CS 456

Programming Languages

Fall 2024

Week 14

Dafny (cont)

Basic setup

- Specify correctness conditions as pre/post-conditions that can be checked (mostly) automatically using a WP inference procedure
- But, not all properties we wish to verify can be expressed in terms of actions on the transition relation defined by axiomatic rules

Need proof techniques that allow us to verify properties over:

1. Inductive datatypes (e.g., lists, trees, …)
2. Semantic objects (e.g., heaps)
3. Imperative data structures (e.g, arrays)

Additionally, Dafny verifies total correctness

- Hoare rules only assert partial correctness properties
- Need additional insight to reason about termination

Lemmas

Sometimes, the property we wish to prove cannot be automatically verified. To help Dafny, we can provide *lemmas*, theorems that exist in service of proving some other property.

```
method FindZero(a: array<int>) returns (index: int)
  requires forall i :: 0 <= i < a.Length ==> 0 <= a[i]
  requires forall i :: 0 < i < a.Length ==> a[i-1]-1 <= a[i]
{
}
```

Precondition restricts input array such that all elements are greater than or equal to zero and each successive element in the array can decrease by at most one from the previous element.

We can take advantage of this observation in searching for the first zero in the array, by skipping elements. E.g., if a[j] = 7, then index of next possible zero cannot be before a[j + a[j]], i.e., if j = 3, then first possible zero can only be at a[10]

Lemmas


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

Dafny complains about the second loop invariant!

Precondition makes a claim about successive elements,
but invariant relies on generalizing this claim to sequences

Lemmas

Introduce a lemma (a ghost method) that claims all elements from `a[j]` to `a[j + a[j]]` are non-zero

```
lemma SkippingLemma(a: array<int>, j: int)
  requires forall i :: 0 <= i < a.Length ==> 0 <= a[i]
  requires forall i :: 0 < i < a.Length ==> a[i-1]-1 <= a[i]
  requires 0 <= j < a.Length
  ensures forall i :: j <= i < j + a[j] && i < a.Length ==> a[i] != 0
{
  //...
}
```

To use this lemma, "invoke it" in the body of `FindZero`!

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

Lemmas

While `FindZero` now verifies, we still need to provide a proof for `SkippingLemma`.
First cut: uses asserts to ascertain what Dafny can understand:

```
lemma SkippingLemma(a: array<int>, j: int)
  requires forall i :: 0 <= i < a.Length ==> 0 <= a[i]
  requires forall i :: 0 < i < a.Length ==> a[i-1]-1 <= a[i]
  requires 0 <= j < a.Length - 3  // strengthened the post-condition
{
  assert a[j  ] - 1 <= a[j+1];
  assert a[j+1] - 1 <= a[j+2];
  assert a[j+2] - 1 <= a[j+3];
  // therefore:
  assert a[j  ] - 3 <= a[j+3];
}
```

Dafny is able to verify this change of reasoning

Lemmas


```
lemma SkippingLemma(a: array<int>, j: int)
  requires forall i :: 0 <= i < a.Length ==> 0 <= a[i]
  requires forall i :: 0 < i < a.Length ==> a[i-1]-1 <= a[i]
  requires 0 <= j < a.Length
  ensures forall k :: j <= k < j + a[j] && k < a.Length ==> a[k] != 0
{
  var i := j;
  while i < j + a[j] && i < a.Length
    invariant i < a.Length ==> a[j] - (i-j) <= a[i]
    invariant forall k :: j <= k < i && k < a.Length ==> a[k] != 0
  {
    i := i + 1;
  }
}
```

The body of the lemma constitutes a proof that its postcondition holds

Lemmas and Induction

Sometimes the property we are interested in proving has a natural inductive interpretation

```
function count(a: seq<bool>): nat
{
  if |a| == 0 then 0 else
  (if a[0] then 1 else 0) + count(a[1..])
}
method m()
{
  assert count([]) == 0;
  assert count([true]) == 1;
  assert count([false]) == 0;
  assert count([true, true]) == 2;
}
```

Suppose we wish to prove that count distributes over addition…

```
forall a, b :: count(a + b) == count(a) + count(b)
```

Lemmas and Induction


```
lemma DistributiveLemma(a: seq<bool>, b: seq<bool>)
  ensures count(a + b) == count(a) + count(b)
{
}

function count(a: seq<bool>): nat
{
  if |a| == 0 then 0 else
    (if a[0] then 1 else 0) + count(a[1..])
}
```

- To prove the lemma, we need to structurally decompose the shape(s) of a and b
- We effectively need an inductive proof principle

Lemmas and Induction

Base case:

```
lemma DistributiveLemma(a: seq<bool>, b: seq<bool>)
  ensures count(a + b) == count(a) + count(b)
{
  if a == [] {
    assert a + b == b;
    assert count(a) == 0;
    assert count(a + b) == count(b);
    assert count(a + b) == count(a) + count(b);
  } else {
    //...
  }
}
function count(a: seq<bool>): nat
{
  if |a| == 0 then 0
  else (if a[0] then 1 else 0) + count(a[1..])
}
```

Lemmas and Induction

Inductive case:

```
lemma DistributiveLemma(a: seq<bool>, b: seq<bool>)
  ensures count(a + b) == count(a) + count(b)


function count(a: seq<bool>): nat
{
  if |a| == 0 then 0
  else (if a[0] then 1 else 0) + count(a[1..])
}
```

```
assert a + b == [a[0]] + (a[1..] + b);
assert count(a + b) == count([a[0]]) + count(a[1..] + b);
```

Lemmas and Induction

Express this inductive property:

```
assert count(a + b) == count([a[0]]) + count(a[1..] + b);
```

using recursion

```
lemma DistributiveLemma(a: seq<bool>, b: seq<bool>)
  ensures count(a + b) == count(a) + count(b)
{
  if a == [] {
    assert a + b == b;
  } else {
    DistributiveLemma(a[1..], b);
    assert a + b == [a[0]] + (a[1..] + b);
  }
}
function count(a: seq<bool>): nat
{
  if |a| == 0 then 0 else
    (if a[0] then 1 else 0) + count(a[1..])
}
```

Proof Calculations

Proof that Nil is idempotent over list appends

```
lemma prop_app_Nil(xs: list)
  ensures app(xs, Nil) == xs;
{
    match xs {
        case Nil =>
        case Cons(y,ys) =>
            calc { app(xs,Nil);
            == app(Cons(y,ys), Nil);
            == Cons(y, app(ys,Nil));
            == { prop_app_Nil(app(ys,Nil)); }  // proof hint
            xs;
      }
    }
}
```

Proof Calculations

Constructive proofs that involve rewrites and simplification

```
calc {
  (x + y) * (x - y);
==
  (x * x) - (x * y) + (y * x) - (y * y);
==
  (x * x) - (x * y) + (x * y) - (y * y);
==
  (x * x) - (y * y);
}
```

Adapted from slides of Bryan Parno

Proof Calculations

Proof that list append is associative

```
lemma prop_app_assoc(xs: list, ys: list, zs: list)
  ensures app(xs, app(ys, zs)) == app(app(xs, ys), zs);
{
    match xs {
        case Nil =>
            calc { app(Nil, app(ys, zs));
            == app(app(Nil, ys), zs);
            }
        case Cons(hd,tl) =>
            calc { app(Cons(hd,tl), app(ys, zs));
            == Cons(hd, app(tl, app(ys, zs)));
            == { prop_app_assoc(tl, ys, zs); }
            Cons(hd, app(app(tl, ys), zs));
            == app(app(Cons(hd,tl), ys), zs);
            == app(app(xs, ys), zs);
            }
    }
}
```

Proof Calculations and Induction


```
datatype Tree<T>  =
      Leaf(data : T)
        | Node(left: Tree<T>, right : Tree<T>)

function mirror<T>(t : Tree<T>) : Tree<T> {
    match t
        case Leaf(_) => t
        case Node(left, right) => Node(mirror(right), mirror(left))
}
```

```
lemma {:induction false} MirrorMirror<T>(t: Tree)
  ensures mirror(mirror(t)) = t
{
   // Proof that mirror is involutive
}
```

Proof Calculations and Induction


```
lemma {:induction false} MirrorMirror<T>(t: Tree)
    ensures mirror(mirror(t)) == t
    {
        match t
            case Leaf(_) =>
            case Node(left,right) =>
                calc
                {
                mirror(mirror(Node(left,right)));
                ==
                mirror(Node(mirror(right),mirror(left)));
                ==
                Node(mirror(mirror(left)),mirror(mirror(right)));
                ==  // IH
            { MirrorMirror(left);    MirrorMirror(right); }
            Node(left, right);
        }
}
```

Proof Calculations and Induction


```
function size<T>(t : Tree<T>): nat {
  match t
       case Leaf(_) => 1
       case Node(left, right) => size(left) + size(right)
}


lemma {:induction false} MirrorSize<T>(t : Tree<T>)
    ensures size(mirror(t)) == size(t)
{
    match t
        case Leaf(_) =>
        case Node(left, right) =>
            calc {
            size(mirror(Node(left,right)));
                 ==
            size(Node(mirror(right),mirror(left)));
                 ==
            size(mirror(right)) + size(mirror(left));
                 ==
            { MirrorSize(right); MirrorSize(left); } // I.H
            size(right) + size(left);
                 ==
             size(Node(left,right));
        }

}
```

Proofs by Contradiction

General shape:

$$\frac{\texttt{!Q -> (R /\ !R)}}{\texttt{Q}}$$

```
lemma Lem(args)
    requires P(x)
    ensures Q(x)
 {
    if !Q(x)                    // property is false
      {
        assert !P(x)            // contradiction: precondition is
        assert false            // true and false
    }
   assert Q(x)
}
```

slide adapted from Albert Nymeyer

Proof by Contradiction


```
lemma L(i : int)
  requires i > 42
  ensures i > 0
 {
   if i <= 0 {
     assert i <= 0 && i > 42;    // define contradiction
     assert false;
   }
 }
```

Dafny can verify this claim without the need of a lemma in this case, but in general the property may require reasoning beyond Dafny's ability to discharge automatically

slide adapted from Albert Nymeyer

Proof by Contradiction

Claim: if s is a singleton set, and i \in s, then s = { i }

```
lemma isSingle(s: set<int>, i: int)
  requires |s| == 1 && i in s
  ensures s == {i}
 {
   if s > {i}     // {i} is a subset of s
     {
        assert |s - {i}| >= 1 ==> |s| > 1;
        assert |s| == 1 && |s| > 1; // from precondition and branch
        assert false;
     }
 }
```

slide adapted from Albert Nymeyer

Proof by Contradiction

Given two disjoint sets A and B, if x \in A then x !\in B

```
lemma Disjoint(a: set<int>, b: set<int>, x: int)
  requires a * b == {}  // a and b are disjoint
  requires x in a
  ensures x !in b
 {
   if x in b       // setup contradiction
    {
      assert (x in b) && (x in a);  // from branch and pre-condition
      assert a*b == {x};  // from assert and precondition
      assert a*b == {x} && a*b == {}; // from assert and precondition
      assert false;
    }
}
```

slide adapted from Albert Nymeyer

Proof by Contradiction

Usage:

```
method Validate() {
  var a: set<int>, b: set<int>;
  var x: int;

  if x in a && a*b == {} {
    Disjoint(a, b, x);  // eliminating this proof causes the
    assert(x !in b);    // assert to fail
  }
}
```

slide adapted from Albert Nymeyer

Assume


```
method ExFalsoQuodlibet() {
  var f := true;
  assume !f   // let f be logically false, so f /\ !f
  var pigscanfly := false;
  assert pigscanfly;
}
```

Dafny will verify (but not compile) this program. Why?

```
method IdVerum() {
  var f := true;
  assume f;
  var pigscanfly := false;
  assert pigscanfly;
}
```

Dafny will not verify this program

```
method EsseVerum() {
  var f : bool;
  assume f;
  var pigscanfly := false;
  assert pigscanfly;
}
```

Dafny will not verify this program

slide adapted from Albert Nymeyer

Functional data structures

- In addition to support for inductive datatypes, Dafny also has specialized support for certain kinds of functional data structures, specifically sets and sequences.
- A set is an *order-less immutable* collection of *distinct* elements

  {2, 3, 3, 2} == {2, 2, 2, 3, 3, 3} == {2, 3}

  {2, 4, 4, 3, 5} == {5, 3, 4, 4, 2} == { 2, 3, 4, 5}
- Sets can be used in both specification and code

```
method Main()
      {
        var a: set<int> := {1,2,3,4};
        var b: set<int> := {4,3,2,1,1,2,3,4};
        assert |a| == |b|== 4;  // same length
        assert a - b == {};     // same sets
        print a, b;             // can print them
}
```

Set comprehension

- Can query and transform the elements of a set using set comprehension syntax

`set x:T | p(x) :: f(x)`

where `T` is the type of each element `x`

`p(x)` a predicate (that is, what e must satisfy)

`f(x)` an optional function (what is done to *x*)

```
var s: set<int> := {4,1,2,3,0};
var t := set e:int | e in s;          // 1. copy s to t
assert t == s;                        // verify
var u := set e:int | e in s && e>1; // 2. copy and filter out
assert u == {2,3,4};

// 3. copy and modify assert w == {1,2,3,4,5};
var w := set e:int | e in s :: e+1;

// verify
assert w == t + {5} – {0};
```

Triggers

Dafny tries to guess the set that matches the constraints defined by a comprehension

```
method Trigger() {
  var u := set e:int| 0<=e<5;
  assert u == {0,1,2,3,4};
}
```

```
Warning: /!\ No terms found to trigger on
```

```
method TriggerFixed() {
  var s: set<int> := {0,-1,1,-2,2,-3,3,-4,4,-5,5,-6,6};
  var u := set e:int| 0<=e<5 && e in s;
  assert u == {0,1,2,3,4};
}
```

```
Dafny program verifier finished with 3 verified, 0 errors
```

```
method SetMustBeFinite()
   {
      var t:set<nat> := set e:nat| e%2==0;
   }
```

Set operations

Assignment syntax overloaded to copy a set: `t := s` copies the contents of `s` to the object referenced by `t`.

Access an arbitrary element using filters

```
method Choose()
    {
        var s:set<nat> := {0,1,2,3,4,5,6,7,8,9};
        var x :| x in s && x%2==0; // non-det choice of x
        assert x==0 || x==2 || x==4 || x==6 || x==8;
    }
```

Non-deterministic selection
Existential instantiation

Existentials


```
method NDFind(a: array<int>, key:int) returns (x:nat)
   requires exists k:: 0<=k<a.Length && a[k]==key
   ensures  0<=x<a.Length && a[x]==key
 {
    x :| 0<=x<a.Length && a[x]==key;
 }
```

```
method Validator() {
   var a := new int[][0,42,0,42];
   assert a[3]==42; // why is this assert necessary?
   var ix := NDFind(a, 42);
   assert ix==1 || ix==3;
}
```

Example


```
method SetCopy(s: set<int>) returns(t: set<int>)
 ensures s==t
 {
    t := {};
    var ls := s;
    while |ls|>0
    invariant s == ls + t
    decreases |ls|
    {
       var e :|e in ls;
       ls := ls - {e};
       t := t + {e};
} }
```

Sequences

A sequence is an ordered immutable list of (possibly non-unique) elements

```
method SeqsAreOrdered() {
    var s: seq<int> := [2,1,3];
    var t: seq<int> := [1,2,3];
    assert s != t;
}
```

```
method CheckLength() {
  var a:array<int> := new int[][1,2,3,4];
  var s:set<int> := {1,2,3,4,4,3,2,1,1,1,1,1};
  var t:seq<int> := [1,2,3,4];
  assert a.Length == |s| == |t| == 4;
}
```

Arrays and Sequences

Arrays can be converted to sequences

```
method ArrayToSeq()
{
  var a: array<int> := new int[][1,42,1]; // an array
  var s: seq<int> := a[..];
  assert s == [1,42,1];

  var t: seq<int> := a[1..];
  assert t == [42,1];

  var b:array<int> := new int[2];
  b[0], b[1] := t[0], t[1];
  assert b[..]==[42,1];
}
```

Notation `a[..]` used to denote a slice of a sequence

Thus, `a[i..j]` defines a slice of a sequence containing elements

`a[i], a[i+1], …, a[j-1]`

Comprehensions

- Can initialize sequences explicitly or through the use of comprehensions

```
seq(k, nat=>expr)
```

that creates and initializes a sequence of length `k`

- values are obtained by evaluating `expr` on the indexes `0` up to `k`
- type `nat` represents the index

```
method SequenceInit()
{
  var odd:seq<int>  := [1, 3, 5, 7, 9];
  var odd2:seq<int> := seq(5, n=>2*n+1);

  var lue:seq<int>  := [42, 42, 42, 42, 42];
  var lue2:seq<int> := seq(5, _=>42);
}
```

Patching

Can selectively update/copy sequences using patch notation sequence `s[i := v]` equals `s[..i] + [v] + s[i+1..]`

```
method SeqPatch()
// left slice + new + right slice element at index 3 needs patching
{
  var s:seq<int> := [10,20,30,42,50];
  var t := s[..3] + [40] + s[4..]; // a patch using slicing
  assert t == [10,20,30,40,50] == s[3 := 40]; // normal patch notation

  var g:seq<int>; // general case: for any sequence g and any value v
  var v:int;
  assert forall i:: 0<=i<|g| ==> g[i := v] == g[..i] + [v] + g[i+1..];
}
```

Multisets

A multiset is:

- like a set insofar as its elements are unordered
- like a sequence insofar as it admits duplicates
  - the multiplicity of each element is recorded

```
method MultisetFromElements() {
  var m1: multiset<int> := multiset{1,1,1,42};
  var m2: multiset<int> := multiset{42,1,1,1};
  var m3: multiset<int> := multiset{1,1,42,1};
  assert m1==m2==m3;
}
```

```
method MultisetFromSeqs() {
  var a:string := "loops";
  var b:string := "spool";
  var ma := multiset(a);
  var mb := multiset(b);
  assert ma == mb;
}
```

```
method Multiplicity() {
  var word: seq<char> := "hippopotomonstrosesquipedaliophobia";
  assert |word| == 35;
  assert !('c' in word);
  assert !exists i:: 0<=i<|word| && word[i]=='c';

  assert multiset(word)['c'] == 0;
  assert 'o' in word;
  assert multiset(word)['o'] == 7;
  assert word[..9] == "hippopoto";
  assert multiset(word[..9])[‘p'] == 3;
```

Higher-Order Functions


```
function method CountHelper<T>(P: T -> bool, a: seq<T>, i: int): int
  requires 0 <= i <= |a|
  decreases |a| - i
{
  if i == |a|
  then 0
  else (if P(a[i]) then 1 else 0) + CountHelper(P, a, i+1)
}
```

```
function method Count<T>(P: T -> bool, a: seq<T>) : int
{
  CountHelper(P, a, 0)
}


method Main()
{
  var a := new int[10] (i => i);
  var evens := Count(x => x % 2 == 0, a);
  print evens, "\n";
  var bigs := Count(x => x >= 5, a);
  print bigs, "\n";
}
```

Example

Proving properties about paths in a graph

```
class Node
{
  // a single field giving the nodes linked to
  var next: seq<Node>
}

predicate closed(graph: set<Node>)
  reads graph
{
  forall i :: i in graph ==>
    forall k :: 0 <= k < |i.next| ==> i.next[k] in graph && i.next[k] != i
}

predicate path(p: seq<Node>, graph: set<Node>)
   requires closed(graph) && 0 < |p|
   reads graph
{
   p[0] in graph &&
   (|p| > 1 ==> p[1] in p[0].next && // the first link is valid, if it exists
      path(p[1..], graph)) // and the rest of the sequence is a valid
}
```

The reads clause defines a frame that constrains the portion of memory a predicate or method is allowed to read. In this case, the clause asserts that every node in the 'next' field of a node in the graph is part of the graph.

Example

Define a path between a start and end node:

```
predicate pathSpecific(p: seq<Node>, start: Node, end: Node,
                       graph: set<Node>)
   requires closed(graph)
   reads graph
{
   0 < |p| && // path is nonempty
   start == p[0] && end == p[|p|-1] && // it starts and ends correctly
   path(p, graph) // and it is a valid path
}
```

Example

A lemma that asserts all paths in a closed subgraph of a graph can never contain nodes not found in the subgraph:

```
lemma ClosedLemma(subgraph: set<Node>, root: Node,
                  goal: Node, graph: set<Node>)
    requires closed(subgraph) && closed(graph) && subgraph <= graph
    requires root in subgraph && goal in graph - subgraph
    ensures !(exists p: seq<Node> :: pathSpecific(p, root, goal, graph))
{
    …
}
```

To prove the non-existence of a property, we need to show that it does not hold for any valid path. We can use another lemma for this purpose.

Example


```
lemma DisproofLemma(p: seq<Node>, subgraph: set<Node>,
                    root: Node, goal: Node, graph: set<Node>)
  requires closed(subgraph) && closed(graph) && subgraph <= graph
  requires root in subgraph && goal in graph - subgraph
  ensures !pathSpecific(p, root, goal, graph)
{
  …
}
```

```
lemma ClosedLemma(subgraph: set<Node>, root: Node,
                  goal: Node, graph: set<Node>)
   requires closed(subgraph) && closed(graph) && subgraph <= graph
   requires root in subgraph && goal in graph - subgraph
   ensures !(exists p: seq<Node> :: pathSpecific(p, root, goal, graph))

{
  forall p {
     DisproofLemma(p, subgraph, root, goal, graph);
   }
}
```

Example

How can a path be invalid?

- it is empty (but the precondition precludes this)
- it contains a root and a goal and:
    * the root is in subgraph and goal is in graph
    * so the path has at least two elements

```
lemma DisproofLemma(p: seq<Node>, subgraph: set<Node>,
                    root: Node, goal: Node, graph: set<Node>)
  requires closed(subgraph) && closed(graph) && subgraph <= graph
  requires root in subgraph && goal in graph - subgraph
  ensures !pathSpecific(p, root, goal, graph)
{
    if 1 < |p| && p[0] == root && p[|p|-1] == goal {
       (further proof)
  }
}
```

Example

Leverage induction:

- for the postcondition to be true, there must be a sub-path that contains consecutive elements such that the first is in the subgraph and the second is not.

```
lemma DisproofLemma(p: seq<Node>, subgraph: set<Node>,
                    root: Node, goal: Node, graph: set<Node>)
  requires closed(subgraph) && closed(graph) && subgraph <= graph
  requires root in subgraph && goal in graph - subgraph
  ensures !pathSpecific(p, root, goal, graph)
{
    if 1 < |p| && p[0] == root && p[|p|-1] == goal {
     if p[1] in p[0].next {
      DisproofLemma(p[1..], subgraph, p[1], goal, graph);
    }
  }
}
```

Frames (Imperative Reasoning)

- A method specification must describe the method is allowed to modify and what it leaves unchanged, in addition to describing the actual modification:
  - A method that modifies the contents of its input array must declare this via a modifies clause

```
method F(a: array<int>, left: int, right: int)
   requires a.Length != 0
   modifies a
{
   a[0] := left;
   a[a.Length - 1] := right;
}
```

Two-State Predicates

- Specifications for imperative programs often need to relate the value of a structure in the pre-state (before the method executes) and the post-state (after the method completes).

- Use old(E) to refer to the value of E in the prestate
  - old tracks heap dereferences

```
method Increment(a : array<int>, i: int)
    requires 0 <= i < a.Length
    modifies a
    ensures a[i] == old(a)[i] + 1
{
    a[i] := a[i] + 1;
}
```

VS.

```
method Increment(a : array<int>, i: int)
    requires 0 <= i < a.Length
    modifies a
    ensures a[i] == old(a[i]) + 1
{
    a[i] := a[i] + 1;
}
```

Read Clauses

- Delineates the portion of memory a function is allowed to read
  - Used to determine if updates to the mutable portion of the heap can invalidate a specification

```
predicate isZeroArray(a : array<int>, lo:int, hi:int)
    requires 0 <= lo <= hi < a.Length
    reads a
    decreases hi - lo
   {
      lo == hi || (a[lo] == 0 && IsZeroArray(a, lo+1, hi);
 }
```

```
predicate isZeroSeq(a : seq<int>, lo:int, hi:int)
    requires 0 <= lo <= hi < |a|
    decreases hi - lo
  {
     lo == hi || (a[lo] == 0 && IsZeroSeq(a, lo+1, hi);
}
```

No reads clause required

Bubble Sort


```
method BubbleSort(a: array<int>)
{
  var i := a.Length - 1;
  while(i > 0)
   {
     var j := 0;
     while(j < i) {
      if (a[j] > a[j+1]) {
        a[j], a[j+1] := a[j+1], a[j];
       }
      j:=j+1;
     }
    i := i - 1
   }
}
```

How do we go about proving this implementation is correct?

- Whenever the method terminates, the contents of a are in sorted ascending order

Following slides adapted from Matt Fredrickson

Specification


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

Given this predicate, the pre- and post-conditions for the method can be easily given:

```
requires a != null
ensures sorted(0, a, a.Length -1)
```

Basic Invariants


```
method BubbleSort(a: array<int>)
  requires a != null
  ensures sorted(0, a, a.Length -1)
{
  var i := a.Length - 1;
  while(i > 0)
    invariant -1 <= i < a.Length
    {
      var j := 0;
      while(j < i)
        invariant 0 < i < a.Length && 0 <= j <= i
      {
       if (a[j] > a[j+1]) {
         a[j], a[j+1] := a[j+1], a[j];
        }
       j:=j+1;
      }
     i := i - 1
    }
}
```

Clearly insufficient to prove the postcondition since they say nothing about a!

More Invariants

Observe that after each iteration of the outer loop the elements from i to a.Length - 1 are sorted:

```
sorted(a, i, a.Length-1)
```

Strengthening the inner invariant requires more thought …

One observation:

- every element at index greater than i is no smaller than any element at index i or less

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

Existing Proof


```
method BubbleSort(a: array<int>)
  requires a != null
  ensures sorted(0, a, a.Length -1)
{
  var i := a.Length - 1;
  while(i > 0)
    invariant -1 <= i < a.Length
    invariant sorted(a, i, a.Length -1)
    invariant partitioned(a, i)
    {
      var j := 0;
      while(j < i)
        invariant 0 < i < a.Length && 0 <= j <= i
        invariant sorted(a, i, a.Length -1)
        invariant partitioned(a, i)
   {
       if (a[j] > a[j+1]) {
         a[j], a[j+1] := a[j+1], a[j];
        }
       j:=j+1;
      }
     i := i - 1
    }
}
```

Dafny is unable to verify this program using just these invariants. Note that the invariants for the inner loop never refer to j!

More Invariants


```
method BubbleSort(a: array<int>)
  requires a != null
  ensures sorted(0, a, a.Length -1)
{
  var i := a.Length - 1;
  while(i > 0)
    invariant -1 <= i < a.Length
    invariant sorted(a, i, a.Length -1)
    invariant partitioned(a, i)
   {
    var j := 0;
    while(j < i)
      invariant 0 < i < a.Length && 0 <= j <= i
      invariant sorted(a, i, a.Length -1)
      invariant partitioned(a, i)
      invariant forall k::0<=k<=j==>a[k]<=a[j]

  {
     if (a[j] > a[j+1]) {
       a[j], a[j+1] := a[j+1], a[j];
      }
     j:=j+1;
    }
   i := i - 1
  }
}
```

Dafny complains that the post-condition might not hold upon exit from the loop. Why?

More Invariants

- Two possibilities for i on exit:
  - i = 0 and the array has been sorted (that's what the loop invariants in the two loops establish)
  - i = -1: the array is empty and the loop never gets executed. Sorted holds trivially.

    Establishing this fact requires knowing that a.Length = 0 when i = -1

```
method BubbleSort(a: array<int>)
  requires a != null
  ensures sorted(0, a, a.Length -1)
{
  var i := a.Length - 1;
  while(i > 0)
    invariant -1 <= i < a.Length
    invariant i < 0 ==> a.Length == 0
    invariant sorted(a, i, a.Length -1)
    invariant partitioned(a, i)
   {
     var j := 0;
     while(j < i)
       invariant 0 < i < a.Length && 0 <= j <= i
       invariant sorted(a, i, a.Length -1)
       invariant partitioned(a, i)
  {
      if (a[j] > a[j+1]) {
        a[j], a[j+1] := a[j+1], a[j];
       }
      j:=j+1;
    }
    i := i - 1
   }
}
```

Strengthening the Spec

- We've proven that BubbleSort produces a sorted array

- We haven't shown that this sorted array has anything to do with the input array, though!

- Our specification should be strengthened to assert that the output array is a sorted permutation of the input array.

```
method BubbleSort(a: array<int>)
  modifies a
  requires a != null
  ensures sorted(0, a, a.Length -1)
  ensures permutation(a[..], old(a[..]))
```

Permutations

An array a of type T is a permutation of an array b if for every value v in T, the number of occurrences of v in a is the same as in b

```
function count(a: seq<int>, v: int): nat {
  if(|a| > 0) then
    if(a[0] == v) then 1 + count(a[1..], v)
    else count(a[1..], v)
  else 0
}


predicate permutation(a: seq<int>, b: seq<int>) {
  forall v :: count(a, v) == count(b, v)
}
```

Strengthening Invariants

To validate the post-condition, Dafny needs to establish that modifications to the array:

```
a[j], a[j+1] := a[j+1], a[j];
```

is permutation-preserving

Introduce ghost variables to track modifications to the array:

```
ghost var a' := a[..];
a[j], a[j+1] := a[j+1], a[j];
assert permutation(a[..], a');
```

and add an additional loop invariant to both loops:

```
invariant perm(old(a[..]), a[..])
```

Strengthening Invariants

- Dafny is unable to generalize a permutation on a pair of elements `j` and `j+1` to a property on the entire array

- Make this relation explicit

```
ghost var a' := a[..];
a[j], a[j+1] := a[j+1], a[j];
ghost var v1, v2 := a[j], a[j+1];
assert a[..] == a[..j] + [v1, v2] + a[j+2..];
assert a' == a[..j] + [v2, v1] + a[j+2..];
assert permutation([v1, v2], [v2, v1]);
assert permutation(a[..], a');
```

Dafny doesn't accept the generalization argument

Culprit: permutations are defined in term of counts on sequences, but the assertions involving reasoning over concatenation of sequences.

Need to show that:

```
forall v :: count(a + b + c, v) == count(a, v) + count(b, v) + count(c, v)
```

Distributive Property of Count


```
function count(a: seq<int>, v: int): nat {
  if (|a| > 0) then
    if (a[0] == v) then 1 + count(a[1..], v)
       else count(a[1..], v)
  else 0
}

lemma count_dist(a: seq<int>, b: seq<int>)
  ensures forall v :: count(a + b, v) == count(a, v) + count(b, v)
{
    forall (v: int)
       ensures count(a + b, v) == count(a, v) + count(b, v)
       {
       }
}
```

Distributive Property of Count


```
if(a == []) {
      calc {
         count(a + b, v);
         == { assert a + b == b; }
            count(b, v);
         == 0 + count(b, v);
         == { assert count(a, v) == 0; }
            count(a, v) + count(b, v);
      }
}
```

Distributive Property of Count


```
else {
 calc {
   count(a + b, v);
   == { assert a + b == [a[0]] + (a[1..] + b); }
      count([a[0]] + (a[1..] + b), v);
   == (if a[0] == v then 1 + count(a[1..] + b, v)
                    else count(a[1..] + b, v));
   == { count_dist(a[1..], b); }
      (if a[0] == v then 1 + count(a[1..], v) + count(b, v)
                    else count(a[1..], v) + count(b, v));
   == count(a, v) + count(b, v);
  }
}
```

Distributive Property of Count

Express this inductive property:

```
assert count(a + b) == count([a[0]]) + count(a[1..] + b);
```

using recursion

```
lemma DistributiveLemma(a: seq<bool>, b: seq<bool>)
  ensures count(a + b) == count(a) + count(b)
{
  if a == [] {
    assert a + b == b;
  } else {
    DistributiveLemma(a[1..], b);
    assert a + b == [a[0]] + (a[1..] + b);
  }
}
function count(a: seq<bool>): nat
{
  if |a| == 0 then 0 else
    (if a[0] then 1 else 0) + count(a[1..])
}
```