CS:5810 Formal Methods in Software Engineering

Reasoning about Programs with Arrays in Dafny

Arrays are references

Type of a is array<string>

```
var a := new string[20];
a[7] := "hello";
var b := a;
assert b[7] == "hello";

b[7] := "hi";
a[8] := "greetings";
assert a[7] == "hi" && b[8] == "greetings";
```

Arrays are references

Type of a is `array<string>`

```
var a := new string[20];
a[7] := "hello";
var b := a;
assert b[7] == "hello";

b[7] := "hi";
a[8] := "greetings";
assert a[7] == "hi" && b[8] == "greetings";

b := new string[8];
b[7] := "long time, no see";
assert a[7] == "hi";
assert a.Length == 20 && b.Length == 8;
```

Multi-dimensional arrays

```
var m := new bool[3, 10];
m[0, 9] := true;
m[1, 8] := false;
assert m.Length0 == 3 && m.Length1 == 10;
```

Type of m is `array2<bool>`

Sequences

Arrays are mutable and are *reference types*

Sequences are immutable and are *value types*, like bool and int

To declare a sequence we use type constructor seq,
e.g., seq<bool>, seq<int>

Examples:

[]	the empty sequence
[58]	singleton integer sequence
["hey", "hola", "tjena"]	string sequence

Sequences

```
var s := [6, 28, 496];
assert s[2] == 496;
assert |s| == 3;   // length function
assert s + [8128] == [6, 28, 496, 8128];

var p := [1, 5, 12, 22, 35]
assert p[2..4] == [12, 22];
assert p[..2] == [1, 5];
assert p[2..] == [12, 22, 35];

a := new int[3];
a[0], a[1], a[2] := 6, 28, 496;
s, p := a[..], a[..2];
assert s == [6, 28, 496] && p == [6, 28];
```

Linear search

```
method LinearSearch<T>(a: array<T>, P: T -> bool)
returns (n: int)
```

Predicate on T

Linear search

```
method LinearSearch<T>(a: array<T>, P: T -> bool)
returns (n: int)
  ensures 0 <= n <= a.Length
  ensures n == a.Length || P(a[n])
```

Predicate on T

Linear search

```
method LinearSearch<T>(a: array<T>, P: T -> bool)
returns (n: int)
  ensures 0 <= n <= a.Length
  ensures n == a.Length || P(a[n])
{
  n := 0;
  while n != a.Length
    invariant 0 <= n <= a.Length




}
```

Linear search

```
method LinearSearch<T>(a: array<T>, P: T -> bool)
returns (n: int)
  ensures 0 <= n <= a.Length
  ensures n == a.Length || P(a[n])
{
  n := 0;
  while n != a.Length
    invariant 0 <= n <= a.Length
  {
    if P(a[n])
      { return; }
    n := n + 1;
  }
}
```

return jumps to end of method, and we need to prove postconditions

Alternative implementation

```
method LinearSearch1<T>(a: array<T>, P:T -> bool)
returns (n: int)
  ensures 0 <= n <= a.Length
  ensures n == a.Length || P(a[n])
{
  n := a.Length;
}
```

Alternative implementation

```
method LinearSearch1<T>(a: array<T>, P:T -> bool)
returns (n: int)
  ensures 0 <= n <= a.Length
  ensures n == a.Length || P(a[n])
{
  n := a.Length;
}
```

To specify that no elements satisfy P, when `n == a.Length` we need to quantify over the elements of a.

We can achieve the same effect by quantifying over the array positions instead:

```
forall i :: 0 <= i < a.Length ==> !P(a[i])
```

Strengthening the contract

```
method LinearSearch1<T>(a: array<T>, P:T -> bool)
returns (n: int)
  ensures 0 <= n <= a.Length
  ensures n == a.Length || P(a[n])
  ensures n == a.Length ==>
            forall i :: 0 <= i < a.Length ==> !P(a[i])
```

can leave off i’s type
since it can be inferred

Strengthening the contract

```
method LinearSearch1<T>(a: array<T>, P:T -> bool)
returns (n: int)
  ensures 0 <= n <= a.Length
  ensures n == a.Length || P(a[n])
  ensures n == a.Length ==>
            forall i :: 0 <= i < a.Length ==> !P(a[i])
```

We use the “replace a constant by a variable”
loop design technique 6.1:

```
  invariant forall i :: 0 <= i < n ==> !P(a[i])
```

Linear search

```
{ forall i :: 0 <= i < n + 1 ==> !P(a[i]) }
n := n + 1;
{ forall i :: 0 <= i < n ==> !P(a[i]) }
```

Linear search

```
{ forall i :: (0 <= i < n || i == n) ==> !P(a[i]) }
{ forall i :: 0 <= i < n + 1 ==> !P(a[i]) }
n := n + 1;
{ forall i :: 0 <= i < n ==> !P(a[i]) }
```

Linear search

```
forall x :: (A || B) ==> C
= (forall x :: A ==> C) && (forall x :: B ==> C)
```

```
{ (forall i :: 0 <= i < n ==> !P(a[i])) &&
  (forall i :: i == n ==> !P(a[i]))
}
{ forall i :: (0 <= i < n || i == n) ==> !P(a[i]) }
{ forall i :: 0 <= i < n + 1 ==> !P(a[i]) }
n := n + 1;
{ forall i :: 0 <= i < n ==> !P(a[i]) }
```

Linear search

```
(forall x :: x == E ==> A) = A[x\E]
```
(one-point rule)

```
{ (forall i :: 0 <= i < n ==> !P(a[i])) && !P(a[n]) }
{ (forall i :: 0 <= i < n ==> !P(a[i])) &&
  (forall i :: i == n ==> !P(a[i]))
}
{ forall i :: (0 <= i < n || i == n) ==> !P(a[i]) }
{ forall i :: 0 <= i < n + 1 ==> !P(a[i]) }
n := n + 1;
{ forall i :: 0 <= i < n ==> !P(a[i]) }
```

Linear search

holds due to invariant

```
{ (forall i :: 0 <= i < n ==> !P(a[i])) && !P(a[n]) }
{ (forall i :: 0 <= i < n ==> !P(a[i])) &&
  (forall i :: i == n ==> !P(a[i]))
}
{ forall i :: (0 <= i < n || i == n) ==> !P(a[i]) }
{ forall i :: 0 <= i < n + 1 ==> !P(a[i]) }
n := n + 1;
{ forall i :: 0 <= i < n ==> !P(a[i]) }
```

holds after `if P(a[n]) { return; }`

Linear search

```
{ (forall i :: 0 <= i < n ==> !P(a[i])) && !P(a[n]) }
{ (forall i :: 0 <= i < n ==> !P(a[i])) &&
  (forall i :: i == n ==> !P(a[i]))
}
{ forall i :: (0 <= i < n || i == n) ==> !P(a[i]) }
{ forall i :: 0 <= i < n + 1 ==> !P(a[i]) }
n := n + 1;
{ forall i :: 0 <= i < n ==> !P(a[i]) }
```

Loop body for `LinearSearch` works here

Full program

```
method LinearSearch1<T>(a: array<T>, P:T -> bool)
returns (n: int)
  ensures 0 <= n <= a.Length
  ensures n == a.Length || P(a[n])
  ensures n == a.Length ==>
               forall i :: 0 <= i < a.Length ==> !P(a[i])
{
  n := 0;
  while n != a.Length
    invariant 0 <= n <= a.Length
    invariant forall i :: 0 <= i < n ==> !P(a[i])
  {
    if P(a[n]) { return; }
    n := n + 1;
  }
}
```

Finding the first element

```
method LinearSearch2<T>(a: array<T>, P:T -> bool)
returns (n: int)
  ensures 0 <= n <= a.Length
  ensures n == a.Length || P(a[n])
  ensures forall i :: 0 <= i < n ==> !P(a[i])
```

The second and third postconditions imply that `n` is the *smallest index* such that `a[n]` satisfies `P`

The loop specification and body of `LinearSearch1` satisfy this contract too

Knowing it is there

If we can assume that at least one element of a satisfies P we can simplify the contract to

```
method LinearSearch3<T>(a: array<T>, P:T -> bool)
returns (n: int)
  requires exists i :: 0 <= i < a.Length && P(a[i])
  ensures 0 <= n < a.Length && P(a[n])
```

An invariant that says where to look

The element we are looking for is at index n or higher

```
invariant exists i ::
            n <= i < a.Length && P(a[i])
```

holds after
```
if P(a[n])
  { return; }
```

holds due to invariant on entry to loop

```
{ !P(a[n]) && exists i ::
               n <= i < a.Length && P(a[i]) }
{ exists i :: n + 1 <= i < a.Length && P(a[i]) }
n := n + 1;
{ exists i :: n <= i < a.Length && P(a[i]) }
```

Implementation of LinearSearch3

```
method LinearSearch3<T>(a: array<T>, P: T -> bool)
returns (n: int)
  requires exists i :: 0 <= i < a.Length && P(a[i])
  ensures 0 <= n < a.Length && P(a[n])
{
  n := 0;
  while true
    invariant 0 <= n < a.Length
    invariant exists i :: n <= i < a.Length && P(a[i])
    decreases a.Length - n
  {
    if P(a[n]) { return; }
    n := n + 1;
  }
}
```

simplified since n never reaches a.Length

Exercises

1. Write a linear-search specification for the method

```
method LinearSearch4<T>(a: array<T>, P: T -> bool)
returns (n: int)
```

that always returns a value strictly less than `a.Length` and uses a negative value (instead of `a.Length`) to signal that no element satisfies P.

Implement the specification.

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))
}
```

`reads a` states that the function may read only the memory locations in the heap occupied by `a`

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 a` states that the function may modify only the memory locations in the heap occupied by `a`

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;
  }
}
```