Commented Dafny Programming Problems

(CVS 14-15, MIEI, DI / FCT / UNL)

Luís Caires

(version 1.0 of April 12, 2015)

1 Introduction

In this note, we discuss some programming exercises and comment on some useful tips for Dafny programming, and code verification in general.

2 maxvec

```
// return the maximum of the values in array a[-]
// stored in positions i such that 0 <= i < a.Length
// a.Length > 0

method maxvec(a:array<int>) returns (m:int)
requires a != null;
requires a.Length > 0;
ensures forall j :: (0 <= j < a.Length) ==> a[j] <=  m ;
ensures exists j :: (0 <= j < a.Length) && a[j] == m ;
{
 var i:int := 1;
 var t:int := a[0];
 while (i != a.Length)
 invariant 0 < i <= a.Length;
 invariant forall j :: ( 0 <= j < i) ==> a[j] <= t ;
 invariant exists j :: ( 0 <= j < i) && a[j] == t ;
 {
   if (a[i] > t) {  t:=a[i]; }
   i := i + 1;
 }
 m := t;
}
```

2.1 Comments

- The first post condition states that m is $\geq$ than any elements in the array, but this does not ensure m is actually the maximum. For that the second post condition is needed, stating that m actually belongs to the array.

- To help Dafny check complex postconditions in programs with while loops, you often need to provide explicit loop invariants. This will make Dafny happy. As the first invariant, you should always (and note: this is a rule you should always follow) start by defining the condition that expresses the interval of variation of the loop control variable. In this case, we write `0<i<=a.Length`.
- To explain Daffy how the loop works to establish the method postconditions, you should then write invariants for the loop, that approximate the post conditions as the iteration progresses (see lectures). At the entry of every ith iteration we know that `forall j::(0<=j<i) ==> a[j]<=t`. This states that the "current" maximum (stored in `t`) is greater that all elements in the array up to (but excluding) position `i`. We also know that at the entry of every ith iteration `exists j::(0<=j<i) && a[j]==t`. This states that the "current" maximum (stored in `t`) is actually present in the array somewhere up to (but excluding) position `i`.
- Notice that indeed all loop invariants are valid at loop entry by making

  ```
  var i:int := 1;
  var t:int := a[0];
  ```

  They are also valid at each iteration and at loop exit (Dafny checks this, using Hoare logic). Then, since at loop exit `i==a.Length`, Dafny deduces that the post conditions are valid when the method terminates.

3 filled

```
// the method filled returns true if and only if
// the first n elements of array are all set to the value v

method filled(a:array<int>, v:int, pos:int) returns (s:bool)
requires a != null;
requires 0 <= pos <= a.Length;
ensures s <==> forall j :: 0<=j<pos ==> a[j]== v;
{
var i:int := 0;
while (i < pos)
invariant (0 <= i <= pos);
invariant forall j :: 0<=j<i ==> a[j]==v ;
        {
          if (a[i] != v) {  s := false; return; }
          i := i + 1;
}
s := true;
}
```

3.1 Comments

- This is an extremely simple example, just to illustrate the introduction of invariants in loops. In this case, the invariant states that all positions of

the already visited initial segment of the array, up to position `i`, contain the value `v`.

4 find

```
// if there is an index j ( 0 <= j < n )
// such that a[j] == v then return true
// otherwise return false
// n cannot exceed the allocated array size

method find(a:array<int>, v:int, n:int) returns (found:bool)
requires a != null && 0 <= n <= a.Length;
ensures found <==> exists j :: 0 <= j < n && a[j] == v;
{
var i : int := 0;
while (i<n)
invariant 0 <= i <= n;
invariant forall j :: 0 <= j < i ==> a[j] != v;
{
  if(a[i] == v) {
    return true;
  }
  i := i + 1;
}
return false;
}
```

4.1 Comments

- The key invariant states that all elements up to (and excluding `i`) are different from `i`, that is, `forall j :: 0<=j<i ==> a[j]!=v`. Hence, when arriving to the end of the loop we can return `false`, to satisfy the post-condition. On the other end, if at some point the loop finds `v`, it may immediately return `true`, and satisfy the post condition as well.
- Notice that in Dafny there is no need to assign the return value to the `found` variable, the return statement does the job. So `return true` means the same as `found := true; return`.

5 prime

```
// the method prime checks if a number is prime
function prime(n:int):bool
{ n >= 0 && forall i :: (2 <= i < n)  ==> (n % i != 0) }

method isprime(n:int) returns (p:bool)
requires n>=0;
ensures p <==> prime(n);
{
```

```
var i : int := 2;
while (i < n/2)
invariant forall j :: (2<= j < i) ==> (n % j != 0) ;
{
     if (n % i == 0) { return false; }
     i := i + 1;
}
return true;
}
```

5.1 Comments

- This example is very similar to the find operation above: to check if a number is prime we just have to find some divisor.
- Notice the the use of a defined function in an assertion. In Dafny, functions are always pure and cannot have side effects. In fact, they are considered "ghost", only used by the verifier, and do not generate code. If you really need to call a function from actual (non "ghost" code) you should define it as function method, e.g.,:

  ```
  function method prime(n:int):bool
  { n >= 0 && forall i :: (2 <= i < n)  ==> (n % i != 0) }
  ```

6 reverse

```
// the method reverse should return a copy of array a
// but with the first nelems elements by reverse order

method reverse(a:array<int>, nelems:int) returns (b:array<int>)
requires a != null;
requires 0 <= nelems < a.Length;
ensures b != null;
ensures b.Length == nelems;
ensures forall k:int :: 0<=k<nelems ==> b[k]==a[nelems-k-1);
{
 b := new int[nelems];
 var i:int := 0;
 while(i < nelems)
 invariant 0 <= i <= nelems;
 invariant forall k:int :: 0<=k<i ==> b[k]==a[nelems-k-1];
 {
    b[i] := a[nelems-i-1];
    i := i + 1;
 }
}
```

6.1 Comments

- There is no much to comment here, but this example illustrates how to create a new array. We take the opportunity to challenge you to write a variation of this method where the array would be reversed in place.

7 incvec

```
// the method incvec increments in place the
// first nelem elements of array a

method incvec(a:array<int>, nelems:int)
requires a != null;
requires 0 <= nelems < a.Length;
ensures forall k:int :: 0<=k<nelems ==> a[k]==1+old(a[k]);
modifies a;
{
 var i:int := 0;
 while(i < nelems)
 modifies a;
 invariant 0 <= i <= nelems;
 invariant forall k:int :: 0<=k<i ==> a[k]==1+old(a[k]);
 invariant forall k:int :: i<=k<nelems ==> a[k]==old(a[k]);
 {
    a[i] := a[i] + 1 ;
    i := i + 1;
 }
}
```

7.1 Comments

- This example illustrates issues related to changing global structures in place through side effects, and how to express invariants that capture ongoing manipulations on such structures.

- Notice that the array `a` is changed in place by the method and by the loop. In this case, the method declaration requires a so-called "frame conditions" to be expressed, in this case via a `modifies` clause.

- Besides the usual invariants, any loop that changes the memory it accesses may also specify the footprint using a `modifies` clause.

- This examples also illustrates the use of the specification `old(-)` construct. For any Dafny expression `e`, the expression `old(e)` denotes the initial value of expression `e` when the envolving method was called. This is useful to specify side effects in post-conditions, as is the case of the `incvec` method. Another example of using old may be:

  ```
  class Up {
  var c:int;
  method increase(d:int)
  ```

```
requires d>0 && c>=0;
modifies this;
ensures c > old(c);
{
 c := c+d;
}
}
```

- Notice that the invariants must say that at any iteration step, the array positions up to `i` have been incremented with respect to the initial value, while the positions from `i` to the end of the array preserve their initial values. Both conditions are needed to allow Dafny to reason about the effect of the loop body. Convince yourself that both invariants are really needed (hint: if the second invariant is omitted, it would be possible to write code in the loop body that changes some position above i without breaking the first invariant, yielding inchrrect code). Dafny forces the programmer to fully specfy the stronger conditions to ensure the method post-condition.

8 sort

```
// the  method sorts array b in place, up to position nelems
method sort(b:array<int>, nelems:int)
modifies b;
requires initarray(b,nelems);
ensures  sorted(b,nelems);
```

We first define a few auxiliary functions, useful for expressing several assertions.

```
// array c is created with sixe enough for nelems
function initarray(c:array<int>,nelems:int):bool
{
c!=null && 0<=nelems<=c.Length
}
// first nelems elements are sorted
function sorted(c:array<int>, nelems:int):bool
requires initarray(c,nelems);
reads c;
{
forall i:: (0<=i<nelems) ==> forall j::(i<j<nelems) ==> c[i]<=c[j]
}
// first i elems are less or equal than elems from i to nelems
function majors(c:array<int>,i:int,nelems:int):bool
requires initarray(c,nelems);
reads c;
{
forall k::0<=k<i ==> forall l::i<=l<nelems ==> (c[k] <= c[l])
}
```

Here is now our code for the `sort` method, using the selection sort algorithm.

```
method sort(b:array<int>, nelems:int)
modifies b;
requires initarray(b,nelems);
ensures  sorted(b,nelems);
{
  var i:int := 0;
  while (i<nelems)
  invariant 0<=i<=nelems;
  invariant sorted(b,i) && majors(b,i,nelems);
  {
    var j := i ;
    while (j < nelems)
    invariant i<=j<=nelems;
    invariant sorted(b,i) && majors(b,i,nelems);
    invariant forall k:: i<=k<j ==> b[i]<= b[k];
    {
      if( b[i] > b[j] ) {
        var s : int := b[i];
        b[i] := b[j];
        b[j] := s;
      }
      j := j+1;
    }
    i := i+1;
  }
}
```

8.1 Comments

- Writing auxiliary functions, such as e.g., `sorted` and `majors` above, to abstract assertions highly contributes to make Dafny code highly readable and modular. Just compare the declaration above of the `sort` method with the equivalent following one, much less readable.

  ```
  method sort(b:array<int>, nelems:int)
  modifies b;
  requires b!=null && 0<=nelems<=b.Length;
  ensures forall i:: 0<=i<nelems ==>
           forall j::i<j<nelems ==> b[i]<=b[j]
  ```

- Notice that help Dafny handle two nested loops, we need to propagate the invariants of the outer loop to the inner loop. This is because the inner loop changes the array in ways that could possible break the invariant of the outer loop.

- The postcondition defined above does not ensure that the final array contents is a permutation of the initial array contents. As an exercise you may change the code above so that it would actually express such a stronger postcondition.

9 strf

```
// the method strf checks if b appears within a, and returns
// in pos the position if that is the case, or -1 if not
// n is the number of chars in a, m is the number of chars in b

function ins(k: int, i:int): bool
{ 0 <= k < i }

method strf(a:seq<int>, n:int, b:seq<int>, m:int) returns (pos:int)
requires 0 <= n <= |a|;
requires 0 <= m <= |b|;
requires n >= m;
ensures -1 <= pos <= n - m;
ensures pos >= 0 ==> forall k::0 <= k < m ==> b[k]==a[pos+k];
ensures pos ==-1 ==> forall k::ins(k,n-m+1) ==>
                       exists p:: 0<=p<m && b[p] != a[k+p];
{
  ghost var f:seq<int> := [];
  var i:int := 0;
  while (i < n-m+1)
   invariant 0 <= i <= n-m+1;
   invariant |f| == i;
   invariant forall k::0<=k<i ==> 0<=f[k]<m;
   invariant forall k:: ins(k,i) ==> b[f[k]] != a[k+f[k]];
  {
    var j:int :=0;
    while (j < m)
    invariant 0 <= j <= m;
    invariant forall k:: 0<=k<j ==> b[k] == a[i+k];
    {
      if (a[i+j] != b[j]) { break; }
      j := j+1;
    }
    if (j==m) { return i; }
    f := f + [j];
    i := i+1;
  }
  return -1;
}
```

9.1 Comments

- This program illustrates two interesting advanced verification techniques: the use of "ghost" variables in the specifications, and the use of helper functions to help Dafny prove the correctness of forall / exists assertions.
- There are two postconditions to check. The first talks about the case where the substring is found (`pos >= 0`). In this case, the postcondition must simply say that the string `b` occurs at position `pos` in `a`.

- For the other case (`pos == -1`), the postcondition must say that the string `b` occurs nowhere in `a`. To express that, we need a more complex forall / exists assertion, namely (X)

  ```
  forall k::in(k,n-m+1) ==> exists p::0<=p<m && b[p] != a[k+p];
  ```

  This says that for any k such that `0 <= k < n-m+1` there is a position `p` with `0<=p<m` that is different in `b` and `a` from position `k`, so that the string `b` does not occur in `a` at position `k`.

- Now, notice that the existential just asserts that there is some index p that makes the occurrence fail, but does not say what is the actual value of such `p`. The assertion is abstract on that, it just says that for any `k` such that bla bla there is some `p` (depending on `k`) such that etc, etc.

- In such situations, Dafny asks the programmer to actually provide some hint about what are the actual values of such positions `p`'s. Of course returning such concrete positions is not needed for the algorithm to work, neither needed by client code, but just to make clear the reasoning envolved to show that the code is correct. We can see such values of `p` as the concrete "proof" that shows that the string `b` does not occur at each position `k` in string `a`, in case the client code would like to check that `strf` is working fine.

- We then make them explicit using so-called "ghost" variables and "ghost" code. Such "ghost" variables and "ghost" code is not compiled (so it is not executed at runtime, exactly like all assertions) and belong to the world of assertions, is just checked at compile time, to make explicit the values of the existentially quantified variables in forall / exist assertions. Such values are technically called "witnesses".

- In the example above, the witnesses are stored in the ghost sequence `f`. The main idea is that for each `k`, let `f[k]` store the position p at which b and the substring of a that starts at k differ. So, for any `k` such that `0<=k<i`, `f[k]` stores the value of the `p` mentioned in `exists::p`, referred in the final postcondition (X) above. The values `f[0]` ... `f[i-1]` witness the fact that string `b` does not occur at each position `0..i-1` in string `a`, and help Dafny to resolve the existentials in assertion (X) above. The key properties of sequence `f` is established by the invariants

  ```
  invariant |f| == i;
  invariant forall k::0<=k<i ==> 0<=f[k]<m;
  invariant forall k:: ins(k,i) ==> b[f[k]] != a[k+f[k]];
  ```

10 to be continued ...