

CS:5810 Formal Methods in Software Engineering

Reasoning about Loops in Dafny

Copyright 2020-21, Graeme Smith and Cesare Tinelli.

Produced by Cesare Tinelli at the University of Iowa from notes originally developed by Graeme Smith at the University of Queensland. These notes are copyrighted materials and may not be used in other course settings outside of the University of Iowa in their current form or modified form without the express written permission of one of the copyright holders. During this course, students are prohibited from selling notes to or being paid for taking notes by any person or commercial firm without the express written permission of one of the copyright holders.

Loops in Dafny

```
while G
  decreases M
  invariant J
{
  Body
}
```

G: *loop guard*, Boolean expression

M: *termination measure*, expression whose value is expected to decrease at each loop iteration

J: *loop invariant*, condition expected to hold at each iteration

Loops in Dafny

```
while G
  decreases M
  invariant J
{
  Body
}
```

While loops are *opaque*: they are *always* abstracted by their invariant

...

```
while G
  invariant J
```

...

Loop specification examples

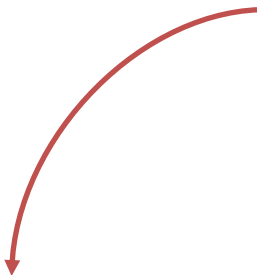
```
while x < 300  
  invariant x % 2 == 0
```

```
while x % 2 == 1  
  invariant 0 <= x <= 100
```

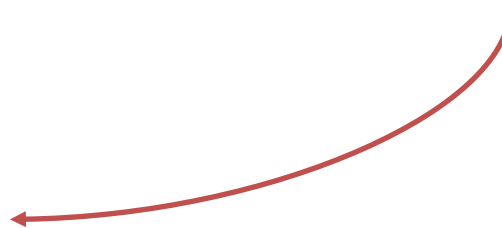
```
x := 2;  
while x < 50  
  invariant x % 2 == 0  
assert x >= 50 && x % 2 == 0;
```

```
x := 0;  
while x % 2 == 0  
  invariant 0 <= x <= 20  
assert x == 19; // not provable
```

After loop we know that the invariant and negation of the guard hold



Would need to prove

$$0 \leq x \leq 20 \ \&\& \ x \% 2 \neq 0 \implies x == 19$$


Attaining equality

```
i := 0;  
while i != 100  
  invariant 0 <= i <= 100  
assert i == 100;
```

Assertion is provable from just the negation of the guard

```
i := 0;  
while i < 100  
  invariant 0 <= i <= 100  
assert i == 100;
```

Assertion requires the invariant **and** the negation of the guard to hold

Attaining equality

```
i := 0;  
while i != 100  
  invariant true  
assert i == 100;
```

Assertion is provable from just the negation of the guard

```
i := 0;  
while i < 100  
  invariant true  
assert i == 100; // not provable
```

Assertion requires the invariant and the negation of the guard to hold

Relations between variables

```
x, y := 0, 0;  
while x < 300  
    invariant 2 * x == 3 * y  
assert 200 <= y;
```

```
x, y := 0, 191;  
while !(0 <= y < 7)  
    invariant 7 * x + y == 191  
assert x == 191 / 7 && y == 191 % 7;
```

```
n, s := 0, 0;  
while n != 33  
    invariant s == n * (n - 1) / 2
```

Relations between variables

```
x, y := 0, 0;  
while x < 300  
    invariant 2 * x == 3 * y  
assert 200 <= y;
```

```
x, y := 0, 191;  
while !(y < 7)  
    invariant 0 <= y && 7 * x + y == 191  
assert x == 191 / 7 && y == 191 % 7;
```

```
n, s := 0, 0;  
while n != 33  
    invariant s == n * (n - 1) / 2
```


Hoare triples for loops

```
{ J }  
while G  
    invariant J  
{ J && !G }
```

Example

```
r := 0;  
N := 104;  
while (r+1)*(r+1) <= N  
    invariant 0 <= r && r*r <= N  
assert 0 <= r && r*r <= N < (r+1)*(r+1);
```

Floyd-Hoare logic for loop body

For a loop

```
while G
  invariant J
  {
    Body
  }
```

we need to prove

```
{ J && G }
Body
{ J }
```

Quotient modulus

```
x := 0;  
y := 191;  
while !(y < 7)  
    invariant 0 <= y && 7*x + y == 191  
assert x == 191 / 7 && y == 191 % 7;
```

Quotient modulus

```
x := 0;
y := 191;
while !(y < 7)
    invariant 0 <= y && 7*x + y == 191
{

}
assert x == 191 / 7 && y == 191 % 7;
```

Quotient modulus

```
x := 0;
y := 191;
while !(y < 7)
  invariant 0 <= y && 7*x + y == 191
  {
    { 0 <= y && 7*x + y == 191 && 7 <= y }

    { 0 <= y && 7*x + y == 191 }
  }
assert x == 191 / 7 && y == 191 % 7;
```

Quotient modulus

```
x := 0;
y := 191;
while !(y < 7)
  invariant 0 <= y && 7*x + y == 191
  {
    { 0 <= y && 7*x + y == 191 && 7 <= y }

    x := x + 1;
    { 0 <= y && 7*x + y == 191 }
  }
assert x == 191 / 7 && y == 191 % 7;
```

Quotient modulus

```
x := 0;
y := 191;
while !(y < 7)
  invariant 0 <= y && 7*x + y == 191
  {
    { 0 <= y && 7*x + y == 191 && 7 <= y }

    { 0 <= y && 7*(x + 1) + y == 191 }
    x := x + 1;
    { 0 <= y && 7*x + y == 191 }
  }
assert x == 191 / 7 && y == 191 % 7;
```

Quotient modulus

```
x := 0;
y := 191;
while !(y < 7)
  invariant 0 <= y && 7*x + y == 191
  {
    { 0 <= y && 7*x + y == 191 && 7 <= y }

    { 0 <= y && 7*x + 7 + y == 191 }
    { 0 <= y && 7*(x + 1) + y == 191 }
    x := x + 1;
    { 0 <= y && 7*x + y == 191 }
  }
assert x == 191 / 7 && y == 191 % 7;
```


Quotient modulus

```
x := 0;
y := 191;
while !(y < 7)
  invariant 0 <= y && 7*x + y == 191
  {
    { 0 <= y && 7*x + y == 191 && 7 <= y }

    y := y - 7;
    { 0 <= y && 7*x + 7 + y == 191 }
    { 0 <= y && 7*(x + 1) + y == 191 }
    x := x + 1;
    { 0 <= y && 7*x + y == 191 }
  }
assert x == 191 / 7 && y == 191 % 7;
```

Full program

```
x := 0;
y := 191;
while !(y < 7)
  invariant 0 <= y && 7*x + y == 191
  {
    { 0 <= y && 7*x + y == 191 && 7 <= y }
    { 0 <= y - 7 && 7*x + 7 + (y - 7) == 191 }
    y := y - 7;
    { 0 <= y && 7*x + 7 + y == 191 }
    { 0 <= y && 7*(x + 1) + y == 191 }
    x := x + 1;
    { 0 <= y && 7*x + y == 191 }
  }
assert x == 191 / 7 && y == 191 % 7;
```

Leap to the answer

```
x := 0;
y := 191;
while !(y < 7)
  invariant 0 <= y && 7*x + y == 191
  {
    { 0 <= y && 7 * x + y == 191 && 7 <= y }

    x, y := 27, 2
    { 0 <= y && 7 * x + y == 191 }
  }
assert x == 191 / 7 && y == 191 % 7;
```

Leap to the answer

```
x := 0;
y := 191;
while !(y < 7)
  invariant 0 <= y && 7*x + y == 191
  {
    { 0 <= y && 7 * x + y == 191 && 7 <= y }
    { true }
    { 0 <= 2 && 7 * 27 + 2 == 191 }
    x, y := 27, 2
    { 0 <= y && 7 * x + y == 191 }
  }
assert x == 191 / 7 && y == 191 % 7;
```

Going twice as fast

Let's try incrementing x by **2** and decrementing y by **14**

```
{ 0 <= y && 7*x + y == 191 && 7 <= y }
```

```
x, y := x + 2, y - 14
```

```
{ 0 <= y && 7 * x + y == 191 }
```

Going twice as fast

Let's try incrementing x by **2** and decrementing y by **14**

```
{ 0 <= y && 7*x + y == 191 && 7 <= y }
```

```
{ 14 <= y && 7*x + y == 191 }
```

```
{ 0 <= y - 14 && 7*(x + 2) + (y - 14) == 191 }
```

```
x, y := x + 2, y - 14
```

```
{ 0 <= y && 7 * x + y == 191 }
```

Going twice as fast

Let's try incrementing x by 2 and decrementing y by 14

```
{ 0 <= y && 7*x + y == 191 && 7 <= y }
```

```
{ 14 <= y && 7*x + y == 191 }
```

```
{ 0 <= y - 14 && 7*(x + 2) + (y - 14) == 191 }
```

```
x, y := x + 2, y - 14
```

```
{ 0 <= y && 7 * x + y == 191 }
```

$14 \leq y$ does not follow from the top line

So this loop body would be incorrect

Computing sums

```
while n != 33
  invariant s == n * (n - 1) / 2
  {
    { s == n * (n - 1) / 2 && n != 33 }

    { s == n * (n - 1) / 2 }
  }
assert s == 33 * 32 / 2;
```


Computing sums

```
while n != 33
  invariant s == n * (n - 1) / 2
  {
    { s == n * (n - 1) / 2 && n != 33 }

    n := n + 1;
    { s == n * (n - 1) / 2 }
  }
assert s == 33 * 32 / 2;
```

Computing sums

```
while n != 33
  invariant s == n * (n - 1) / 2
  {
    { s == n * (n - 1) / 2 && n != 33 }

    { s == (n + 1) * (n + 1 - 1) / 2 }
    n := n + 1;
    { s == n * (n - 1) / 2 }
  }
assert s == 33 * 32 / 2;
```

Computing sums

```
while n != 33
  invariant s == n * (n - 1) / 2
  {
    { s == n * (n - 1) / 2 && n != 33 }

    { s == (n + 1) * n / 2 }
    { s == (n + 1) * (n + 1 - 1) / 2 }
    n := n + 1;
    { s == n * (n - 1) / 2 }
  }
assert s == 33 * 32 / 2;
```

Computing sums

```
while n != 33
  invariant s == n * (n - 1) / 2
  {
    { s == n * (n - 1) / 2 && n != 33 }

    { s == (n*n + n) / 2 }
    { s == (n + 1) * n / 2 }
    { s == (n + 1) * (n + 1 - 1) / 2 }
    n := n + 1;
    { s == n * (n - 1) / 2 }
  }
assert s == 33 * 32 / 2;
```

Computing sums

```
while n != 33
  invariant s == n * (n - 1) / 2
  {
    { s == n * (n - 1) / 2 && n != 33 }

    { s == (n*n - n + 2*n) / 2 }
    { s == (n*n + n) / 2 }
    { s == (n + 1) * n / 2 }
    { s == (n + 1) * (n + 1 - 1) / 2 }
    n := n + 1;
    { s == n * (n - 1) / 2 }
  }
assert s == 33 * 32 / 2;
```

Computing sums

```
while n != 33
  invariant s == n * (n - 1) / 2
  {
    { s == n * (n - 1) / 2 && n != 33 }

    { s = (n*n - n) / 2 + 2*n / 2 }
    { s == (n*n - n + 2*n) / 2 }
    { s == (n*n + n) / 2 }
    { s == (n + 1) * n / 2 }
    { s == (n + 1) * (n + 1 - 1) / 2 }
    n := n + 1;
    { s == n * (n - 1) / 2 }
  }
assert s == 33 * 32 / 2;
```

Computing sums

```
while n != 33
  invariant s == n * (n - 1) / 2
  {
    { s == n * (n - 1) / 2 && n != 33 }

    { s = n * (n - 1) / 2 + n }
    { s = (n*n - n) / 2 + 2*n / 2 }
    { s == (n*n - n + 2*n) / 2 }
    { s == (n*n + n) / 2 }
    { s == (n + 1) * n / 2 }
    { s == (n + 1) * (n + 1 - 1) / 2 }
    n := n + 1;
    { s == n * (n - 1) / 2 }
  }
assert s == 33 * 32 / 2;
```

Computing sums

```
while n != 33
  invariant s == n * (n - 1) / 2
  {
    { s == n * (n - 1) / 2 && n != 33 }

    s := s + n;
    { s = n * (n - 1) / 2 + n }
    { s = (n*n - n) / 2 + 2*n / 2 }
    { s == (n*n - n + 2*n) / 2 }
    { s == (n*n + n) / 2 }
    { s == (n + 1) * n / 2 }
    { s == (n + 1) * (n + 1 - 1) / 2 }
    n := n + 1;
    { s == n * (n - 1) / 2 }
  }
assert s == 33 * 32 / 2;
```


Computing sums

```
while n != 33
  invariant s == n * (n - 1) / 2
  {
    { s == n * (n - 1) / 2 && n != 33 }
    { s == n * (n - 1) / 2 }
    s := s + n;
    { s = n * (n - 1) / 2 + n }
    { s = (n*n - n) / 2 + 2*n / 2 }
    { s == (n*n - n + 2*n) / 2 }
    { s == (n*n + n) / 2 }
    { s == (n + 1) * n / 2 }
    { s == (n + 1) * (n + 1 - 1) / 2 }
    n := n + 1;
    { s == n * (n - 1) / 2 }
  }
assert s == 33 * 32 / 2;
```

Full program

Need to choose initial values of s and n to establish invariant

```
var s := 0;
var n := 0;
while n != 33
    invariant s == n * (n - 1) / 2
{
    s := s + n;
    n := n + 1;
}
```

Loop termination

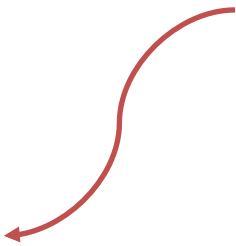
For a loop

```
while G
  invariant J
  decreases D
{
  Body
}
```

we need to prove

```
{ J && G }
ghost var d := D;
Body
{d > D }
```

Ghost variables are for reasoning only. They are not part of the compiled code.



Termination of quotient modulus

```
x, y := 0, 191;
while 7 <= y
  invariant 0 <= y && 7 * x + y == 191
  decreases y
{
  y := y - 7;
  x := x + 1;
}
```

```
{ 0 <= y && 7 * x + y == 191 && 7 <= y }
```

```
ghost var d := y;
```

```
y := y - 7;
```

```
x := x + 1;
```

```
{ d > y }
```

- $y < d$ follows from $y := y - 7$
- $0 \leq d$ follows from $0 \leq y$ in invariant

Quick body

```
x, y := 0, 191;
while 7 <= y
  invariant 0 <= y && 7 * x + y == 191
  decreases y
{
  y := 2;
  x := 27;
}
```

```
{ 0 <= y && 7 * x + y == 191 && 7 <= y }
```

```
ghost var d := y;
```

```
y := 2;
```

```
x := 27;
```

```
{ d > y }
```

- $y < d$ follows from $7 \leq y$ in invariant
- $0 \leq d$ follows from $0 \leq y$ in invariant

Default decreases in Dafny

If the loop guard is an arithmetic comparison of the form $E < F$
or $E \leq F$ then

`decreases` $F - E$

If the loop guard is an arithmetic comparison of the form $E \neq F$
then

`decreases` `if` $E < F$ `then` $F - E$ `else` $E - F$

Complete loop rule

```
{ J }
```

```
while G
```

```
    invariant J
```

```
    decreases D
```

```
{
```

```
    Body
```

```
}
```

```
{ J && !G }
```

```
{ J && G }
```

```
ghost var d := D;
```

```
Body
```

```
{ J && d > D }
```

Integer square root

```
method SquareRoot(N: nat) returns (r: nat)  
  ensures r*r <= N < (r+1)*(r+1)
```


Integer square root

```
method SquareRoot(N: nat) returns (r: nat)
  ensures r*r <= N && N < (r+1)*(r+1)
```

Loop design technique 5.1

For a postcondition $A \ \&\& \ B$, use
 A as the invariant and $\neg B$ as the guard

```
{
```

```
  while (r+1)*(r+1) <= N
    invariant r*r <= N
```

```
}
```

Integer square root

```
method SquareRoot(N: nat) returns (r: nat)
  ensures r*r <= N && N < (r+1)*(r+1)
```

Loop design technique 5.1

For a postcondition $A \ \&\& \ B$, use
 A as the invariant and $\neg B$ as the guard

```
{
  r := 0;
  while (r+1)*(r+1) <= N
    invariant r*r <= N
    { r := r + 1; }
}
```

A more efficient algorithm

Rather than calculate $(r + 1) * (r + 1)$ on each iteration, add a **new variable** s and maintain **invariant**

$$s == (r + 1) * (r + 1)$$

A more efficient algorithm

Rather than calculate $(r + 1) * (r + 1)$ on each iteration add a new variable $s == (r + 1) * (r + 1)$

Then we have s initially 1 , loop guard $s \leq N$ and invariant $s == (r + 1) * (r + 1)$

$$\{ s == (r + 1) * (r + 1) \}$$
$$\{ s == (r + 1 + 1) * (r + 1 + 1) \}$$
$$r := r + 1$$
$$\{ s == (r + 1) * (r + 1) \}$$

A more efficient algorithm

Rather than calculate $(r + 1) * (r + 1)$ on each iteration add new variable $s == (r + 1) * (r + 1)$

Then we have s initially 1 , loop guard $s \leq N$ and invariant $s == (r + 1) * (r + 1)$

$$\{ s == (r + 1) * (r + 1) \}$$
$$\{ s == r * r + 4 * r + 4 \}$$
$$\{ s == (r + 1 + 1) * (r + 1 + 1) \}$$
$$r := r + 1$$
$$\{ s == (r + 1) * (r + 1) \}$$

A more efficient algorithm

Rather than calculate $(r + 1) * (r + 1)$ on each iteration add new variable $s == (r + 1) * (r + 1)$

Then we have s initially 1 , loop guard $s \leq N$ and invariant $s == (r + 1) * (r + 1)$

$\{ s == (r + 1) * (r + 1) \}$

$\{ s == r * r + 2 * r + 1 + 2 * r + 3 \}$

$\{ s == r * r + 4 * r + 4 \}$

$\{ s == (r + 1 + 1) * (r + 1 + 1) \}$

$r := r + 1$

$\{ s == (r + 1) * (r + 1) \}$

A more efficient algorithm

Rather than calculate $(r + 1) * (r + 1)$ on each iteration add new variable $s == (r + 1) * (r + 1)$

Then we have s initially 1 , loop guard $s \leq N$ and invariant $s == (r + 1) * (r + 1)$

$\{ s == (r + 1) * (r + 1) \}$

$\{ s == (r + 1) * (r + 1) + 2 * r + 3 \}$

$\{ s == r * r + 2 * r + 1 + 2 * r + 3 \}$

$\{ s == r * r + 4 * r + 4 \}$

$\{ s == (r + 1 + 1) * (r + 1 + 1) \}$

$r := r + 1$

$\{ s == (r + 1) * (r + 1) \}$

A more efficient algorithm

Rather than calculate $(r + 1) * (r + 1)$ on each iteration add new variable $s == (r + 1) * (r + 1)$

Then we have s initially 1 , loop guard $s \leq N$ and invariant $s == (r + 1) * (r + 1)$

$\{ s == (r + 1) * (r + 1) \}$

$s := s + 2 * r + 3;$

$\{ s == (r + 1) * (r + 1) + 2 * r + 3 \}$

$\{ s == r * r + 2 * r + 1 + 2 * r + 3 \}$

$\{ s == r * r + 4 * r + 4 \}$

$\{ s == (r + 1 + 1) * (r + 1 + 1) \}$

$r := r + 1$

$\{ s == (r + 1) * (r + 1) \}$

A more efficient algorithm

Rather than calculate $(r + 1) * (r + 1)$ on each iteration add new variable $s == (r + 1) * (r + 1)$

Then we have s initially 1 , loop guard $s \leq N$ and invariant $s == (r + 1) * (r + 1)$

```
{ s == (r + 1) * (r + 1) }
{ s + 2*r + 3 == (r + 1) * (r + 1) + 2*r + 3 }
s := s + 2*r + 3;
{ s == (r + 1) * (r + 1) + 2*r + 3 }
{ s == r*r + 2*r + 1 + 2*r + 3 }
{ s == r*r + 4*r + 4 }
{ s == (r + 1 + 1) * (r + 1 + 1) }
r := r + 1
{ s == (r + 1) * (r + 1) }
```

Full program

```
method SquareRoot(N: nat) returns (r: nat)
  ensures r*r <= N < (r+1)*(r+1)
{
  r := 0;
  var s := 1;
  while s <= N
    invariant r*r <= N
    invariant s == (r+1)*(r+1)
    {
      s := s + 2*r + 3;
      r := r + 1;
    }
}
```

Exercises

1. For each of the following uses of loop specifications, indicate whether or not the loop's initial proof obligation is met and whether or not the assertion following the loop can be proved to hold.

a) `x := 0;`
`while x != 100`
 invariant `true`
assert `x == 100;`

b) `x := 20;`
`while 10 < x`
 invariant `x % 2 == 0`
assert `x == 10;`

c) `x := 20;`
`while x < 20`
 invariant `x % 2 == 0`
assert `x == 20;`

d) `x := 3;`
`while x < 2`
 invariant `x % 2 == 0`
assert `x % 2 == 0;`

e) `if 50 < x < 100 {`
 `while x < 85`
 invariant `x % 2 == 0`
 assert `x < 85 && x % 2 == 1;`
}

f) `if 0 <= x {`
 `while x % 2 == 0`
 invariant `x < 100`
 assert `0 <= x;`
}

g) `x := 0;`
`while x < 100`
 invariant `0 <= x < 100`
assert `x == 25;`

Exercises

2. For each program below, give a possible value of `i` of type `int` after the loop that shows that the assertion is not provable.

a) `i := 0;`
`while i < 100`
 `invariant 0 <= i`
`assert i == 100;`

b) `i := 100;`
`while 0 < i`
 `invariant true`
`assert i == 0;`

c) `i := 0;`
`while i < 97`
 `invariant 0 <= i <= 99`
`assert i == 99;`

d) `i := 22;`
`while i % 5 != 0`
 `invariant 10 <= i <= 100`
`assert i == 55;`

3. For each program in Exercise 2, strengthen the invariant so that the invariant both holds on entry to the loop and suffices to prove the assertion.