CS:5810 Formal Methods in Software Engineering

Reasoning about Programs with Arrays in Dafny

Part II

Reading arrays in functions

If a function/predicate accesses the elements of an input array a*,*
its specification must include reads a

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

Modifying arrays

If a method modifies values accessible through reference parameters (and stored in the heap),
its specification must identify the relevant parts of the heap using frames

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

modifies clause

If a method changes the elements of an input array a*,*
its specification must include `modifies a`

```
method Aliases(a: array<int>, b: array<int>)
  requires 100 <= a.Length
  modifies a
{
  a[0] := 10;
  var c := a;
  if b == a {
    b[10] := b[0] + 1;    // ok since b == a
  }
  c[20] := a[14] + 2;     // ok since c == a
}
```

old qualifier

The expression `old(E)` *denotes the value of* `E` *on entry to the enclosing method*

```
method UpdateElements(a: array<int>)
  requires a.Length == 10
  modifies a
  ensures old(a[4]) < a[4]
  ensures a[6] <= old(a[6])
  ensures a[8] == old(a[8])
{
  a[4], a[8] := a[4] + 3, a[8] + 1;
  a[7], a[8] := 516, a[8] - 1;
}
```

old qualifier

`old` *affects only the heap dereferences in its argument*

For example, in

```
method OldVsParameters(a: array<int>, i: int)
returns (y: int)
  requires 0 <= i < a.Length
  modifies a
  ensures old(a[i] + y) == 25
```

only `a` is interpreted in the pre-state of the method

New arrays

A method is allowed to allocate a new array and change its elements without mentioning the array in the `modifies` *clause*

```
method NewArray() returns (a: array<int>)
  ensures a.Length == 20
{
  a := new int[20];
  var b := new int[30];
  a[6] := 216;
  b[7] := 343;
}
```

Fresh arrays

```
method Caller()
{
  var a := NewArray();
  a[8] := 512;      // error: modification of a not allowed
}
```

To fix error, strengthen specification of `NewArray` to

```
method NewArray() returns (a: array<int>)
  ensures fresh(a)
  ensures a.Length == 20
```

Initializing arrays

```
method InitArray<T>(a: array<T>, d: T)
  modifies a
  ensures forall i :: 0 <= i < a.Length ==> a[i] == d
{
  var n := 0;
  while n != a.Length
    invariant 0 <= n <= a.Length
    invariant forall i :: 0 <= i < n ==> a[i] == d
  {
    a[n] := d;
    n := n + 1;
  }
}
```

Incrementing the values in an array

```
method IncrementArray(a: array<int>)
  modifies a
  ensures forall i :: 0 <= i < a.Length ==> a[i] == old(a[i]) + 1
{
  var n := 0;
  while n != a.Length
    invariant 0 <= n <= a.Length
    invariant forall i :: 0 <= i < n ==> a[i] == old(a[i]) + 1

  {
    a[n] := a[n] + 1;
    n := n + 1;
  }  // error: second loop invariant not maintained
}
```

Incrementing the values in an array

```
method IncrementArray(a: array<int>)
  modifies a
  ensures forall i :: 0 <= i < a.Length ==> a[i] == old(a[i]) + 1
{
  var n := 0;
  while n != a.Length
    invariant 0 <= n <= a.Length
    invariant forall i :: 0 <= i < n ==> a[i] == old(a[i]) + 1
    invariant forall i :: n <= i < a.Length ==> a[i] == old(a[i]) // needed
  {
    a[n] := a[n] + 1;
    n := n + 1;
  }
}
```

We need to add the invariant that elements not yet visited maintain the old value

Copying arrays

```
method CopyArray<T>(a: array<T>, b: array<T>)
  requires a.Length == b.Length
  modifies b
  ensures forall i :: 0 <= i < a.Length ==> b[i] == old(a[i])
{
  var n := 0;
  while n != a.Length
    invariant 0 <= n <= a.Length
    invariant forall i :: 0 <= i < n ==> b[i] == old(a[i])
    invariant forall i ::
                0 <= i < a.Length ==> a[i] == old(a[i])
  { b[n] := a[n];
    n := n + 1;
  }
}
```