CS 477

2/6/2019

Verification using invariants in Dafny

Quantifiers

```
method m(n: nat)
{
  var i: int := 0;
  while i < n
    invariant
  {
    i := i + 1;
  }
  assert i == n;
}
```

$i \leq n$

:= vs ==

method

assert

$\forall i \in \mathbb{N}, n \in \mathbb{N}$ $i = 0 \Rightarrow i \leq n$

```
method m(n: nat)
{
  var i: int := 0;
  while i != n
    invariant
  {
    i := i + 1;
  }
  assert i == n;
}
```

$i \leq n$

$\forall i \in \mathbb{N}, n \in \mathbb{N}$

$i \leq n \wedge i \neq n$

$\wedge\ i' = i + 1$

$\Rightarrow i' \leq n$

Verification Conditions.

$\forall i \in \mathbb{N}, n \in \mathbb{N}$

$i \leq n \wedge \neg(i \neq n) \Rightarrow i = n$

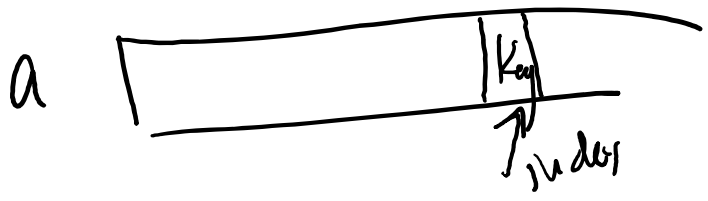
```
method m(n: nat)
{
  var i: int := 2;
  while i != n
    invariant i % 2 == 0
  {
    i := i *2;
    n := n-1;
  }
  assert i % 2 == 0;
}
```

R: {i = 2^k | for some k}

i is even

```
function fib(n: nat): nat
{
  if n == 0 then 0 else
  if n == 1 then 1 else
               fib(n - 1) + fib(n - 2)
}
method ComputeFib(n: nat) returns (b: nat)
  ensures b == fib(n)
{
  var i := 1;
  var a := 0;
     b := 1;
  while i < n
    invariant 0 < i <= n
    invariant a == fib(i - 1)
    invariant b == fib(i)
  {
    a, b := b, a + b;
    i := i + 1;
  }
}
```

```
function fib(n: nat): nat
{
  if n == 0 then 0 else
  if n == 1 then 1 else
               fib(n - 1) + fib(n - 2)
}
method ComputeFib(n: nat) returns (b: nat)
  ensures b == fib(n)
{
   if n == 0 { return 0; }
  var i := 1;
  var a := 0;
     b := 1;
  while i < n
    invariant 0 < i <= n
    invariant a == fib(i - 1)
    invariant b == fib(i)
  {
    a, b := b, a + b;
    i := i + 1;
  }
}
```



Arraya and Quantifiers

```
method Find(a: array<int>, key: int) returns (index: int)
  ensures 0 <= index ==> index < a.Length && a[index] == key
{
    index := -1;
}
```

Linear Search

index < 0

$\Rightarrow \forall k \; 0 \le k < |a| \Rightarrow a[k] \neq key$

```
method Find(a: array<int>, key: int) returns (index: int)
  ensures 0 <= index ==> index < a.Length && a[index] == key
  ensures index < 0 ==> forall k :: 0 <= k < a.Length ==> a[k] != key
{
  index := 0;
  while index < a.Length
  invariant
  {
    if a[index] == key { return; }
    index := index + 1;
  }
  index := -1;
}
```

$\forall j < index \quad a[j] \neq key$

$j \geq 0$

Linear Search

```
method Find(a: array<int>, key: int) returns (index: int)
  ensures 0 <= index ==> index < a.Length &amp;amp;&amp;amp; a[index] == key
  ensures index < 0 ==> forall k :: 0 <= k < a.Length ==> a[k] != key
{
  index := 0;
  while index < a.Length
    invariant 0 <= index <= a.Length
    invariant forall k :: 0 <= k < index ==> a[k] != key
  {
    if a[index] == key { return; }
    index := index + 1;
  }
  index := -1;
}
```

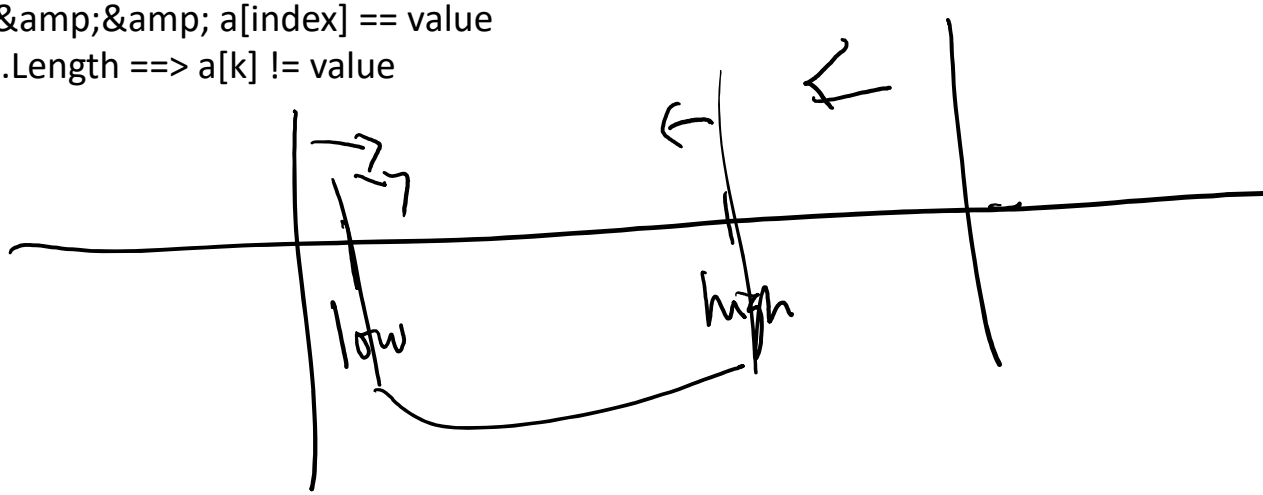
When I get out,
index ≥ a.Length
So index = a.Length.

Binary Search

```
predicate sorted(a: array<int>)
  requires a != null
  reads a
{
  forall j, k :: 0 <= j < k < a.Length ==> a[j] <= a[k]
}

method BinarySearch(a: array<int>, value: int) returns (index: int)
  requires a != null && 0 <= a.Length && sorted(a)
  ensures 0 <= index ==> index < a.Length && a[index] == value
  ensures index < 0 ==> forall k :: 0 <= k < a.Length ==> a[k] != value
{
  // Code here
}
```

```
method BinarySearch(a: array<int>, value: int) returns (index: int)
  requires a != null &amp;&amp; 0 <= a.Length &amp;&amp; sorted(a)
  ensures 0 <= index ==> index < a.Length &amp;&amp; a[index] == value
  ensures index < 0 ==> forall k :: 0 <= k < a.Length ==> a[k] != value
{
  var low, high := 0, a.Length;
  while low < high
    invariant …
  {
    var mid := (low + high) / 2;
    if a[mid] < value
    {       low := mid + 1;  }
    else if value < a[mid]
    {   high := mid;  }
    else
    {  return mid; }
  }
  return -1;
}
```



```
method BinarySearch(a: array<int>, value: int) returns (index: int)
  requires a != null &amp;amp;&amp;amp; 0 <= a.Length &amp;amp;&amp;amp; sorted(a)
  ensures 0 <= index ==> index < a.Length &amp;amp;&amp;amp; a[index] == value
  ensures index < 0 ==> forall k :: 0 <= k < a.Length ==> a[k] != value
{
  var low, high := 0, a.Length;
  while low < high
    invariant …
  {
    var mid := (low + high) / 2;
    if a[mid] < value
    {       low := mid + 1;  }
    else if value < a[mid]
    {   high := mid;  }
    else
    {  return mid; }
  }
  return -1;
}
```

```
invariant 0 <= low <= high <= a.Length
invariant forall i ::
      0 <= i < a.Length &amp;amp;&amp;amp; !(low <= i < high) ==> a[i] != value
```