CS:5810
Formal Methods in Software Engineering

Reasoning about Iterative Programs in Dafny

Iterative Fibonacci

```
function Fib(n: nat): nat {
  if n < 2 then n else Fib(n-2) + Fib(n-1)
}

method ComputeFib(n: nat) returns (x: nat)
  ensures x == Fib(n)
{
  x := 0;
  var i := 0;
  while i != n
    invariant 0 <= i <= n
    invariant x == Fib(i)
}
```

Iterative Fibonacci

Loop design technique 6.1 (*Replace a constant by a variable)*
For a loop to establish a condition P[C], where C is an expression that maintains a constant value throughout the loop, use a variable k that the loop changes until it equals C, and make P[k] a loop invariant

Example: to establish `x == Fib(n)` introduce `i` and

```
invariant x == Fib(i)
```

Iterative Fibonacci

```
method ComputeFib(n: nat) returns (x: nat)
  ensures x == Fib(n)
{
  x := 0;
  var i := 0;
  while i != n
    invariant 0 <= i <= n
    invariant x == Fib(i)
  {
    ...
    i := i + 1;
  }
}
```

Iterative Fibonacci

```
method ComputeFib(n: nat) returns (x: nat)
  ensures x == Fib(n)
{
  x := 0;
  var i := 0;
  while i != n
    invariant 0 <= i <= n
    invariant x == Fib(i) && y == Fib(i + 1)
  {
    ...
    i := i + 1;
  }
}
```

Cannot use `y == Fib(i-1)`
as not defined when `i == 0`

Iterative Fibonacci

```
method ComputeFib(n: nat) returns (x: nat)
  ensures x == Fib(n)
{
  x := 0;
  var i := 0;
  while i != n
    invariant 0 <= i <= n
    invariant x == Fib(i) && y == Fib(i + 1)
  {
    ...
    i := i + 1;
  }
}
```

Can use `(i == 0 || y == Fib(i-1))` but will lead to more complex code

Loop body

```
{ 0 <= i <= n  && x == Fib(i)
               && y == Fib(i+1) &&  i != n}
```

```
i := i + 1;
{ 0 <= i <= n && x == Fib(i) && y == Fib(i+1) }
```

Loop body

```
{ 0 <= i <= n  && x == Fib(i)
               && y == Fib(i+1) &&  i != n}
```

```
{ 0 <= i+1 <= n && x == Fib(i+1)
                && y == Fib(i+1+1) }
i := i + 1;
{ 0 <= i <= n && x == Fib(i) && y == Fib(i+1) }
```

Loop body

```
{ 0 <= i <= n  && x == Fib(i)
               && y == Fib(i+1) &&  i != n}









{ 0 <= i+1 <= n && x == Fib(i+1) && y == Fib(i+2)}
{ 0 <= i+1 <= n && x == Fib(i+1)
                && y == Fib(i+1+1) }
i := i + 1;
{ 0 <= i <= n && x == Fib(i) && y == Fib(i+1) }
```

Loop body

```
{ 0 <= i <= n  && x == Fib(i)
               && y == Fib(i+1) &&  i != n}




{ 0 <= i+1 <= n && x == Fib(i+1)
                && y == Fib(i) + Fib(i+1) }
{ 0 <= i+1 <= n && x == Fib(i+1) && y == Fib(i+2)}
{ 0 <= i+1 <= n && x == Fib(i+1)
                && y == Fib(i+1+1) }
i := i + 1;
{ 0 <= i <= n && x == Fib(i) && y == Fib(i+1) }
```

Loop body

```
{ 0 <= i <= n  && x == Fib(i)
               && y == Fib(i+1) &&  i != n}




x, y := y, x + y;
{ 0 <= i+1 <= n && x == Fib(i+1)
                && y == Fib(i) + Fib(i+1) }
{ 0 <= i+1 <= n && x == Fib(i+1) && y == Fib(i+2)}
{ 0 <= i+1 <= n && x == Fib(i+1)
                && y == Fib(i+1+1) }
i := i + 1;
{ 0 <= i <= n && x == Fib(i) && y == Fib(i+1) }
```

Loop body

```
{ 0 <= i <= n  && x == Fib(i)
               && y == Fib(i+1) &&  i != n}



{ 0 <= i+1 <= n && y == Fib(i+1)
                && x+y == Fib(i) + Fib(i+1) }
x, y := y, x + y;
{ 0 <= i+1 <= n && x == Fib(i+1)
                && y == Fib(i) + Fib(i+1) }
{ 0 <= i+1 <= n && x == Fib(i+1) && y == Fib(i+2)}
{ 0 <= i+1 <= n && x == Fib(i+1)
                && y == Fib(i+1+1) }
i := i + 1;
{ 0 <= i <= n && x == Fib(i) && y == Fib(i+1) }
```

Loop body

```
{ 0 <= i <= n  && x == Fib(i)
               && y == Fib(i+1) &&  i != n}

{ 0 <= i+1 <= n && x == Fib(i) && y == Fib(i+1) }
{ 0 <= i+1 <= n && y == Fib(i+1)
                && x+y == Fib(i) + Fib(i+1) }
x, y := y, x + y;
{ 0 <= i+1 <= n && x == Fib(i+1)
                && y == Fib(i) + Fib(i+1) }
{ 0 <= i+1 <= n && x == Fib(i+1) && y == Fib(i+2)}
{ 0 <= i+1 <= n && x == Fib(i+1)
                && y == Fib(i+1+1) }
i := i + 1;
{ 0 <= i <= n && x == Fib(i) && y == Fib(i+1) }
```

Loop body

```
{ 0 <= i <= n  && x == Fib(i)
               && y == Fib(i+1) &&  i != n}
{ 0 <= i <= n   && x == Fib(i) && y == Fib(i+1) }
{ 0 <= i+1 <= n && x == Fib(i) && y == Fib(i+1) }
{ 0 <= i+1 <= n && y == Fib(i+1)
                && x+y == Fib(i) + Fib(i+1) }
x, y := y, x + y;
{ 0 <= i+1 <= n && x == Fib(i+1)
                && y == Fib(i) + Fib(i+1) }
{ 0 <= i+1 <= n && x == Fib(i+1) && y == Fib(i+2)}
{ 0 <= i+1 <= n && x == Fib(i+1)
                && y == Fib(i+1+1) }
i := i + 1;
{ 0 <= i <= n && x == Fib(i) && y == Fib(i+1) }
```

Full program

```
method ComputeFib(n: nat) returns (x: nat)
  ensures x == Fib(n)
{
  x := 0;
  var i := 0;
  while i != n
    invariant 0 <= i <= n
    invariant x == Fib(i)
    invariant y == Fib(i + 1)
  {
    x, y := y, x + y;
    i := i + 1;
  }
}
```

Powers of 2

Define a function that computes 2^n using the facts

$2^0 == 1$ and, for any other exponent n,

$2^n == 2 * 2^{n-1}$

```
function Power(n: nat): nat {
  if n == 0 then 1 else 2 * Power(n-1)
}

method ComputePower(n: nat) returns (p: nat)
  ensures p == Power(n)
```

The usual invariant

```
{
  p := 1;
  var i := 0;
  while i != n
    invariant 0 <= i <= n
    invariant p == Power(i)
}
```

The usual invariant

```
{
  p := 1;
  var i := 0;
  while i != n
    invariant 0 <= i <= n
    invariant p == Power(i)
}

  { 0 <= i <= n && p == Power(i) && i != n }
  { 0 <= i + 1 <= n &&  2 * p == Power(i + 1) }
  p := 2 * p;
  { 0 <= i + 1 <= n && p == Power(i + 1)   }
  i := i + 1;
  { 0 <= i <= n && p == Power(i) }
```

An alternative invariant

The previous invariant on p focuses on *what has been computed so far*

We can also focus on *what is left to do*

```
p := 1;
var i := 0;
while i != n
  invariant 0 <= i <= n
  invariant p * Power(n-i) == Power(n)
```

The invariant holds initially, and after the loop

```
p * Power(0) == Power(n)
```

An alternative invariant

Loop design technique 6.2

If you're trying to solve a problem of the form `p == F(n)`, you may be able to do so with a loop index `i` satisfying `0 <= i <= n` and either the *what-has-been-done* invariant

```
invariant p == F(i)
```

or the *what's-yet-to-be-done* invariant

```
invariant p ★ F(n - i) == F(n)
```

where ★ is some kind of combination operation

Fibonacci squared

```
method SquareFib(N: nat) returns (x: nat)
  ensures x == Fib(N) * Fib(N)
```

Loop design technique 6.3
If a problem can be made simpler by having a precomputed quantity Q, then introduce a new variable q with the intention of establishing and maintaining the invariant `q == Q`

A simple start

```
method SquareFib(N: nat) returns (x: nat)
  ensures x == Fib(N) * Fib(N)
{
  x := 0;
  var n := 0;
  while n != N
    invariant 0 <= n <= N
    invariant x == Fib(n) * Fib(n)
}
```

```
{ x == Fib(n+1)*Fib(n+1) }
n := n + 1;
{ x == Fib(n)*Fib(n) }
```

Cannot expand Fib(n + 1) to Fib(n) and Fib(n - 1) since n-1 may be negative

A wish

Let’s *wish* that we had a variable

```
y == Fib(n+1) * Fib(n+1)
```

```
{ x == Fib(n)*Fib(n) && n != N }
{ true }
{ Fib(n+1)*Fib(n+1) == Fib(n+1)*Fib(n+1) }
x := y;          // where y == Fib(n+1)*Fib(n+1)
{ x == Fib(n+1)*Fib(n+1) }
n := n + 1;
{ x == Fib(n)*Fib(n) }
```

A wish

Add a new invariant:

```
invariant y == Fib(n+1) * Fib(n+1)
```

```
{ y == Fib(n+2)*Fib(n+2) }
n := n + 1;
{ y == Fib(n+1)*Fib(n+1) }
```

A wish

Add a new invariant:

```
invariant y == Fib(n+1) * Fib(n+1)
```

```
{ y == (Fib(n) + Fib(n+1))*(Fib(n) + Fib(n+1)) }
{ y == Fib(n+2)*Fib(n+2) }
n := n + 1;
{ y == Fib(n+1)*Fib(n+1) }
```

A wish

Add a new invariant:

```
        invariant y == Fib(n+1) * Fib(n+1)
```

```
{ y == Fib(n)*Fib(n) + 2*Fib(n)*Fib(n+1)
                     + Fib(n+1)*Fib(n+1) }
{ y == (Fib(n) + Fib(n+1))*(Fib(n) + Fib(n+1)) }
{ y == Fib(n+2)*Fib(n+2) }
n := n + 1;
{ y == Fib(n+1)*Fib(n+1) }
```

A wish

Add a new invariant:

```
invariant y == Fib(n+1) * Fib(n+1)
```

x == Fib(n) * Fib(n)

y == Fib(n+1) * Fib(n+1)

```
{ y == Fib(n)*Fib(n) + 2*Fib(n)*Fib(n+1)
                     + Fib(n+1)*Fib(n+1) }
{ y == (Fib(n) + Fib(n+1))*(Fib(n) + Fib(n+1)) }
{ y == Fib(n+2)*Fib(n+2) }
n := n + 1;
{ y == Fib(n+1)*Fib(n+1) }
```

A wish

Add a new invariant:

```
        invariant y == Fib(n+1) * Fib(n+1)
```

```
y := x + k + y; // where k == 2*Fib(n)*Fib(n+1)
{ y == Fib(n)*Fib(n) + 2*Fib(n)*Fib(n+1)
                     + Fib(n+1)*Fib(n+1) }
{ y == (Fib(n) + Fib(n+1))*(Fib(n) + Fib(n+1)) }
{ y == Fib(n+2)*Fib(n+2) }
n := n + 1;
{ y == Fib(n+1)*Fib(n+1) }
```

A wish

Add a new invariant:

```
        invariant y == Fib(n+1) * Fib(n+1)
```

```
{ x + k + y == x + k + Fib(n+1)*Fib(n+1) }
y := x + k + y; // where k == 2*Fib(n)*Fib(n+1)
{ y == Fib(n)*Fib(n) + 2*Fib(n)*Fib(n+1)
                     + Fib(n+1)*Fib(n+1) }
{ y == (Fib(n) + Fib(n+1))*(Fib(n) + Fib(n+1)) }
{ y == Fib(n+2)*Fib(n+2) }
n := n + 1;
{ y == Fib(n+1)*Fib(n+1) }
```

A wish

Add a new invariant:

```
        invariant y == Fib(n+1) * Fib(n+1)
```

```
{ y == Fib(n+1)*Fib(n+1) }
{ x + k + y == x + k + Fib(n+1)*Fib(n+1) }
y := x + k + y; // where k == 2*Fib(n)*Fib(n+1)
{ y == Fib(n)*Fib(n) + 2*Fib(n)*Fib(n+1)
                     + Fib(n+1)*Fib(n+1) }
{ y == (Fib(n) + Fib(n+1))*(Fib(n) + Fib(n+1)) }
{ y == Fib(n+2)*Fib(n+2) }
n := n + 1;
{ y == Fib(n+1)*Fib(n+1) }
```

Another wish

Add a new invariant:

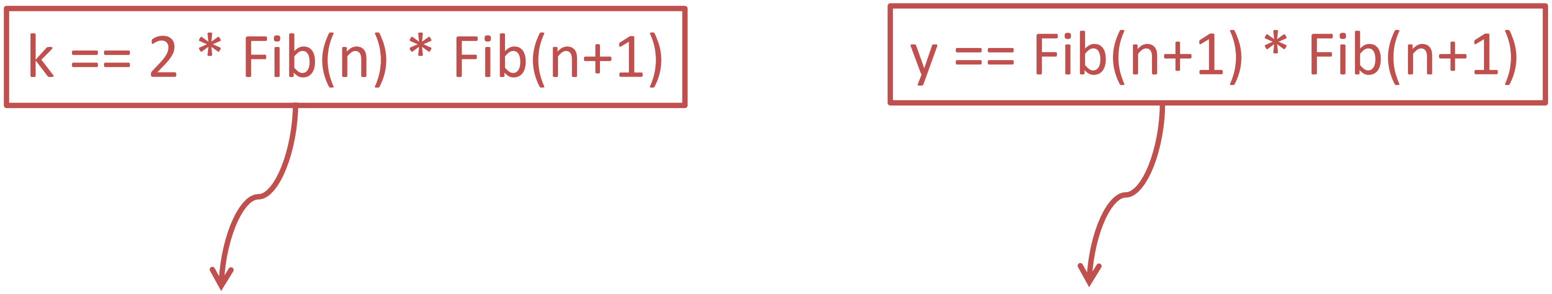
```
        invariant k == 2 * Fib(n) * Fib(n+1)
```

```
{ k == 2*Fib(n)*Fib(n+1) + 2*Fib(n+1)*Fib(n+1) }
{ k == 2*Fib(n+1)*(Fib(n) + Fib(n+1)); }
{ k == 2*Fib(n+1)*Fib(n+2) }
n := n + 1;
{ k == 2*Fib(n)*Fib(n+1) }
```

Another wish

Add a new invariant:

```
invariant k == 2 * Fib(n) * Fib(n+1)
```



k == 2 * Fib(n) * Fib(n+1)

y == Fib(n+1) * Fib(n+1)

```
{ k == 2*Fib(n)*Fib(n+1) + 2*Fib(n+1)*Fib(n+1) }
{ k == 2*Fib(n+1)*(Fib(n) + Fib(n+1)); }
{ k == 2*Fib(n+1)*Fib(n+2) }
n := n + 1;
{ k == 2*Fib(n)*Fib(n+1) }
```

Another wish

Add a new invariant:

```
invariant k == 2 * Fib(n) * Fib(n+1)
```

```
k := k + y + y;
{ k == 2*Fib(n)*Fib(n+1) + 2*Fib(n+1)*Fib(n+1) }
{ k == 2*Fib(n+1)*(Fib(n) + Fib(n+1)); }
{ k == 2*Fib(n+1)*Fib(n+2) }
n := n + 1;
{ k == 2*Fib(n)*Fib(n+1) }
```

Another wish

Add a new invariant:

```
invariant k == 2 * Fib(n) * Fib(n+1)
```

```
{ k + y + y == 2*Fib(n)*Fib(n+1) + 2*y }
k := k + y + y;
{ k == 2*Fib(n)*Fib(n+1) + 2*Fib(n+1)*Fib(n+1) }
{ k == 2*Fib(n+1)*(Fib(n) + Fib(n+1)); }
{ k == 2*Fib(n+1)*Fib(n+2) }
n := n + 1;
{ k == 2*Fib(n)*Fib(n+1) }
```

Another wish

Add a new invariant:

```
invariant k == 2 * Fib(n) * Fib(n+1)
```

```
{ k == 2*Fib(n)*Fib(n+1) }
{ k + y + y == 2*Fib(n)*Fib(n+1) + 2*y }
k := k + y + y;
{ k == 2*Fib(n)*Fib(n+1) + 2*Fib(n+1)*Fib(n+1) }
{ k == 2*Fib(n+1)*(Fib(n) + Fib(n+1)); }
{ k == 2*Fib(n+1)*Fib(n+2) }
n := n + 1;
{ k == 2*Fib(n)*Fib(n+1) }
```

Putting it all together

```
method SquareFib(N: nat) returns (x: nat)
  ensures x == Fib(N) * Fib(N)
{
  x := 0;
  var n := 0;
  var y := 1; // as Fib(0+1)*Fib(0+1) == 1*1 == 1
  var k := 0; // as 2 * 0 * 1 == 0
  while n != N
    invariant 0 <= n <= N
    invariant x == Fib(n) * Fib(n)
    invariant y == Fib(n+1) * Fib(n+1)
    invariant k == 2 * Fib(n) * Fib(n+1)
}
```

Putting it all together

The loop body is

```
{
  x, y, k := y, x + k + y, k + y + y;
  n := n + 1;
}
```

We could replace the simultaneous assignment with

```
{
  var prev_x := x;  var prev_y := y;
  x := prev_y;
  y := prev_x + k + prev_y;
  k := k + prev_y + prev_y;
}
```

Exercises

1. Below is the `ComputePower` method from the lecture without the loop body.

```
function Power(n: nat): nat { if n == 0 then 1 else 2 * Power(n-1) }

method ComputePower(n: nat) returns (p: nat)
  ensures p == Power(n)
{
  p := 1;
  var i := 1;
  while i < n
    invariant 0 <= i <= n
    invariant p == Power(i)
}
```

How does the Dafny verifier respond if you

a) change `p := 1` to `p := 2`?

b) change `p := 1` to `p := 2` and change the invariant to `p == Power( i + 1)`?

c) change `p := 1` to `p := 2` and change `i := 0` to `i := 1`?

Exercises

2. Implement the following method

```
method Cube(n: nat) returns (c: nat)
  ensures c == n * n * n
```

with a loop that iterates `n` times and only does addition (no multiplication).