CS 565

Programming Languages (graduate)
Spring 2024

Week 13

Data Structures and Proof Development

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..])
}
```