

Practical Formal Methods

Reasoning about Arrays and Objects in Dafny

Copyright 2020-22, 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];  
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;
```

Type of a is
array<string>

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", "salut" ]
```

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; }      return jumps to end of
      n := n + 1;      method, and we need to
    }                  prove the postcondition
}
```

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 seen before:

```
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; }`
(i.e., now we know that `!P(a[n])` holds)

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;
  }
}
```

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 (but not `y`)

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 (why?)
}
```

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

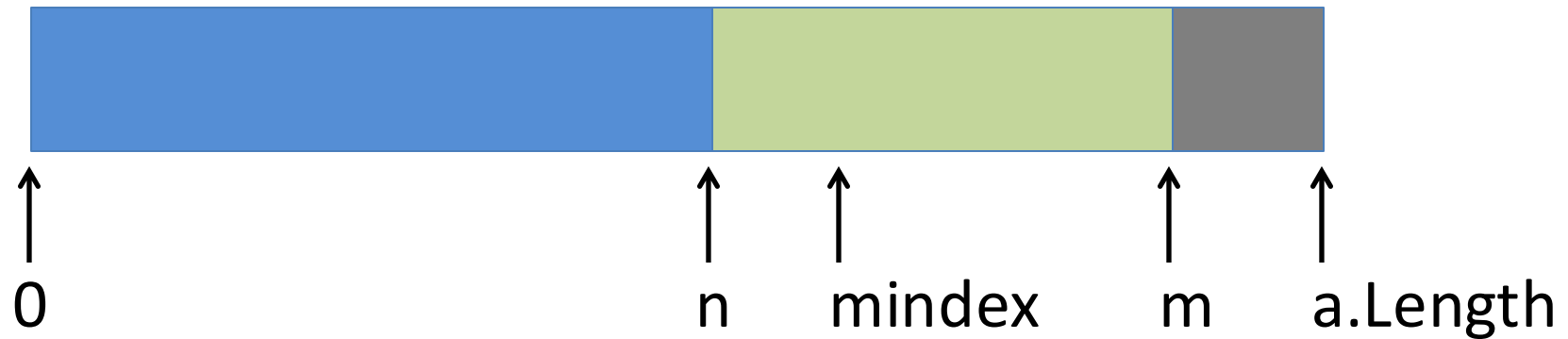
```
method CopyArray<T>(a: array<T>, b: array<T>)
  requires a.Length == b.Length
  modifies b
  ensures forall i :: 0 <= i < a.Length ==> b[i] == old(a[i])
{
  var n := 0;
  while n != a.Length
    invariant 0 <= n <= a.Length
    invariant forall i :: 0 <= i < n ==> b[i] == old(a[i])
    invariant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


Minimum

- See the code *Min.dfy*

Selection sort

- See the code *SelectionSort.dfy*



QuickSort

- See the code *QuickSort.dfy*

Reasoning about Objects

Checksum Objects

An *object* is an instance of a *class*, and like arrays, has a *reference type*

```
class ChecksumMachine {  
  var data: string  
  
  constructor ()  
    ensures data == ""  
  
  method Append(d: string)  
    modifies this  
    ensures data == old(data) + d  
  
  function Checksum(): int  
    reads this  
    ensures Checksum() == Hash(data)  
  
  ...  
}
```

string is shorthand
for seq<char>

Checksum Objects

...

```
function Hash(s: string): int {  
    SumChars(s) % 137  
}
```

```
function SumChars(s: string): int {  
    if |s| == 0 then 0 else  
        var last := |s| - 1;  
        SumChars(s[..last]) + s[last] as int  
    }  
}
```

converts char to int



Test client

```
method Main() {  
    var m := new ChecksumMachine();  
    m.Append("green ");  
    m.Append("grass");  
    var c := m.Checksum();  
    print "Checksum is ", c, "\n";  
}
```

*A method is allowed to allocate new arrays and objects and change their state (that is, the elements of the arrays and the fields of the objects) without mentioning these arrays and objects in the **modifies** clause*

Class Invariant

To write efficient implementation, want to keep track of checksum so far:

```
var cs: int
```

We want to use data in specifications, but not in compiled program:

```
ghost var data: string
```

```
predicate Valid(d)  
  reads this  
{ cs == Hash(d) }
```

A predicate is a
Boolean function

If a function accesses the fields of an object o, its specification must include
reads o

Class Invariant

```
class ChecksumMachine {  
  ghost var data: string  
  
  predicate Valid()  
    reads this  
  
  constructor ()  
    ensures Valid(data) && data == ""  
  
  method Append(d: string)  
    requires Valid(data)  
    modifies this  
    ensures Valid data && data == old(data) + d  
  
  function method Checksum(): int  
    requires Valid(data)  
    reads this  
    ensures Checksum() == Hash(data)  
}
```

Implementation

```
method Append(d: string)
  requires Valid(data)
  modifies this
  ensures Valid(data)
  ensures data == old(data) + d
{
  var i := 0;
  while i != |d|
    invariant 0 <= i <= |d|
    invariant Valid(data)
    invariant data == old(data) + d[..i]
    {
      cs := (cs + d[i] as int) % 137;
      data := data + [d[i]];
      i := i + 1;
    }
}
```

Implementation

```
constructor ()  
  ensures Valid(data) && data == ""  
{ cs := 0;  
  data := ""  
}
```

*A constructor is allowed to assign to the fields of the object being constructed, **this**, without mentioning **this** in the **modifies** clause*

```
function method Checksum(): int  
  requires Valid(data)  
  reads this  
  ensures CheckSum() == Hash(data)  
{ cs }
```

Summary

- Verification Dafny works by means of computing *weakest preconditions* and discharging proof obligations via SMT
- Loops require explicit *invariants* and *termination measures* (aka *variants, ranking functions*)
- Object invariants should be preserved by each method

Abstracting Object State via Ghost Fields

Coffee maker components

```
class Grinder {  
  var HasBeans: bool
```

```
  predicate Valid()  
    reads this
```

```
  constructor ()  
    ensures Valid()
```

```
  method AddBeans()  
    requires Valid()  
    modifies this  
    ensures Valid() && HasBeans
```

```
  method Grind()  
    requires Valid() && HasBeans  
    modifies this  
    ensures Valid()  
}
```

```
class WaterTank {  
  var Level: nat
```

```
  predicate Valid()  
    reads this
```

```
  constructor ()  
    ensures Valid()
```

```
  method Fill()  
    requires Valid()  
    modifies this  
    ensures Valid() && Level == 10
```

```
  method Use()  
    requires Valid() && Level != 0  
    modifies this  
    ensures Valid() && Level == old(Level) - 1  
}
```

```
class Cup {  
  constructor ()
```

```
}
```

Coffee maker version 0

```
class CoffeeMaker {  
  predicate Valid() reads this  
  
  constructor () ensures Valid()  
  
  predicate method Ready()  
    requires Valid()  
    reads this  
  
  method Restock()  
    requires Valid()  
    modifies this  
    ensures Valid() && Ready()  
  
  method Dispense(double: bool) returns (c: Cup)  
    requires Valid() && Ready()  
    modifies this  
    ensures Valid()  
}
```

Coffee maker version 0

State:

```
var g: Grinder
```

```
var w: WaterTank
```

```
predicate Valid()
```

```
  reads this
```

```
{ g.Valid() && w.Valid() } // error: insufficient  
                          // reads clause
```

Require:

```
predicate Valid()
```

```
  reads this, g, w
```

Similar change also needed for **reads** of Ready() and **modifies** clauses of Restock and Dispense

Representation sets

The expanded modifies and reads clauses violate the principles of information hiding.

Therefore, we abstract the state of an object to a *representation set*.

For this implementation of the coffee maker, the representation set is

$$\{o, o.g, o.w\}$$

but the coffee maker may also be implemented in terms of different objects.

Coffee maker version 1

Add new variable to state:

```
ghost var Repr: set<object>
```

Change modifies clauses of Restock and Dispense to

```
modifies Repr
```

Change read clauses of Valid and Ready to

```
reads Repr
```

Add the following to the body of Valid

```
this in Repr &&  
g in Repr && g.Valid() &&  
w in Repr && w.Valid()
```

Typically specify
lower bound on
objects in Repr

Coffee maker version 1

In Valid:

```
reads Repr // error: insufficient reads clause
```

This is because `this` is not in Repr unless Valid's predicate holds (and Valid may return `true` or `false`).

We require:

```
predicate Valid()  
  reads this, Repr  
{  
  this in Repr &&  
  g in Repr && g.Valid() &&  
  w in Repr && w.Valid()  
}
```

Class implementation

```
constructor ()
  ensures Valid()
{
  g := new Grinder();
  w := new WaterTank();
  Repr := {this, g, w};
}

predicate method Ready()
  requires Valid()
  reads Repr
{
  g.HasBeans && 2 <= w.Level
}
```

Class implementation

```
method Restock()  
  requires Valid()  
  modifies Repr  
  ensures Valid() && Ready()  
{  
  g.AddBeans(); w.Fill();  
}  
  
method Dispense(double: bool) returns (c: Cup)  
  requires Valid() && Ready()  
  modifies Repr  
  ensures Valid()  
{  
  g.Grind();  
  if double { w.Use(); w.Use(); } else { w.Use(); }  
  c := new Cup();  
}
```

Test harness

```
method CoffeeTestHarness() {  
    var cm := new CoffeeMaker();  
    cm.Restock(); // modifies clause violated  
    var c := cm.Dispense(true); // modifies clause violated  
}
```

The test harness has no **modifies** clause and so is only allowed to modify the fields of fresh objects

Our specification of the coffee maker didn't specify that created objects were fresh

Coffee maker version 2

Add to constructor:

```
ensures fresh(Repr)
```

This removes error with Restock, but not Dispense.

Add to Restock and Dispense:

```
ensures Repr == old(Repr)
```

Alternatively, make Repr *immutable* by declaring it as

```
ghost const Repr: set<object>
```

Changing Repr

What if implementation needs to change Repr, e.g., a method of the coffee maker needs to change the grinder?

Third (and preferred) alternative for `ensures` clauses of methods which mutate Repr:

```
ensures fresh(Repr - old(Repr))
```

That is, any new objects added to Repr are *fresh*

Less common situations

```
method ChangeGrinder()  
  requires Valid()  
  modifies Repr  
  ensures Valid() && fresh(Repr - old(Repr))  
{  
  g := new Grinder();  
  Repr := Repr + {g};  
}
```

Old grinder is still in Repr, but is no longer referenced

The run-time system will eventually reclaim the storage for this object

Less common situations

```
method InstallCustomGrinder(grinder: Grinder)
  requires Valid() && grinder.Valid()
  modifies Repr
  ensures Valid() && fresh(Repr - old(Repr) - {grinder})
{
  g := grinder;
  Repr := Repr + {g};
}
```

Since Repr can dynamically change, this approach to specification is referred to as *dynamic frames*

Dafny is a permutation of certain letters in Dyn^{amic} fra^{mes}

Summary

- Object often can be specified using *ghost fields*
- *Freshness* (aka ownership) needs to be ensured to work with objects without modifies-permissions

Conclusion

What Have We Learned

- State-Transition Systems
- State Machines as Relations
- Safety Properties
- Temporal Formulas
- Fairness and Liveness
- Model Checking with TLA+
- Constraint Programming
- Satisfiability, Validity
- Basics of SMT
- Hoare Logic
- Weakest Preconditions
- Loop Invariants
- Specifications in Dafny
- Proofs about Arrays and Objects

What's Next

- Research Project:
 - Formalising a Distributed Protocol in TLA+
 - Reducing Search Problem to SMT
 - Verifying an Algorithm in Dafny
- For inspiration:
 - Some ideas: <https://ilyasergey.net/PFM24/projects.html>
 - Past projects: <https://ilyasergey.net/CS5232/>
 - I'm happy to discuss project ideas!

And then what?

- Books to read next:
 - *The Hitchhiker’s Guide to Logical Verification* (2023)
 - *Software Foundations* (2008)
 - *Decision Procedures: An Algorithmic Point of View* (2017)
- Conferences to check out: POPL, PLDI, SPLASH, CAV
 - PLMW: <https://www.sigplan.org/Conferences/PLMW/>
 - VMW: <https://i-cav.org/2024/workshops/mentoring/>
- Researchers to follow (very incomplete list):
 - Isil Dillig, Peter Müller, Arie Gurfinkel, Sharon Shoham, Peter O’Hearn, Byron Cook, Leonardo de Moura, Rustan Leino, Kenneth McMillan, Ranjit Jhala, Thomas Wies, Rajeev Alur, Marta Kwiatkowska, Derek Dreyer

Thanks!