Lexical Scope

software engineering

CATEGORY ARCHIVES: DAFNY

Dafny Sum and Max solution

Posted on **April 10, 2015**, Modified on April 11, 2015

A Dafny solution to the Sum and Max verification benchmark. I added some additional specification to verify that the result `max` is really the maximum and the `sum` is really the sum.

```
method SumMax(a: array<int>) returns (sum:int, max:int)
 requires a != null;
 requires a.Length >= 0;
 requires forall i :: 0 <= i < a.Length ==> a[i] >= 0;
 ensures sum <= a.Length*max;
 ensures forall i :: 0 <= i < a.Length ==> a[i] <= max;
 ensures a.Length == 0 ||
         (exists i :: 0 <= i < a.Length && a[i] == max);
 ensures sum == Sum(a);
{
 max := 0;
 sum := 0;

 var idx:int := 0;
 while idx<a.Length
  invariant idx <= a.Length;
  invariant forall i :: 0 <= i < idx ==> a[i] <= max;
  invariant (idx==0 && max==0) ||
            exists i :: 0 <= i < idx && a[i] == max;
  invariant sum <= idx*max;
  invariant sum == Sum'(a, 0, idx);
 {
  if (max < a[idx])
  {
   max := a[idx];
  }
  sum := sum + a[idx];
  idx := idx + 1;
 }
}

function Sum(a: array<int>) : int
```

```
 reads a;
 requires a != null;
{
 Sum'(a, 0, a.Length)
}

function Sum'(a: array<int>, i:int, j:int) : int
 reads a;
 requires a != null;
 requires 0 <= i <= j <= a.Length;
{
 if j == i then 0 else Sum'(a, i, j-1) + a[j-1]
}

lemma SumOfIntervalOneIsElement(a: array<int>, i:int)
 requires a != null;
 requires 0 <= i < a.Length;
 ensures a.Length > 0;
 ensures Sum'(a, i, i+1) == a[i];
{ }

lemma SumOfIntervalIsSumOfSubIntervals
  (a: array<int>, i:int, j:int, k:int)
 requires a != null;
 requires 0 <= i <= j <= k <= a.Length;
 ensures Sum'(a, i, j) + Sum'(a, j, k) == Sum'(a, i, k);
{ }
```

Posted in **dafny** | **Leave a reply**

Distributivity of Sequence Map over Function Composition in Dafny

Posted on **January 21, 2015**, Modified on January 21, 2015

Distributivity of Sequence Map over Function Composition in Dafny

```
lemma MapDistributivity(xs:seq<int>, f:int->int, g:int->int)
   requires forall x :: x in xs ==> f.requires(x);
   requires forall x :: x in xs ==> g.requires(f(x));
   ensures forall x :: x in MapSeq(xs,f) ==> g.requires(x);
   ensures MapSeq(MapSeq(xs,f), g) == MapSeq(xs, Compose(f,g));
{
   if xs != []
   {
     MapDistributivity(xs[1..], f, g);
   }
}

function Compose(f:int->int, g:int->int) : int->int
{
```

```
  x
    reads f.reads(x)
    reads if f.requires(x) then g.reads(f(x)) else {}
    requires f.requires(x)
    requires g.requires(f(x))
      -> g(f(x))
}

function MapSeq(xs:seq<int>, f:int->int) : seq<int>
   reads MapSeqReads(xs, f);
   requires forall x :: x in xs ==> f.requires(x);
   ensures |xs| == |MapSeq(xs,f)|;
   ensures forall x :: x in xs ==> f(x) in MapSeq(xs,f);
{
  if xs == [] then []
  else [f(xs[0])] + MapSeq(xs[1..], f)
}

function MapSeqReads(xs:seq<int>, f:int->int) : set<object>
   reads if |xs| > 0 then f.reads(xs[0]) + MapSeqReads(xs[1..], f) else {};
   decreases xs;
   ensures forall x :: x in xs ==> f.reads(x) <= MapSeqReads(xs,f);
{
  if xs == [] then {}
  else f.reads(xs[0]) + MapSeqReads(xs[1..],f)
}
```

Posted in **dafny** | **Leave a reply**

Dafny: Permutations, sequences and multisets

Posted on **November 11, 2014**, Modified on November 11, 2014

Dafny has support for multisets, which are really useful for reasoning about programs that permute, or otherwise deal with permutations, of sequences or arrays.

```
predicate Subpermutation(xs:seq, ys:seq)
   ensures Subpermutation(xs,ys) ==> forall x :: x in xs ==> x in ys;
{
  assert forall x :: x in xs ==> x in multiset(xs);
  multiset(xs) <= multiset(ys)
}

// THEOREM
lemma SubpermutationIsSmaller(xs:seq, ys:seq)
  requires Subpermutation(xs,ys);
  ensures |xs| <= |ys|;
{
  assert |multiset(xs)| == |xs|;
  assert |multiset(ys)| == |ys|;

  var xs',ys' := xs,ys;
```

```
  var XS',YS' := multiset(xs),multiset(ys);
  var XS'',YS'' := multiset{},multiset{};
  while(|XS'|>0)
     invariant Subpermutation(xs',ys');
     invariant XS' == multiset(xs');
     invariant YS' == multiset(ys');
     invariant XS' + XS'' == multiset(xs);
     invariant YS' + YS'' == multiset(ys);
     invariant XS'' == YS'';
     invariant XS' <= YS';
  {
    assert RemoveFromSequenceReducesMultiSet(xs,multiset(xs),multiset(xs[1..]))
    var x := xs'[0];
    xs' := Remove(x,xs');

    ys' := Remove(x,ys');

    XS' := XS'[x := XS'[x] - 1];
    XS'' := XS''[x := XS''[x] + 1];
    YS' := YS'[x := YS'[x] - 1];
    YS'' := YS''[x := YS''[x] + 1];
  }
}

// SUPPORTING DEFINITIONS

// following is a function lemma
predicate RemoveFromSequenceReducesMultiSet(xs:seq<T>, XS:multiset<T>, XS':mul
   requires xs != [];
   requires XS' == multiset(xs[1..]);
   ensures XS' == multiset(xs)[xs[0] := multiset(xs)[xs[0]] - 1];
   ensures RemoveFromSequenceReducesMultiSet(xs,XS,XS');
{
  assert [xs[0]]+xs[1..] == xs;
  assert multiset([xs[0]]+xs[1..]) == multiset(xs);
  true
}

function Remove(x:T, xs:seq<T>) :seq<T>
   requires x in xs;
   ensures multiset(Remove(x,xs)) == multiset(xs)[x := multiset(xs)[x] - 1];
   ensures |Remove(x,xs)|+1 == |xs|;
{
     assert RemoveFromSequenceReducesMultiSet(xs,multiset(xs),multiset(xs[1..]
     if xs[0]==x
     then xs[1..]
     else [xs[0]] + Remove(x,xs[1..])
}
```

Posted in **dafny** | **Leave a reply**

Dafny Function Lemmas

Posted on **August 17, 2014**, Modified on October 8, 2014

Sometimes it is hard to use a normal Dafny method lemma where you need it. For example, if you want to use a lemma inside a function or in a contract. There is a simple solution.

Given a method lemma:

```
static lemma SomethingValid(s:S)
   requires P(s);
   ensures Q(s);
```

We can produce a function lemma:

```
static lemma SomethingValidFn(s:S)
   requires P(s);
   ensures Q(s);
   ensures SomethingValidFn(s);
{
   SomethingValid(s); true
}
```

Posted in **dafny** | **Leave a reply**

Dafny: Proving forall x :: P(x) ==> Q(x)

Posted on **July 31, 2014**, Modified on October 8, 2014

The general method in Dafny for proving something of the form `forall x :: P(x) ==> Q(x)` is:

```
lemma X()
    ensures forall x :: P(x) ==> Q(x);
{
    forall x | P(x)
        ensures Q(x);
    {
       // prove Q(x)
    }
}
```

Posted in **dafny** | **Leave a reply**

Using {:verify false} in Dafny

Posted on **May 20, 2014**, Modified on May 20, 2014

Due to the nature of the underlying SMT used by Dafny for proof checking, Dafny sometimes takes a very long time to verify things. I find it useful to ask Dafny to work on a single lemma at a time. This can be achieved using the *{:verify false}* annotation. I set every function and method in my proof to *{:verify false}*, then change only the one I am working on back to *{:verify true}*. Occasionally I use find and replace to change all the *{:verify false}* annotations to *{:verify true}* and then run the whole proof through the Dafny command line.

Posted in **dafny** | **Leave a reply**

Opaque functions in Dafny (performance/proveablity,readbility)

Posted on **May 15, 2014**, Modified on May 22, 2014

I am using Opaque functions in Dafny quite heavily, and for three main reasons:

OPAQUE FUNCTIONS CAN IMPROVE PERFORMANCE

I have a very large predicate which judges partial isomorphism between two program states. This predicate appears in the requires, ensures and bodies of a large number of lemmas and functions in my proof. The number of facts that are introduce by Dafny automatically revealing these function definitions seems to substantially slow down the proof step, sometimes to the point where it no longer goes through in an amount of time I am willing to wait.

For example take a predicate like this

```
prediate I(a:T,b:T)
{
   P(a,b) && Q(a,b) && R(a,b) && S(a,b)
}
predicate P(a:T,b:T) { ... }
predicate Q(a:T,b:T) { ... }
...
static lemma IThenSomethingThatReliesOnQ(a:T,b:T)
  requires I(a,b);
  ensures SomethingThatReliesOnQ(a,b);
{
   // Q automatically revealed, but so is P,R and S
}
```

Instead we can write

```
prediate I(a:T,b:T)
{
   P(a,b) && Q(a,b) && R(a,b) && S(a,b)
```

```
}
predicate {:opaque true} P(a:T,b:T) { ... }
predicate {:opaque true} Q(a:T,b:T) { ... }
...
static lemma IThenSomethingThatReliesOnQ(a:T,b:T)
  requires I(a,b);
  ensures SomethingThatReliesOnQ(a,b);
{
   reveal_Q(); // reveals Q only, not P,R and S
}
```

OPAQUE FUNCTIONS CAN IMPROVE READABILITY

I initially really liked the automatic function definition revealing feature, but as my proof got larger I liked it less. Particularly in the presence of big predicates (like my isomorphism predicate), I found it can get quite hard to understand which parts of the proof rely on which facts. This made it harder for me to work out what I need to establish in other parts of the proof (in this case, that the isomorphism has certain properties and that those can be used to show it is preserved by some particular program execution steps).

The example above shows how using opaque and reveal makes the proof more self documenting, allows us to automatically check that documentation and provides (I think) more insight into the proof.

OPAQUE FUNCTIONS HELP YOU FIND PLACES THAT NEED RE-VERIFYING

If you are using {:verify false} to restrict re-verification only to the elements you are working on, then having functions set to opaque helps you find all the places that depend on their definition. So, if you change the definition then you can easily find all the places that will need to be re-verified with the new definition.

Posted in **dafny** | **Leave a reply**

Naming quantifiers in Dafny

Posted on **May 15, 2014**, Modified on May 15, 2014

It seems to help Dafny with instantiation of quantifiers during proofs if you give the quantified expression a name by encapsulating it in a predicate. If you don't you can end up in a situation where you prove some quantified property, but are then unable to subsequently assert that quantified property, I think due to Dafny not understanding that the quantification is the same. Naming the property (or more likely, only writing it in one place rather than two places) fixes this.

For example rather than writing:

```
  ...
  ensures forall c :: P(a, c) == P(b, c);
  ...
```

```
assert forall c :: P(a, c) == P(b, c);
```

Write this:

```
predicate Q(a:S,b:S)
{ forall c :: P(a, c) == P(b, c) }
...
ensures Q(a, b);
...
assert Q(a, b);
```

Posted in **dafny** | **Leave a reply**

Building Dafny Visual Studio Extension

Posted on **April 28, 2014**, Modified on April 12, 2015

You will need the Visual Studio 2012 SDK installed you can tell this because Visual Studio tells you the project is incompatible and refuses to open it. But if you use a text editor to look at the associated .csproj file, that file has the GUID “82b43b9b-a64c-4715-b499-d71e9ca2bd60” in the “project types” section. And this GUID means “needs the visual studio SDK”.

You probably need to build boogie first. You may find that you need to enable nuget restore for the boogie solution, in order to get visual studio to download the correct version of nunit.

Then you need to open the solution Dafny.sln in the source directory, and build it with visual studio. Then build the visual studio extension DafnyExtension.sln. Then in “Extensions and Updates” you can uninstall the DafnyLanguageMode if you already have it. Then you can install Binaries\DafnyLanguageService.vsix to get the VS extension.

Also, on the Dafny VS extension color codes. I think perhaps: yellow means “line changed since last save”; orange means “line changed since the last verify run”; pink means “line currently being verified”; and green means line verified.

Documentation

1. Dafny type system documentation

Posted in **dafny** | **1 Reply**

Inverting Maps in Dafny

Posted on **April 17, 2014**, Modified on August 28, 2014

I had cause to need to prove some things about injective maps and inverses.

```
// union on maps does not seem to be defined in Dafny
function union<U, V>(m: map<U,V>, m': map<U,V>): map<U,V>
        requires m !! m'; // disjoint
        ensures forall i :: i in union(m, m') <==> i in m || i in m';
        ensures forall i :: i in m ==> union(m, m')[i] == m[i];
        ensures forall i :: i in m' ==> union(m, m')[i] == m'[i];
{
        map i | i in (domain(m) + domain(m')) :: if i in m then m[i] else m'[i
}

// the domain of a map is the set of its keys
function domain<U,V>(m: map<U,V>) : set<U>
        ensures domain(m) == set u : U | u in m :: u;
        ensures forall u :: u in domain(m) ==> u in m;
{
                set u : U | u in m :: u
}

// the domain of a map is the set of its values
function range<U,V>(m: map<U,V>) : set<V>
        ensures range(m) == set u : U | u in m :: m[u];
        ensures forall v :: v in range(m) ==> exists u :: u in m && m[u] == v;
{
        set u : U | u in m :: m[u]
}

// here a map m is smaller than m' if the domain of m is smaller than
// the domain of m', and every key mapped in m' is mapped to the same
// value that it is in m.
predicate mapSmaller<U,V>(m: map<U,V>, m': map<U,V>)
        ensures mapSmaller(m,m') ==>
                (forall u :: u in domain(m) ==> u in domain(m'));
{
        forall a :: a in m ==> a in m' && m[a] == m'[a]
}

// map m is the inverse of m' if for every key->value in m
// there is value->key in m', and vice versa
predicate mapsAreInverse<U,V>(m: map<U,V>, m': map<V,U>)
{
        (forall a :: a in m ==> m[a] in m' && m'[m[a]] == a) &&
        (forall a :: a in m' ==> m'[a] in m && m[m'[a]] == a)
}

// map m is injective if no two keys map to the same value
predicate mapInjective<U,V>(m: map<U,V>)
{
        forall a,b :: a in m && b in m ==> a != b ==> m[a] != m[b]
}

// here we prove that injective map m has an inverse, we prove
```

```
// this by calculating the inverse for an arbitrary injective map.
// maps are finite in Dafny so we have no termination problem
lemma invertMap<U,V>(m: map<U,V>) returns (m': map<V,U>)
        requires mapInjective(m);
        ensures mapsAreInverse(m,m');
{
        var R := m;     // part of m left to invert
        var S := map[]; // part of m already inverted
        var I := map[]; // inverted S

        while R != map[]       // while something left to invert
                decreases R;   // each loop iteration makes R smaller
                invariant mapSmaller(R, m);
                invariant mapSmaller(S, m);
                invariant R !! S; // disjoint
                invariant m == union(R, S);
                invariant mapsAreInverse(S,I);
        {
                var a :| a in R;   // take something arbitrary in R
                var v := R[a];
                var r := map i | i in R && i != a :: R[i];  // remove a from R
                I := I[v:=a];
                S := S[a:=v];
                R := r;
        }
        m' := I;  // R is empty, S == m, I inverts S
}

// here we prove that every injective map has an inverse
lemma injectiveMapHasInverse<U,V>(m: map<U,V>)
        requires mapInjective(m);
        ensures exists m' :: mapsAreInverse(m, m');
{
    var m' := invertMap(m);
}

// here we prove that no non-injective map has an inverse
lemma nonInjectiveMapHasNoInverse<U,V>(m: map<U,V>)
        requires !mapInjective(m);
        ensures !(exists m' :: mapsAreInverse(m, m'));
{ }

// here we prove that if m' is the inverse of m, then the domain of m
// is the range of m', and vice versa
lemma invertingMapSwapsDomainAndRange<U,V>(m: map<U,V>, m': map<V,U>)
        requires mapsAreInverse(m, m');
        ensures domain(m) == range(m') && domain(m') == range(m);
{ }

// a map m strictly smaller than map m' has fewer elements in its domain
lemma strictlySmallerMapHasFewerElementsInItsDomain<U,V>(m: map<U,V>, m': map<U
        requires mapSmaller(m,m') && m != m';
        ensures domain(m') - domain(m) != {};
{
        var R,R' := m,m';
        while R != map[]
                decreases R;
                invariant mapSmaller(R,R');
                invariant R != R';
        {
                var a :| a in R && a in R';
```

```
                var v := R[a];

                var r := map i | i in R && i != a :: R[i];
                var r' := map i | i in R' && i != a :: R'[i];

                R := r;
                R' := r';
        }
        assert R == map[];
        assert R' != map[];

        assert domain(R) == {};
        assert domain(R') != {};
}

function invert<U,V>(m:map<U,V>) : map<V,U>
        requires mapInjective(m);
        ensures mapsAreInverse(m,invert(m));
{
        injectiveMapHasInverse(m);

        var m' :| mapsAreInverse(m,m');
        m'
}
```

Posted in **dafny** | **Leave a reply**