CS 565

Programming Languages (graduate)
Spring 2024

Week 12

Dafny Proof Techniques

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

Decreases clause


```
function seqSum (s : seq<int>, lo : int, hi : int) : int
    requires 0 <= lo <= hi <= |s|
{
    if (lo == hi) then 0 else s[lo] + seqSum(s, lo+1, hi)
}
```

Dafny complains that it cannot prove the recursive call terminates - it is unable to identify a termination metric that signals every recursive call gets "smaller"

```
function seqSum (s : seq<int>, lo : int, hi : int) : int
    requires 0 <= lo <= hi <= |s|
    decreases hi - lo
{
    if (lo == hi) then 0 else s[lo] + seqSum(s, lo+1, hi)
}
```

What about using -lo as a decreases clause?

Examples


```
function F(x : int) : int {
  if x < 10 then x else F(x - 1)

}
```

Are decreases clauses required?

```
function G(x : int) : int {
   if 0 <= x then G(x - 2) else x
}
```

```
function H(x : int) : int
  decreases x + 60
{
   if x < -60 then x else H (x - 1)
}
```

Are decreases clauses required here?

Why would `decreases x` not work?

```
function L(x: int) : int
  decreases 100 - x
{
   if x < 100 then L(x + 1) + 10 else x
}
```

Are decreases clauses required here?

Well-Founded Relations

A binary relation $\succeq$ is well-founded if it is:

- irreflexive
- transitive
- satisfies a descending chain condition, i.e. there is no infinite sequence of values a0, a1, … such that a0 $\succeq$ a1 $\succeq$ …

Can establish such a relation for any datatype

- E.g., Booleans (true $\succeq$ false), a less-than ordering relation on integers, or a subset relation on a set

```
function M(x: int, b: bool) : int
  decreases if b then 0 else 1
{
  if b then x else M(x + 25, true)
}
```

Lexiographic Tuples

- Component-wise comparison of decreases clauses:

$4, 12 \succeq 4, 11$

$4, 6, 0 \succeq 4, 6, 0, 25, 3$

$2, 5 \succeq 1$

What decreases clause is necessary to allow Dafny to verify this program?

```
function Ack(m : nat, n: nat) : nat
   decreases m, n
{
   if m == 0 then
     n + 1
   else if n == 0 then
     Ack(m - 1, 1)
   else Ack(m - 1, Ack(m, n - 1))
}
```

Midterm

Mean: 68.93

Median: 68.65

Max: 118.65

Distribution:

> 100:	4	(A+)	- 8%
80 - 100:	10	(A)	- 20%
70 - 80:	6	(A-)	- 12%
50 - 70:	20	(B)	- 29%
< 50:	15	(B-)	- 30%

Regrade requests open for one week.

Midterm solutions will be posted on Piazza later today.

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

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 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

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