract will be satsified for any n?

- Weakest precondition calculus works backward, statement-by-statement.

```
while ( x < n ) {
    x = x + 1;
}
```

- During execution, the loop may be iterated zero or more times.
- The question is how many times must the backward deduction be pushed through the loop?

P': x<2 Q': x<5

Does P' $\Rightarrow$ Q'?

Q = Q'

WP start

1. Loops are unbounded

How can one be sure the contract will be satsified for any n?

- Weakest precondition calculus works backward, statement-by-statement.

```
while ( x < n ) {
    x = x + 1;
}
```

- During execution, the loop may be iterated zero or more times.
- The question is how many times must the backward deduction be pushed through the loop?

P': x<2 Q': x<5

Does P' $\Rightarrow$ Q'?

$x < 5$

x<2 $\Rightarrow$ x< 5? ✓

Q = Q'

WP start

How can one be sure the contract will be satsified for any n?

- Weakest precondition calculus works backward, statement-by-statement.

```
while ( x < n ) {
    x = x + 1;
}
```

- During execution, the loop may be iterated zero or more times.
- The question is how many times must the backward deduction be pushed through the loop?

P': x<2 Q': x<5 Does P' $\Rightarrow$ Q'?

$x < 4$	x<2 $\Rightarrow$ x< 4? ✓
$\boxed{x = x + 1;}$	$\uparrow$
$x < 5$	x<2 $\Rightarrow$ x< 5? ✓
Q = Q'	WP start

How can one be sure the contract will be satsified for any n?

- Weakest precondition calculus works backward, statement-by-statement.

```
while ( x < n ) {
    x = x + 1;
}
```

- During execution, the loop may be iterated zero or more times.
- The question is how many times must the backward deduction be pushed through the loop?

P': $x<2$ Q': $x<5$

Does P' $\Rightarrow$ Q'?

$x < 3$	$x<2 \Rightarrow x< 3?$ ✓
$x = x + 1;$	$\uparrow$
$x < 4$	$x<2 \Rightarrow x< 4?$ ✓
$x = x + 1;$	$\uparrow$
$x < 5$	$x<2 \Rightarrow x< 5?$ ✓
Q = Q'	WP start

How can one be sure the contract will be satsified for any n?

- Weakest precondition calculus works backward, statement-by-statement.

```
while ( x < n ) {
    x = x + 1;
}
```

- During execution, the loop may be iterated zero or more times.
- The question is how many times must the backward deduction be pushed through the loop?

P': x<2 Q': x<5

Does P' $\Rightarrow$ Q'?

$x < 2$	x<2 $\Rightarrow$ x< 2? ✓
$x = x + 1;$	$\uparrow$
$x < 3$	x<2 $\Rightarrow$ x< 3? ✓
$x = x + 1;$	$\uparrow$
$x < 4$	x<2 $\Rightarrow$ x< 4? ✓
$x = x + 1;$	$\uparrow$
$x < 5$	x<2 $\Rightarrow$ x< 5? ✓
Q = Q'	WP start

How can one be sure the contract will be satsified for any n?

- Weakest precondition calculus works backward, statement-by-statement.

```
while ( x < n ) {
    x = x + 1;
}
```

- During execution, the loop may be iterated zero or more times.
- The question is how many times must the backward deduction be pushed through the loop?

P': x<2 Q': x<5	Does P' ⇒ Q'?
$x < 1$	$x<2 \Rightarrow x< 1$? X
$x = x + 1;$	$\uparrow$
$x < 2$	$x<2 \Rightarrow x< 2$? ✓
$x = x + 1;$	$\uparrow$
$x < 3$	$x<2 \Rightarrow x< 3$? ✓
$x = x + 1;$	$\uparrow$
$x < 4$	$x<2 \Rightarrow x< 4$? ✓
$x = x + 1;$	$\uparrow$
$x < 5$	$x<2 \Rightarrow x< 5$? ✓
Q = Q'	WP start

2. Loop may run forever

What is the guarantee that the loop will eventually terminate?

2. Loop may run forever

What is the guarantee that the loop will eventually terminate?

```
while ( i < n ) {
    ......
    ......
    i = i + 1
}
```

- i never gets incremented.

What is the guarantee that the loop will eventually terminate?

```
while ( i < n ) {
    ......
    ......
    ~~i = i + 1~~
}
```

```
while ( i != n ) {
    ......
    i = i + 1
    n = n + 1
}
```

- i never gets incremented.
- n increases along with i.

What is the guarantee that the loop will eventually terminate?

```
while ( i < n ) {
    ......
    ......
    ~~i = i + 1~~
}
```

```
while ( i != n ) {
    ......
    i = i + 1
    n = n + 1
}
```

```
i = 1
while ( i != 10 ) {
    ......
    i = i + 2
}
```

- i never gets incremented.
- n increases along with i.
- i will never take a value of 10.

What is the guarantee that the loop will eventually terminate?

```
while ( i < n ) {
    ......
    ......
    i = i + 1
}
```

```
while ( i != n ) {
    ......
    i = i + 1
    n = n + 1
}
```

```
i = 1
while ( i != 10 ) {
    ......
    i = i + 2
}
```

- i never gets incremented.
- n increases along with i.
- i will never take a value of 10.

Bottomline:

- The programmers can write their code in any manner.
- They can state the input and output conditions in any way.
- The proof system must not make any assumptions about the code.
- Proof construction must be based on generic principles.

Weakest precondition for while loop

Weakest precondition asks the user to provide a magic property that will serve as both pre- and post-condition for the loop. It will check if this property is satisfied each time the while condition is evaluated.

Weakest precondition asks the user to provide a magic property that will serve as both pre- and post-condition for the loop. It will check if this property is satisfied each time the while condition is evaluated.

while B do
S

Q

Weakest precondition asks the user to provide a magic property that will serve as both pre- and post-condition for the loop. It will check if this property is satisfied each time the while condition is evaluated.

- This magic property is called loop invariant.

  i.e. $I = wp(S,I)$ where I is the loop invariant.

```
while B do
  S
```

I

Q

Weakest precondition asks the user to provide a magic property that will serve as both pre- and post-condition for the loop. It will check if this property is satisfied each time the while condition is evaluated.

- This magic property is called loop invariant.

  i.e. I = wp(S,I) where I is the loop invariant.
- The loop body S is executed only if B is true.

```
while B do
  S
```

I

Q

Weakest precondition asks the user to provide a magic property that will serve as both pre- and post-condition for the loop. It will check if this property is satisfied each time the while condition is evaluated.

- This magic property is called loop invariant.

  i.e. $I = wp(S,I)$ where I is the loop invariant.
- The loop body S is executed only if B is true.

  1. $B \Rightarrow S$ is equivalent to $B \wedge I \Rightarrow S$

while B do
S

I

Q

Weakest precondition asks the user to provide a magic property that will serve as both pre- and post-condition for the loop. It will check if this property is satisfied each time the while condition is evaluated.

- This magic property is called loop invariant.

  i.e. $I = wp(S,I)$ where I is the loop invariant.

- The loop body S is executed only if B is true.

  1. $B \Rightarrow S$ is equivalent to $B \wedge I \Rightarrow S$
  2. $\neg B \Rightarrow Q$ is equivalent to $\neg B \wedge I \Rightarrow Q$

while B do
S

I

Q

Weakest precondition asks the user to provide a magic property that will serve as both pre- and post-condition for the loop. It will check if this property is satisfied each time the while condition is evaluated.

- This magic property is called loop invariant.

  i.e. $I = wp(S,I)$ where I is the loop invariant.

- The loop body S is executed only if B is true.

  1. $B \Rightarrow S$ is equivalent to $B \wedge I \Rightarrow S$
  2. $\neg B \Rightarrow Q$ is equivalent to $\neg B \wedge I \Rightarrow Q$

while B do
S

I

Q

- wp(while B do S,Q)

Weakest precondition asks the user to provide a magic property that will serve as both pre- and post-condition for the loop. It will check if this property is satisfied each time the while condition is evaluated.

- This magic property is called loop invariant.

  i.e. $I = wp(S,I)$ where I is the loop invariant.

- The loop body S is executed only if B is true.

  1. $B \Rightarrow S$ is equivalent to $B \wedge I \Rightarrow S$
  2. $\neg B \Rightarrow Q$ is equivalent to $\neg B \wedge I \Rightarrow Q$

while B do
S

I

Q

- wp(while B do S,Q)

  $=$ wp(while $B \wedge I$ do S,Q)

Weakest precondition asks the user to provide a magic property that will serve as both pre- and post-condition for the loop. It will check if this property is satisfied each time the while condition is evaluated.

- This magic property is called loop invariant.

  i.e. $I = wp(S,I)$ where I is the loop invariant.

- The loop body S is executed only if B is true.

  1. $B \Rightarrow S$ is equivalent to $B \wedge I \Rightarrow S$
  2. $\neg B \Rightarrow Q$ is equivalent to $\neg B \wedge I \Rightarrow Q$

while B do
S

I

Q

- wp(while B do S,Q)

$$= wp(\text{while } B \wedge I \text{ do } S, Q)$$
$$= B \wedge I \Rightarrow wp(S,I) \bigwedge \neg B \wedge I \Rightarrow Q$$

Weakest precondition asks the user to provide a magic property that will serve as both pre- and post-condition for the loop. It will check if this property is satisfied each time the while condition is evaluated.

- This magic property is called loop invariant.

  i.e. $I = wp(S,I)$ where I is the loop invariant.

- The loop body S is executed only if B is true.
  1. $B \Rightarrow S$ is equivalent to $B \wedge I \Rightarrow S$
  2. $\neg B \Rightarrow Q$ is equivalent to $\neg B \wedge I \Rightarrow Q$

- wp(while B do S,Q)

$$= wp(\text{while } B \wedge I \text{ do } S,Q)$$
$$= B \wedge I \Rightarrow wp(S,I) \bigwedge \neg B \wedge I \Rightarrow Q$$

while B do
S

I

Q

How do we come up with this loop invariant?

Any thumb rules?

Weakest precondition asks the user to provide a magic property that will serve as both pre- and post-condition for the loop. It will check if this property is satisfied each time the while condition is evaluated.

- This magic property is called loop invariant.

  i.e. $I = wp(S,I)$ where I is the loop invariant.

- The loop body S is executed only if B is true.

  1. $B \Rightarrow S$ is equivalent to $B \wedge I \Rightarrow S$
  2. $\neg B \Rightarrow Q$ is equivalent to $\neg B \wedge I \Rightarrow Q$

while B do
S

I

Q

- wp(while B do S,Q)

$$= wp(\text{while } B \wedge I \text{ do } S,Q)$$
$$= B \wedge I \Rightarrow wp(S,I) \bigwedge \neg B \wedge I \Rightarrow Q$$

How do we come up with this loop invariant?

Any thumb rules?

Loop invariant must capture the progress made as iterations proceed.

Weakest precondition asks the user to provide a magic property that will serve as both pre- and post-condition for the loop. It will check if this property is satisfied each time the while condition is evaluated.

- This magic property is called loop invariant.

  i.e. I = wp(S,I) where I is the loop invariant.

- The loop body S is executed only if B is true.

  1. $B \Rightarrow S$ is equivalent to $B \wedge I \Rightarrow S$
  2. $\neg B \Rightarrow Q$ is equivalent to $\neg B \wedge I \Rightarrow Q$

while B do
S

I

Q

- wp(while B do S,Q)

  $= $ wp(while $B \wedge I$ do S,Q)

  $= B \wedge I \Rightarrow wp(S,I) \bigwedge \neg B \wedge I \Rightarrow Q$

How do we come up with this loop invariant?

Any thumb rules?

Loop invariant must capture the progress made as iterations proceed.

Loop invariant must capture the span of entry and exit condition range.

Let's apply this to the example

File: sigma-loop.c

```
/*@ requires n > 0;
    ensures \result == n*(n+1)/2;
*/
int sigma(int n) {
    int s = 0;
    int i = 1;
    /*@



    */
    while (i <= n) {
        s = s + i;
        i = i + 1;
    }
    return s;
}
```

Loop invariant must capture the progress made as iterations proceed.

Loop invariant must capture the span of entry and exit condition range.

File: sigma-loop.c

```
/*@ requires n > 0;
    ensures \result == n*(n+1)/2;
*/
int sigma(int n) {
    int s = 0;
    int i = 1;
    /*@



    */
    while (i <= n) {
        s = s + i;
        i = i + 1;
    }
    return s;
}
```

1. Capturing progress

Evaluation of while entry condition	i	s
Before iteration 1	1	0
Before iteration 2	2	1
Before iteration 3	3	3
Before iteration 4	4	6
...	..	..
...	..	..
Before iteration n	n	(n-1)*n/2
After iteration n	n+1	n*(n+1)/2

Loop invariant must capture the progress made as iterations proceed.

Loop invariant must capture the span of entry and exit condition range.

Let's apply this to the example

File: sigma-loop.c

```
/*@ requires n > 0;
    ensures \result == n*(n+1)/2;
*/
int sigma(int n) {
    int s = 0;
    int i = 1;
    /*@



    */
    while (i <= n) {
        s = s + i;
        i = i + 1;
    }
    return s;
}
```

1. Capturing progress

Evaluation of while entry condition	i	s
Before iteration 1	1	0
Before iteration 2	2	1
Before iteration 3	3	3
Before iteration 4	4	6
...	..	..
...	..	..
Before iteration n	n	(n-1)*n/2
After iteration n	n+1	n*(n+1)/2

Progress made is captured by (i−1)*i/2.

Loop invariant must capture the progress made as iterations proceed.

Loop invariant must capture the span of entry and exit condition range.

Let's apply this to the example

File: sigma-loop.c

```
/*@ requires n > 0;
    ensures \result == n*(n+1)/2;
*/
int sigma(int n) {
    int s = 0;
    int i = 1;
    /*@
      loop invariant s == (i−1)*i/2;


    */
    while (i <= n) {
        s = s + i;
        i = i + 1;
    }
    return s;
}
```

1. Capturing progress

Evaluation of while entry condition	i	s
Before iteration 1	1	0
Before iteration 2	2	1
Before iteration 3	3	3
Before iteration 4	4	6
...	..	..
...	..	..
Before iteration n	n	(n-1)*n/2
After iteration n	n+1	n*(n+1)/2

Progress made is captured by (i−1)*i/2.

Loop invariant must capture the progress made as iterations proceed.

Loop invariant must capture the span of entry and exit condition range.

File: sigma-loop.c

```
/*@ requires n > 0;
    ensures \result == n*(n+1)/2;
*/
int sigma(int n) {
    int s = 0;
    int i = 1;
    /*@
      loop invariant s == (i-1)*i/2;


    */
    while (i <= n) {
        s = s + i;
        i = i + 1;
    }
    return s;
}
```

1. Capturing progress

Evaluation of while entry condition	i	s
Before iteration 1	1	0
Before iteration 2	2	1
Before iteration 3	3	3
Before iteration 4	4	6
...	..	..
...	..	..
Before iteration n	n	(n-1)*n/2
After iteration n	n+1	n*(n+1)/2

Progress made is captured by (i−1)*i/2.

2. Capturing the entry & exit range

- Entry condition: i ranges from 1 to n
- Exit condition: i takes the value n+1

Loop invariant must capture the progress made as iterations proceed.

Loop invariant must capture the span of entry and exit condition range.

File: sigma-loop.c

```
/*@ requires n > 0;
    ensures \result == n*(n+1)/2;
*/
int sigma(int n) {
    int s = 0;
    int i = 1;
    /*@
      loop invariant s == (i-1)*i/2;


    */
    while (i <= n) {
        s = s + i;
        i = i + 1;
    }
    return s;
}
```

Loop invariant must capture the progress made as iterations proceed.

1. Capturing progress

Evaluation of while entry condition	i	s
Before iteration 1	1	0
Before iteration 2	2	1
Before iteration 3	3	3
Before iteration 4	4	6
...	..	..
...	..	..
Before iteration n	n	(n-1)*n/2
After iteration n	n+1	n*(n+1)/2

Progress made is captured by (i−1)*i/2.

2. Capturing the entry & exit range

- Entry condition: i ranges from 1 to n
- Exit condition: i takes the value n+1

Combining, we get 1 <= i <= n+1.

Loop invariant must capture the span of entry and exit condition range.

File: sigma-loop.c

```
/*@ requires n > 0;
    ensures \result == n*(n+1)/2;
*/
int sigma(int n) {
    int s = 0;
    int i = 1;
    /*@
      loop invariant s == (i−1)*i/2;
      loop invariant 1 <= i <= n+1;

    */
    while (i <= n) {
        s = s + i;
        i = i + 1;
    }
    return s;
}
```

Loop invariant must capture the progress made as iterations proceed.

1. Capturing progress

Evaluation of while entry condition	i	s
Before iteration 1	1	0
Before iteration 2	2	1
Before iteration 3	3	3
Before iteration 4	4	6
...	..	..
...	..	..
Before iteration n	n	(n-1)*n/2
After iteration n	n+1	n*(n+1)/2

Progress made is captured by (i−1)*i/2.

2. Capturing the entry & exit range

- Entry condition: i ranges from 1 to n
- Exit condition: i takes the value n+1

Combining, we get 1 <= i <= n+1.

Loop invariant must capture the span of entry and exit condition range.

Let's apply this to the example

File: sigma-loop.c

```
/*@ requires n > 0;
    ensures \result == n*(n+1)/2;
*/
int sigma(int n) {
    int s = 0;
    int i = 1;
    /*@
      loop invariant s == (i-1)*i/2;
      loop invariant 1 <= i <= n+1;
      loop assigns s, i;
    */
    while (i <= n) {
        s = s + i;
        i = i + 1;
    }
    return s;
}
```

Loop invariant must capture the progress made as iterations proceed.

1. Capturing progress

Evaluation of while entry condition	i	s
Before iteration 1	1	0
Before iteration 2	2	1
Before iteration 3	3	3
Before iteration 4	4	6
...	..	..
...	..	..
Before iteration n	n	(n-1)*n/2
After iteration n	n+1	n*(n+1)/2

Progress made is captured by (i−1)*i/2.

2. Capturing the entry & exit range

- Entry condition: i ranges from 1 to n
- Exit condition: i takes the value n+1

Combining, we get 1 <= i <= n+1.

Loop invariant must capture the span of entry and exit condition range.

The loop invariant will help prove partial corrrectness of programs.

- Partial correctness: The correctness criteria will be met if the loop would terminate.
- Total correctness: The program is guaranteed to terminate and the correctness criteria will be met.

Partial vs. Total correctness

The loop invariant will help prove partial corrrectness of programs.

- Partial correctness: The correctness criteria will be met if the loop would terminate.
- Total correctness: The program is guaranteed to terminate and the correctness criteria will be met.

Proving termination

- To prove termination, one has to specify a non-negative expression that will decrease as the while loop executes and eventually becomes 0.
- In our example, since i increases, the expression $n - i$ decreases. In ACSL, this is specified using the annotation loop variant $n - i$.
- At most one loop variant clause is allowed.

The loop invariant will help prove partial corrrectness of programs.

- Partial correctness: The correctness criteria will be met if the loop would terminate.
- Total correctness: The program is guaranteed to terminate and the correctness criteria will be met.

Proving termination

- To prove termination, one has to specify a non-negative expression that will decrease as the while loop executes and eventually becomes 0.
- In our example, since i increases, the expression n − i decreases. In ACSL, this is specified using the annotation loop variant n − i.
- At most one loop variant clause is allowed.

File: sigma-loop.c

```
/*@ requires n > 0;
    ensures \result == n*(n+1)/2;
*/
int sigma(int n) {
    int s = 0;
    int i = 1;
    /*@
      loop invariant s == (i-1)*i/2;
      loop invariant 1 <= i <= n+1;
      loop assigns s, i;
      loop variant n - i;
    */
    while (i <= n) {
        s = s + i;
        i = i + 1;
    }
    return s;
}
```

AMRITA VISHWA VIDYAPEETHAM

Apply these variations to sigma program to improve your understanding.

1. Remove //@ loop invariant s == (i−1)*i/2;
2. Remove loop invariant 1 <= i <= n+1;
3. Replace (i−1)*i/2 with i*(i+1)/2 in first loop invariant.
4. Replace loop invariant 1 <= i <= n+1; with loop invariant i <= n+1;
5. Replace loop invariant 1 <= i <= n+1; with loop invariant 1 <= i <= n;
6. Do as in bullet 4. In addition, replace while (i <= n) with while (i < n).
7. Remove the statement i = i + 1;
8. Replace loop invariant 1 <= i <= n+1; with loop invariant 1 <= i <= n+2;. Now, modify your program such that criteria is met but program is wrong.
9. Re-write the while loop to iterate in reverse way. i.e. n + (n−1) + . . . + 1. What changes would you have to make to prove all goals?

Follow these instructions when you try the variations.

- Implement one variation at a time and reason out the frama-c output.
- Run frama-c-gui -wp ⟨program⟩ to see which goal cannot be proved.
- Don't make silly errors and waste time resolving them. Focus on checking logic.