

Program verification

Nelma Moreira

Departamento de Ciência de Computadores da FCUP

Program verification
Lecture 9

- Programming language equipped with a static program verifier
- Empowers developers to write provably correct code w.r.t. specifications
- Annotated programs are automatically verified
- Corrected specifications are needed using contracts: pre and post conditions.
- It is also needed to specify invariants, variants, safe conditions, etc..
- includes several compilers for C++, Java, C#, Java, Go, etc.

```
method Abs(x: int) returns (y: int)
  ensures 0 <= y
  ensures 0 <= x ==> x == y
  ensures x < 0 ==> y == -x
{
  if x < 0
    { return -x; }
  else
    { return x; }
}
```

- `requires`: precondition
- `ensures`: postcondition
- `invariant`: invariant
- `decreases`: variant
- Programs are statically verified w.r.t. total correctness: all programs has to provably terminated
- `assert`: a condition that has to hold always
- `reads`: heap memory locations that a function is allowed to read. Corresponds to the *frame* of the function.

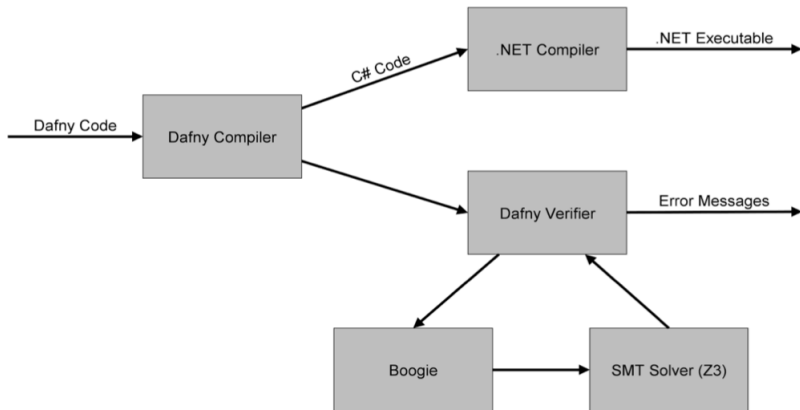
method `M(a: A, b: B, c: C)` returns `(x: X, y: Y, z: Y)`

- method defines a code sequence
- method's correction are verified w.r.t. postconditions
- methods parameters have types
- parameters can be read but not assigned in the method
- and `returns` expresses the type and name of returning variables
- declaration of local variables: `var x:T`

- Programming language
 - Multiple paradigm: combine imperative, functional and object-oriented features
 - allows to write implementations (programs) and specifications (conditions and annotations)
- Programming environment
 - Uses the intermediate language Boogie (that includes the VCGen)
 - Z3 Verifier
 - Compilers for C++, Java, C#, Go, etc
 - Extension for VSCode
 - Extension for Emacs
 - Command line
`alias dafny="mono /path/to/dafny/Dafny.exe"`

- Functional
 - immutable types
 - Pure functions and predicates (without side effects)
 - Typed
- Imperative (data structures and objects)
 - Typed variables
 - immutable and mutable data structures
 - Commands
 - Methods
 - Modules
 - Classes
 - Trait
 - inductive datatypes
 - Iterators
 - ...

Dafny structure



Used in specifications, can define the semantics of a imperative program (do not produce code, except if declared as function method)

```
function fact(n: int): int
  requires 0 <= n
  ensures 1 <= fact(n)
  decreases n
  {if n == 0 then 1 else fact(n-1) * n}
```

If the type is nat the precondition is not necessary.

```
function fib(n: nat): nat
  decreases n
{
  if n == 0 then 0 else
  if n == 1 then 1 else
    fib(n - 1) + fib(n - 2)
}

method ComputeFib(n: nat) returns (b: nat)
  ensures b == fib(n)
{
}
}
```

```
method ComputeFib(n: nat) returns (b: nat)
  ensures b == fib(n)
{
  if n == 0 { return 0; }
  var i: int := 1;
  var a := 0;
      b := 1;
  while i < n
  {
    a, b := b, a + b;
    i := i + 1;
  }
}
```

```
method ComputeFib(n: nat) returns (b: nat)
  ensures b == fib(n)
{
  if n == 0 { return 0; }
  var i: int := 1;
  var a := 0;
    b := 1;
  while i < n
    invariant 0 < i <= n
    invariant a == fib(i - 1)
    invariant b == fib(i)
  {
    a, b := b, a + b;
    i := i + 1;
  }
}
```

See Dafny Cheat Sheet

- immutable types (values)
 - basic types: bool, int , nat, real, char
 - tuples,
 - collections,
 - inductive
- Mutable (references): arrays, classes, etc . Dynamic allocated in the *heap*

<i>bool</i>	! == != && ==> <== <==>
<i>int, nat, real</i>	== != < <== > + - * /
<i>char</i>	== != < <== >== >

Command	Syntax	Example
Declaration	<code>array<T></code>	<code>var a:array<int>= new int[3];</code>
Instance	<code>new T[n]</code>	
Assignment	<code>a[i]:=value</code>	<code>a[1],a[2]:=2,4</code>
Size	<code>a.Length</code>	<code>assert a.Length ==3;</code>
Sequence	<code>a[lo..hi]</code>	<code>assert a[..]=[1,5,6]</code>

method Find(a: array<int>, key: int) returns (i: int)

- a.Length: gives the size of the array
- In conditions we can quantify over the variables
forall k: int :: 0 <= k < a.Length ==> a[k] != key


```
method Find(a: array<int>, key: int)
    returns (index: int)
    ensures 0 <= index ==> index < a.Length
        && a[index] == key
    ensures index < 0 ==>
forall k :: 0 <= k < a.Length ==>
    a[k] != key
{
    index := 0;
    while index < a.Length
        {
            if a[index] == key { return; }
            index := index + 1;
        }
    index := -1;
}
```

Find a value in a array

```
method Find(a: array<int>, key: int)
    returns (index: int)
    ensures 0 <= index ==> index < a.Length
        && a[index] == key
    ensures index < 0 ==>
forall k :: 0 <= k < a.Length ==>
    a[k] != key
{
    index := 0;
    while index < a.Length
        invariant 0 <= index <= a.Length
        invariant forall k :: 0 <= k < index ==> a[k] != key
    {
        if a[index] == key { return; }
        index := index + 1;
    }
    index := -1;
}
```

Maximum of an array

```
method maxarray(arr:array? <int>) returns(max:int)
  requires arr!=null && arr.Length > 0
  ensures 0<= max < arr.Length
  ensures (forall j :int :: (j >= 0 && j < arr.Length
    ==> arr[max] >= arr[j]))
{
  max:=0;
  var i:int :=1;
  while(i < arr.Length)
  invariant (1<=i<=arr.Length)
  invariant 0<= max < i
  invariant (forall j:int :: j>=0 && j<i ==>
    arr[max] >= arr[j])
  decreases (arr.Length-i)
  {
    if(arr[i] > arr[max]){max := i;}
    i := i + 1;
  }
}
```

- Are used to write conditions (pre/post).
- Predicates are just (pure) functions that return a Boolean.

```
predicate sorted(a: array?<int>)
  requires a != null
  reads a
{
  forall j, k :: 0 <= j < k < a.Length ==> a[j] <= a[k]
}
```

```
method BinarySearch(a: array?<int>, value: int) returns (index: int)
  requires a != null && 0 <= a.Length && sorted(a)
  ensures 0 <= index ==> index < a.Length && a[index] == value
  ensures index < 0 ==> forall k :: 0 <= k < a.Length ==> a[k] != value
{
  var low, high := 0, a.Length;
  while low < high
    invariant 0 <= low <= high <= a.Length
    invariant forall i ::
      0 <= i < a.Length && !(low <= i < high) ==> a[i] != value
  {
    var mid := (low + high) / 2;
    if a[mid] < value
    {
      low := mid + 1;
    }
    else if value < a[mid]
    {
      high := mid;
    }
    else
    {
      return mid;
    }
  }
}
return -1;
}
```

- for parameters passed by reference (allocated in the heap): arrays, object (not local variables nor collections, tuples, sets, etc)
- functions and predicates cannot have side-effects and one needs to state which memory positions (mutable) can be read and if other positions are modified those values should not change: `reads`
- methods do not need to state what they read but need to tell which variables they can modify: `modifies`
- functions are *transparent* and methods are *opaque*: functions can be used anywhere.

```
type T = int
predicate isSorted(a: array<T>)
  reads a
{
  forall i, j :: 0 <= i < j < a.Length ==> a[i] <= a[j]
}

method bubbleSort(a: array<T>)
  modifies a
  ensures isSorted(a)
  ensures multiset(a[..]) == multiset(old(a[..]))
```

```
method bubbleSort(a: array<T>)
{
    var i := a.Length;
    while i > 1
    {
        var j := 0; // used to scan left subarray
        while j < i - 1
        {
            if (a[j] > a[j+1])
            {
                a[j], a[j+1] := a[j+1], a[j];
            }
            j := j+1;
        }
        i := i-1;
    }
}
```


7, 2, 6, 3, 4

$i = 5 \rightarrow 2, 6, 3, 4, 7$

$i = 4 \rightarrow 2, 3, 4, 6, 7$

$i = 3 \rightarrow 2, 3, 4, 6, 7$

$i = 2 \rightarrow 2, 3, 4, 6, 7$

$i = 1 \rightarrow 2, 3, 4, 6, 7$

Invariant: for indexes $\geq i$ the values are larger than any other and ordered. Variant: i

```

var i := a.Length; // len of left subarray to sort
while i > 1
  decreases i
  invariant 0 <= i <= a.Length
  invariant forall l, r :: 0 <= l < r < a.Length && r >= i
    ==> a[l] <= a[r]
  invariant multiset(a[..]) == multiset(old(a[..]))

```

the first invariant can be omitted (because of the variant)

Invariant: for indexes $\leq j$ are less or equal to the value of j and the outer invariant

Variant: $i - j$

```

var j := 0;
while j < i - 1
  decreases i - j
  invariant 0 <= j <= i-1
  invariant forall l, r :: 0 <= l < r < a.Length &&
    (r >= i || r == j) ==> a[l] <= a[r]
  invariant multiset(a[..]) == multiset(old(a[..]))
  {
    if (a[j] > a[j+1])
    {
      a[j], a[j+1] := a[j+1], a[j];
    }
    j := j+1;

    i := i-1;
  }

```

```
method quicksort(a: array<int>)
{
    quicksort2(a, 0, a.Length-1);
}
method quicksort2(a: array<int>, lo: int, hi: int){
{
    if lo < hi
    {
        var pivot := partition(a, lo, hi);
        quicksort2(a, lo, pivot - 1);
        quicksort2(a, pivot + 1, hi);
    }
}
```

```
method partition(a: array<int>, lo: int, hi: int)
    returns(pivot: int) {
    var i := lo;
    var j := lo;
    pivot := hi;
    while j < hi
    {
        if a[j] < a[hi]
        {
            a[i], a[j] := a[j], a[i];
            i := i + 1;
        }
        j := j+1;
    }
    a[hi], a[i] := a[i], a[hi];
    pivot := i;
    return pivot;}
```

$lo = 0, hi = 4$

7, 2, 6, 3, 4

$i = j = 0 \rightarrow 7, 2, 6, 3, 4$
 $i = 0, j = 1 \rightarrow 7, 2, 6, 3, 4$
 $i = 1, j = 2 \rightarrow 2, 7, 6, 3, 4$
 $i = 1, j = 3 \rightarrow 2, 7, 6, 3, 4$
 $i = 2, j = 4 \rightarrow 2, 3, 6, 7, 4$
 $i = 2, j = 5 \rightarrow 2, 3, 6, 7, 4$
 $i = 2, j = 5 \rightarrow 2, 3, 4, 7, 6$

 $pivot = 2$

```
predicate sorted(a: array<int>, lo: int, hi: int)
reads a
{
    forall i, j :: 0 <= lo <= i < j <= hi < a.Length ==>
        a[i] <= a[j]
}
// Checks if values of array 'a' in the range
[0 .. lo-1] <= [lo..hi] <= [hi+1..a.Length-1]
predicate partitioned(a: array<int>, lo: int, hi: int)
reads a
{
    (forall i, j :: 0 <= i < lo <= j <= hi < a.Length
        ==> a[i] <= a[j])
    && (forall i, j :: 0 <= lo <= i <= hi < j < a.Length
        ==> a[i] <= a[j])
}
```

- quicksort2:
requires $0 \leq lo \leq hi + 1 \leq a.Length$
- partition:
requires $0 \leq lo \leq hi < a.Length$
requires `partitioned(a, lo, hi)`

- quicksort:
 - ensures sorted(a, 0, a.Length-1)
 - ensures multiset(a[..]) == multiset(old(a)[..])
- quicksort2:
 - ensures sorted(a, lo, hi)
 - ensures multiset(a[lo..hi+1]) == multiset(old(a)[lo..hi+1])
 - ensures forall k :: 0 <= k < lo || hi < k < a.Length
 - ==> a[k] == old(a[k])
 - ensures partitioned(a, lo, hi)
- partition: (p is pivot)
 - ensures lo <= pivot <= hi
 - ensures forall k :: lo <= k < p ==> a[k] <= a[p]
 - ensures forall k :: p < k <= hi ==> a[k] >= a[p]
 - ensures multiset(a[lo..hi+1]) == multiset(old(a)[lo..hi+1])
 - ensures forall k :: 0 <= k < lo || hi < k < a.Length ==>
 - a[k] == old(a[k])
 - ensures partitioned(a, lo, hi)

For the while of partition. Note that `pivot==hi`.

```
invariant lo <= i <= j <= hi
invariant forall k :: lo <= k < i ==> a[k] < a[pivot]
invariant forall k :: i <= k < j ==> a[k] >= a[pivot]
invariant multiset(a[lo..hi+1]) ==
           multiset(old(a)[lo..hi+1])
invariant forall k :: 0 <= k < lo || hi < k < a.Length
           ==> a[k] == old(a[k])
invariant partitioned(a, lo, hi)
```