CS:5810 Formal Methods in Software Engineering

Lemmas and Proofs in Dafny

Declaring a lemma

```
function More(x: int): int {
  if x <= 0 then 1 else More(x - 2) + 3
}
```

Is this function's output always greater than its input?

To prove this, we can add the *lemma*:

```
lemma MoreIsIncreasing(x: int)
  ensures x < More(x)
{
}
```

What is a lemma

Lemmas are distinguished methods with no output

They can have input parameters and a contract

- The contract describes (extrinsic) properties of some other method or function
- The body is a mix of program statements and specs that help Dafny prove the contract

Lemma application example

```
function More(n: int): int {
  if x <= 0 then 1 else More(x - 2) + 3 }

lemma MoreIsIncreasing(x: int)
  ensures x < More(x)

method ExampleLemmaUse(a: int)
  ensures a + 2 <= More(More(a))
{                                  // a is any integer
  var b := More(a);                // b == More(a)
  var c := More(b);                // c == More(More(a))
}
```

What we know

Dafny may not be able to conclude method's postcondition

Lemma application example

```
function More(n: int): int {
  if x <= 0 then 1 else More(x - 2) + 3 }

lemma MoreIsIncreasing(x: int)
  ensures x < More(x)

method ExampleLemmaUse(a: int)
  ensures a + 2 <= More(More(a))
{                                  // a is any integer
  var b := More(a);                // b == More(a)
  MoreIsIncreasing(a);             // a < More(a)
  MoreIsIncreasing(b);             // More(a) < More(More(a))
  var c := More(b);                // c == More(More(a))
  assert c - a >= 2;               // Dafny can verify this!
}
```

What we know

Proving a lemma

To prove a lemma, we may need to provide a body

```
lemma MoreIsIncreasing(x: int)
  ensures x < More(x)
{ } // Dafny can verify postcondition automatically, by induction
```

For illustrative purposes, let's turn induction off

```
lemma {:induction false} MoreIsIncreasing(x: int)
  ensures x < More(x)
{ } // Dafny cannot verify this automatically
```

Proving a lemma

Base structure of proof on structure of method

```
function method More(x: int): int {
   if x <= 0 then 1 else More(x - 2) + 3 }
```

```
lemma {:induction false} MoreIsIncreasing(x: int)
   ensures x < More(x)
{
   if x <= 0 { }              // this case verifies
   else { }                   // this case does not verify yet
}
```

Proof sketch

Proving a lemma

Base structure of proof on structure of method

```
function method More(x: int): int {
   if x <= 0 then 1 else More(x - 2) + 3 }
```

For **else** branch want to prove: `x < More(x - 2) + 3`

or, equivalently: `x - 3 < More(x - 2)`

```
lemma {:induction false} MoreIsIncreasing(x: int)
   ensures x < More(x)
{
   if x <= 0 { }
   else { }
}
```

Proving a lemma

Base structure of proof on structure of method

```
function method More(x: int): int {
   if x <= 0 then 1 else More(x - 2) + 3 }
```

For **else** branch want to prove: `x < More(x - 2) + 3`

or, equivalently: `x - 3 < More(x - 2)`

```
lemma {:induction false} MoreIsIncreasing(x: int)
   ensures x < More(x)
{
   if x <= 0 { }
   else { MoreIsIncreasing(x - 2); }   // x - 2 < More(x - 2)
}
```

Lemma fully verifies!

Checking termination

Note: Dafny provides default termination metric: `decreases x`

Termination check is necessary to avoid bogus proofs like:

```
lemma {:induction false} MoreIsIncreasing(x: int)
   ensures x < More(x)
{
   MoreIsIncreasing(x);
}
```

Floyd-Hoare logic proof

```
lemma {:induction false} MoreIsIncreasing(x: int)
   ensures x < More(x)
{  { true }
   if (x <= 0) {
      { x <= 0 }
      ?
      { x < More(x) }
   } else {
      {0 < x }
      ?
      { x < More(x) }
   }
   { x < More(x) }
}
```

Floyd-Hoare logic proof

then branch of `MoreIsIncreasing`:

```
if (x <= 0) {
   { x <= 0 }
   ?
   { x < More(x) }
}
```

```
function More(x: int): int {
  if x <= 0 then 1
  else More(x - 2) + 3
}
```

Floyd-Hoare logic proof

then branch of `MoreIsIncreasing`:

```
function More(x: int): int {
  if x <= 0 then 1
  else More(x - 2) + 3
}
```

```
if (x <= 0) {
    { x <= 0 }
    ?
    { x < More(x) &&
      More(x) == 1 }
}
```

(by definition of More)

Formula above ? implies that below ?,
so proof for **then** branch is done

Floyd-Hoare logic proof

else branch of `MoreIsIncreasing`:

```
{
    { 0 < x }
    { 0 < x &&
      More(x) == More(x - 2) + 3 }
     MoreIsIncreasing(x - 2); // induction hypothesis
    { 0 < x &&
      More(x) == More(x - 2) + 3 && x - 2 < More(x - 2) }
    { More(x) == More(x - 2) + 3 && x + 1 < More(x - 2) + 3 }
    { x + 1 < More(x) }
    { x < More(x) }
}
```

```
function More(x: int): int {
  if x <= 0 then 1
  else More(x - 2) + 3
}
```

Line before ? implies the line after, so proof is done

Proof assertions

then branch:

```
if (x <= 0) {
   assert x <= 0;
   assert More(x) == 1;        // by definition of More
   assert x < More(x);
}
```

know from if guard

postcondition of lemma

```
if (x <= 0) {
   assert More(x) == 1;       // by definition of More
}
```

Proof assertions

else branch:

```
{
   assert 0 < x;                          // from if guard
   assert More(x) == More(x - 2) + 3;     // by definition of More
   MoreIsIncreasing(x - 2);               // induction hypothesis
   assert More(x) == More(x - 2) + 3;     // preserved by lemma call
   assert x - 2 < More(x - 2);            // by induction hypothesis
   assert x + 1 < More(x - 2) + 3;        // by arithmetic
   assert x + 1 < More(x);                // by second assert
   assert x < More(x);                    // by arithmetic
}
```

```
function More(x: int): int {
  if x <= 0 then 1
  else More(x - 2) + 3
}
```

Proof by calculations

```
calc {
      5 * (x + 3);
   == 5 * x + 5 * 3; // distribute multiplication over addition
   == 5 * x + 15;    // compute 5*3
}
```

```
calc {
      (x + y) * (x - y);
   == x*x - x *y + y*x - y*y;  // distribute * over + and -
   == x*x - x*y + x*y - y*y;   // * is commutative: y*x == x*y
   == x*x - y*y                // the terms -x*y and x*y cancel
}
```

Proof calculations

If n has type nat

```
calc {
      3*x + n + n;
  == 3*x + 2*n;    // n + n = 2*n
  <= 3*n + 3*n;    // since 0 <= n
  == 3*(x + n);    // distribute * and +
}
```

That is, `3*x + n + n <= 3*(x + n)`

Checked proof hints

```
lemma {:induction false} MoreIsIncreasing(x: int)
  ensures x < More(x)
{
  if x <= 0 { }
  else {
    calc {
          More(x);
      == More(x - 2) + 3;                                  // def. of More
       > { MoreIsIncreasing(x - 2); } x - 2 + 3;           // induction
       > x;                                                // arithmetic
    }
  }
}
```

Alternative using Boolean expressions

```
lemma {:induction false} MoreIsIncreasing(x: int)
  ensures x < More(x)
{
  if x <= 0 { }
  else {
    calc {
           true
       ==> { MoreIsIncreasing(x - 2); } x - 2 < More(x - 2);
      <==> x + 1 < More(x - 2) + 3;     // add 3 to each side
      <==> x + 1 < More(x);             // def. of More
       ==> x < More(x);                 // arithmetic
    }
  }
}
```

Shorter calculations

Default operator

```
calc ==
{
      3*x + n + n;
      3*x + 2*x;
   <= 3*x + 3*n;
      3*(x + n);
}
```

```
calc {
   5 * (x + 3);
   5 * x + 5 * 3;
   5 * x + 15;
}
```

When no operator is given,
== is the default default operator

Example: Reduce

```
function Reduce(m: nat, x: int): int {
   if m == 0 then x else Reduce(m / 2, x + 1) - m
}
```

Does `Reduce(m,x)` ever return anything larger than `x`?

```
lemma {:induction false} ReduceUpperBound(m: nat, x: int)
   ensures Reduce(m, x) <= x
{
   if m == 0 { // trivial }
   else { // ... }
}
```

Example: Reduce

```
lemma {:induction false} ReduceUpperBound(m: nat, x: int)
  ensures Reduce(m, x) <= x
{
  if m == 0 { // trivial } else {
    calc {
        Reduce(m, x);
      == Reduce(m / 2, x + 1) - m;
      <= { ReduceUpperBound(m / 2, x + 1);
           assert Reduce(m / 2, x + 1) <= x + 1; }
        x + 1 - m;
      <= {assert 0 < m } x;
    }
  }
}
```

```
function Reduce(m: nat, x: int): int {
  if m == 0 then x
  else Reduce(m / 2, x + 1) - m
}
```

A lower bound

```
lemma {:induction false} ReduceLowerBound(m: nat, x: int)
  ensures x - 2*m <= Reduce(m, x)
{
  if m == 0 {
  } else {
     calc {
        Reduce(m, x);
     == Reduce(m/2, x+1) - m;   // def. of Reduce



     }
  }
}
```

A lower bound

```
lemma {:induction false} ReduceLowerBound(m: nat, x: int)
  ensures x - 2*m <= Reduce(m, x)
{
  if m == 0 {
  } else {
     calc {
        Reduce(m, x);
     == Reduce(m/2, x+1) - m;   // def. of Reduce
     >= { ReduceLowerBound(m/2, x+1);
             assert x+1 - 2*(m/2) <= Reduce(m/2, x+1); }
                    x+1 - 2*(m/2) - m;
     }
  }
}
```

A lower bound

```
lemma {:induction false} ReduceLowerBound(m: nat, x: int)
  ensures x - 2*m <= Reduce(m, x)
{
  if m == 0 {
  } else {
     calc {
        Reduce(m, x);
     == Reduce(m/2, x+1) - m;   // def. of Reduce
     >= { ReduceLowerBound(m/2, x+1);
             assert x+1 - 2*(m/2) <= Reduce(m/2, x+1); }
                    x+1 - 2*(m/2) - m;
        == x+1 - m - m;  }
  }
}
```

A lower bound

```
lemma {:induction false} ReduceLowerBound(m: nat, x: int)
  ensures x - 2*m <= Reduce(m, x)
{
  if m == 0 {
  } else {
     calc {
        Reduce(m, x);
     == Reduce(m/2, x+1) - m;   // def. of Reduce
     >= { ReduceLowerBound(m/2, x+1);
             assert x+1 - 2*(m/2) <= Reduce(m/2, x+1); }
                    x+1 - 2*(m/2) - m;
        == x+1 - m - m;  } // error: this might not hold
  }
}
```

A lower bound

```
lemma {:induction false} ReduceLowerBound(m: nat, x: int)
  ensures x - 2*m <= Reduce(m, x)
{
  if m == 0 {
  } else {
     calc {
        Reduce(m, x);
     == Reduce(m/2, x+1) - m;    // def. of Reduce
     >= { ReduceLowerBound(m/2, x+1);
             assert x+1 - 2*(m/2) <= Reduce(m/2, x+1); }
                    x+1 - 2*(m/2) - m;
        == { assert 2*(m/2) == m; }
              x+1 - m - m;  // wrong!
  }
}
```

A lower bound

```
lemma {:induction false} ReduceLowerBound(m: nat, x: int)
  ensures x - 2*m <= Reduce(m, x)
{
  if m == 0 {
  } else {
     calc {
        Reduce(m, x);
     == Reduce(m/2, x+1) - m;   // def. of Reduce
     >= { ReduceLowerBound(m/2, x+1);
             assert x+1 - 2*(m/2) <= Reduce(m/2, x+1); }
                    x+1 - 2*(m/2) - m;
        >= x+1 - m - m;  }
  }
}
```

A lower bound

```
lemma {:induction false} ReduceLowerBound(m: nat, x: int)
  ensures x - 2*m <= Reduce(m, x)
{
  if m == 0 {
  } else {
     calc {
        Reduce(m, x);
     == Reduce(m/2, x+1) - m;   // def. of Reduce
     >= { ReduceLowerBound(m/2, x+1);
             assert x+1 - 2*(m/2) <= Reduce(m/2, x+1); }
                    x+1 - 2*(m/2) - m;
        > x - 2*m;  }
  }
}
```

Commutativity of multiplication

```
function Mult(x: nat, y: nat): nat {
    if y == 0 then 0 else x + Mult(x, y-1)
}

lemma {:induction false} MultCommutative(x: nat, y: nat)
ensures Mult(x, y) == Mult(y, x)
{

}
```

Commutativity of multiplication

```
function Mult(x: nat, y: nat): nat {
    if y == 0 then 0 else x + Mult(x, y-1)
}

lemma {:induction false} MultCommutative(x: nat, y: nat)
ensures Mult(x, y) == Mult(y, x)
{
    if x == y { }









}
```

Commutativity of multiplication

```
function Mult(x: nat, y: nat): nat {
    if y == 0 then 0 else x + Mult(x, y-1)
}

lemma {:induction false} MultCommutative(x: nat, y: nat)
ensures Mult(x, y) == Mult(y, x)
{
    if x == y { }
    else if x == 0 { MultCommutative(x, y-1); }




}
```

Since Mult(x, y) = Mult(x, y-1) when x == 0.

Commutativity of multiplication

```
function Mult(x: nat, y: nat): nat {
    if y == 0 then 0 else x + Mult(x, y-1)
}

lemma {:induction false} MultCommutative(x: nat, y: nat)
    ensures Mult(x, y) == Mult(y, x)
{
    if x == y { }
    else if x == 0 { MultCommutative(x, y-1); }
    else if y < x { MultCommutative(y, x); }  // y, x < x, y




}
```

Since default termination metric x, y decreases when y < x.

Commutativity of multiplication

```
function Mult(x: nat, y: nat): nat {
    if y == 0 then 0 else x + Mult(x, y-1)
}

lemma {:induction false} MultCommutative(x: nat, y: nat)
    ensures Mult(x, y) == Mult(y, x)
{
    if x == y { }
    else if x == 0 { MultCommutative(x, y-1); }
    else if y < x { MultCommutative(y, x); }  // y, x < x, y
    else {
        ...
    }
}
```

Commutativity of multiplication

```
calc {
     Mult(x,y);
== x + Mult(x, y-1);   // def. of Mult




}
```

Commutativity of multiplication

```
calc {
     Mult(x,y);
== x + Mult(x, y-1);   // def. of Mult
== { MultCommutative(x, y-1); }  x + Mult(y-1, x);




}
```

Commutativity of multiplication

```
calc {
     Mult(x,y);
== x + Mult(x, y-1);   // def. of Mult
== { MultCommutative(x, y-1); }  x + Mult(y-1, x);
== x + y -1 + Mult(y-1, x-1);   // def. of Mult




}
```

Commutativity of multiplication

```
calc {
     Mult(x,y);
== x + Mult(x, y-1);   // def. of Mult
== { MultCommutative(x, y-1); }  x + Mult(y-1, x);
== x + y -1 + Mult(y-1, x-1);   // def. of Mult
```

Can’t call MultCommutative(y-1, x-1)
since y-1, x-1 is not smaller than x, y .

```
}
```

Commutativity of multiplication

```
calc {
     Mult(x,y);
== x + Mult(x, y-1);   // def. of Mult
== { MultCommutative(x, y-1); }  x + Mult(y-1, x);
== x + y -1 + Mult(y-1, x-1);   // def. of Mult
== { MultCommutative(x-1, y-1); }   x + y -1 + Mult(x-1,y-1);




}
```

Commutativity of multiplication

```
calc {
     Mult(x,y);
== x + Mult(x, y-1);   // def. of Mult
== { MultCommutative(x, y-1); }  x + Mult(y-1, x);
== x + y -1 + Mult(y-1, x-1);   // def. of Mult
== { MultCommutative(x-1, y-1); }   x + y -1 + Mult(x-1,y-1);
== y + Mult(x-1, y);   // def. of Mult
== { MultCommutative(x-1, y); }  y + Mult(y, x-1);
== Mult(y, x)   // def. of Mult
}
```

Summary of proof

Cases:

- x == y
- 0 == x != y
- y < x
- 0 < x < y

Recursive calls:

- MultCommutative(x, y-1) in a context where 0 < y
- MultCommutative(y, x) in a context where y < x
- MultCommutative(x, y-1), again where 0 < y
- MultCommutative(x-1, y-1) in a context where 0 < x and 0 < y
- MultCommutative(x-1, y) in a context where 0 < x