

CS:5810 Formal Methods in Software Engineering

Reasoning about Programs with Arrays in Dafny

Copyright 2020-23, Graeme Smith and Cesare Tinelli.

Produced by Cesare Tinelli at the University of Iowa from notes originally developed by Graeme Smith at the University of Queensland. These notes are copyrighted materials and may not be used in other course settings outside of the University of Iowa in their current form or modified form without the express written permission of one of the copyright holders. During this course, students are prohibited from selling notes to or being paid for taking notes by any person or commercial firm without the express written permission of one of the copyright holders.

Arrays are references

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

Type of a is array<string>

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

Arrays are references

```
var a := new string[20];           Type of a is array<string>
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;
```

Two arrays

```
var m := new bool[3, 10];
```

Type of m is array2<bool>

```
m[0, 9] := true;
```

```
m[1, 8] := false;
```

```
assert m.Length0 == 3 && m.Length1 == 10;
```

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])
```



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:

```
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

$(\text{forall } x :: x == E ==> A) = A[x \setminus E]$ (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