CS:5810 Formal Methods in Software Engineering

Reasoning about Loops

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 <= x <= 20 && x % 2 != 0 ==>
  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
assert s == 33 * 32 / 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
assert s == 33 * 32 / 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

To prove the partial correctness of a loop

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

G and J can be used to find an implementation of Body

we need to prove the validity of

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

Example: 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 <= y does not follow from the top line

So this loop body would be incorrect

Loop termination

To prove the total correctness of a loop

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

we also 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 && d >= 0}
```

- `d > y` follows from `y := y – 7`
- `d >= 0` follows from `0 <= 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 && d >= 0 }
```

- d > y follows from 7 <= y in invariant
- d >= 0 follows from 0 <= y in invariant

Default decreases in Dafny

If the loop guard is an arithmetic comparison of the form `E < F` or `E <= F` then

```
decreases F – E
```

If the loop guard is an arithmetic comparison of the form `E != 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 }
```

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;
}
```

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 11.0
For a postcondition A && B, use
A as the invariant and !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 11.0

For a postcondition A && B, use
A as the invariant and !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 <= 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 a new variable `s == (r + 1)*(r + 1)`

Then we have `s` initially `1`, loop guard `s <= 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 a new variable `s == (r + 1)*(r + 1)`

Then we have `s` initially `1`, loop guard `s <= 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 a new variable `s == (r + 1)*(r + 1)`

Then we have `s` initially `1`, loop guard `s <= 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 a new variable `s == (r + 1)*(r + 1)`

Then we have `s` initially `1`, loop guard `s <= 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 a new variable `s == (r + 1)*(r + 1)`

Then we have `s` initially 1, loop guard `s <= 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;
   }
}
```