Program verification

Nelma Moreira

Departamento de Ciência de Computadores da FCUP

Program verificaction
Verifying programs with Dafny
Lecture 5

Dafny (https://dafny.org)

- Programming language equipped with a static program verifier
- Empowers developers to write provably correct code w.r.t. specifications
- Annoted 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++, Python, 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 termined
- `assert`: a condition that has to hold always
- `assume`: a condition tthat we assume that holds
- `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`

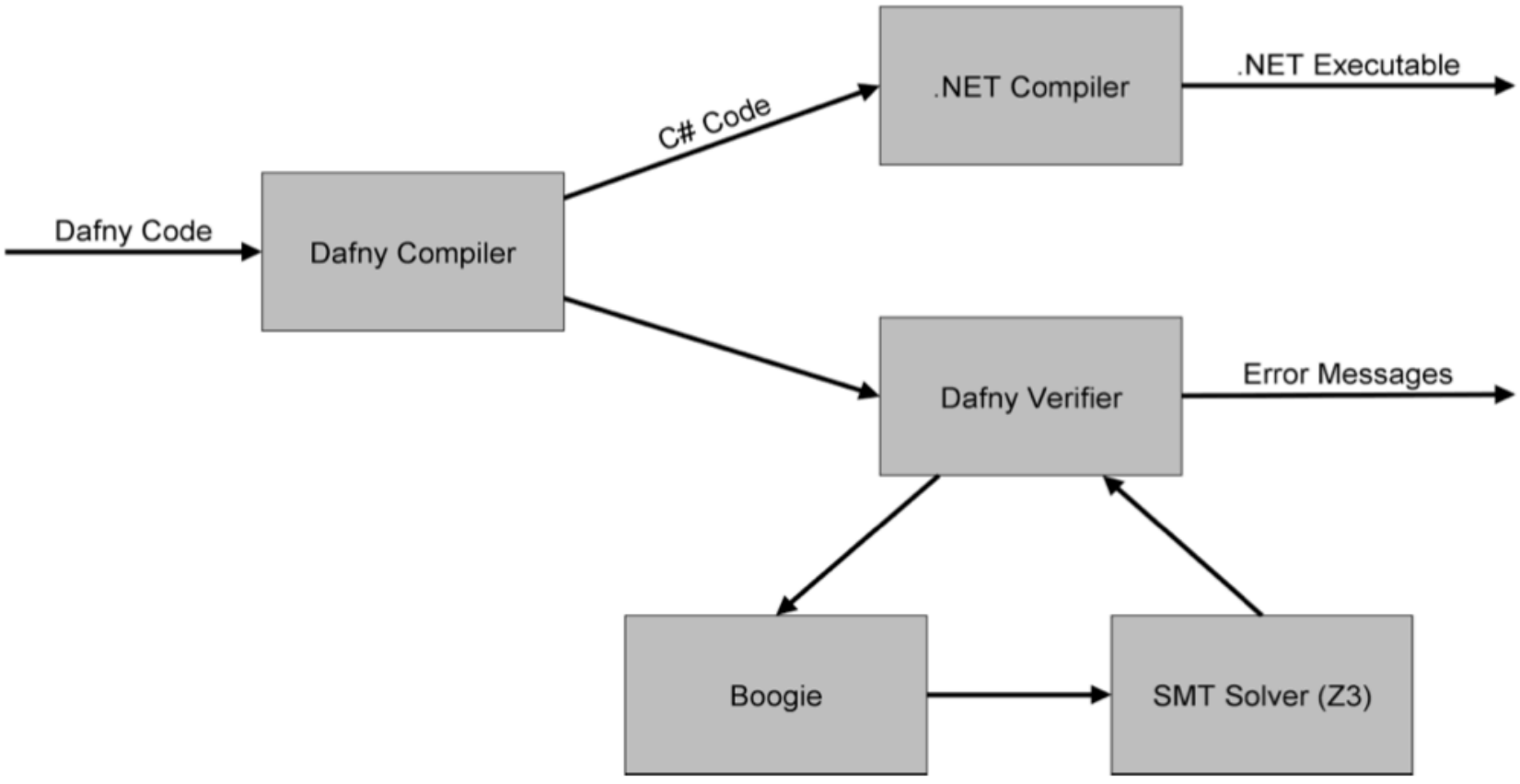
Dafny

- 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
  - Requires `dotnet`
  - Install: Linux/Mac/Windows
  - Extension for VSCode:
    `ext install dafny-lang.ide-vscode`
  - Extension for Emacs
    `(setq flycheck-dafny-executable "[path]/dafny/Scripts/da`
  - Command line
    `alias dafny="mono /path/to/dafny/Dafny.exe"`

Programming paradigms

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

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 fact(n: nat) : nat
  decreases n
{
    if n == 0 then 1 else  fact(n-1) * n
}

method factIter(n: nat) returns (f : nat) // Iterative calc
  ensures f == fact(n)
{
  f := 1;
  var i := 0;
  while i < n
    decreases n - i
    invariant 0 <= i <= n && f == fact(i)
  {
    i := i + 1;
    f := f * i;
  }
  return f;
}
```

Fibonacci Sequence

```
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
        invariant 0 < i <= n
        invariant a == fib(i - 1)
        invariant b == fib(i)
    {
        a , b := ?
        i := i + 1;
    }
}
```

Using Hoare logic one can "built" the program!

The invariant must hold at the beginning and at the end of the loop and using the *wp* for the assignment one can conclude that after incrementing i we should have $a = fib(i) \wedge b = fib(i+1)$. So we can conclude that

```
a:= b
b:= a+b
```

for the invariant to hold! Verify building the *tableaux*!

```
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*

Basic operators

bool	$!,\ ==\ .\ !=, \&\&.\ \ ==>,\ <==,\ <==>,$
int, *nat*, *real*	$==,\ !=,\ <,\ <=,\ =>,\ +,\ -,\ *,\ /,$
char	$==,\ !=,\ <,\ <=,\ >=,\ >,$

Collections

Description	Declaration	Examples
Sets (Without repetions)	`set<T>`	`var s:set<int>:={1,2,3}`
Sequences (Lists)	`seq<T>`	`var s:seq<int>:=[1,2,3,3]`
Multiset (With Repetions)	`multiset<T>`	`var s:multiset<int>:=` `multiset{2,3,3}`
String	`string`	`"hello world\n"`
Map (Dictionary)	`map<K,V>`	`map<string, int>:=` `map["one":=1,"two":= 2]`

Set operations

operator	description
<	proper subset
<=	subset
>=	superset
>	proper superset

operator	description
!!	disjointness
+	set union
-	set difference
*	set intersection

expression	description
|s|	set cardinality
e in s	set membership
e !in s	set non-membership

multiset(s): set conversion to multiset<T>

Sets defined by comprehension: values $Q(x_1, \ldots, x_n)$ that satisfy $P(x_1, \ldots, x_n)$:

$$\mathit{var}\ S \quad := \quad x_1 : T_1, \ldots, x_n\, T_n \ldots \mid P(x_1, \ldots, x_n) :: Q(x_1, \ldots, x_n)$$

Example:

```
var S := set x:nat, y:nat | x < 2 && y < 2 :: (x, y)
```

gets

$$S = \{(0,0), (0,1), (1,0), (1,1)\}$$

Sequence operators

operator	description
<	proper prefix
<=	prefix

operator	description
+	concatenation

expression	description
`|s|`	sequence length
`s[i]`	sequence selection `0 <= i < |s|`
`s[i := e]`	sequence update
`e in s`	sequence membership
`e !in s`	sequence non-membership
`s[lo..hi]`	subsequence `0 <= lo <= hi <= |s|`
`s[lo..]`	drop
`s[..hi]`	take
`s[slices]`	slice
`multiset(s)`	sequence conversion to a `multiset<T>`

lo,*hi* are sizes not positions: *s*[*lo*..] not the first *lo* elements;*s*[..*hi*] first *hi* elements;

Slices

$s[i : j : k]$ with $i, j, k >= 0$ and $i + j + k < |s|$, slices s in three subsequences of lengths i, j and k ignoring the rest

```
var t := [3.14, 2.7, 1.41, 1985.44, 100.0, 37.2][1:0:3];
assert |t| == 3 && t[0] == [3.14] && t[1] == [];
assert t[2] == [2.7, 1.41, 1985.44];
```

Multiset operations

operator	description
`<`	proper multiset subset
`<=`	multiset subset
`>=`	multiset superset
`>`	proper multiset superset

operator	description
`!!`	multiset disjointness
`+`	multiset union
`-`	multiset difference
`*`	multiset intersection

expression	description
`|s|`	multiset cardinality
`e in s`	multiset membership
`e !in s`	multiset non-membership
`s[e]`	multiplicity of `e` in `s`
`s[e := n]`	multiset update (change of multiplicity)

Map (Dictionary) operators

The comprehension maps are defined as sets are.

$$map\ x : int \mid 0 <= x <= 10 :: x * x$$

expression	description
`|fm|`	map cardinality
`m[d]`	map selection
`m[t := u]`	map update
`t in m`	map domain membership
`t !in m`	map domain non-membership

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

Arrays

```
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
```

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
        {
      if a[index] == key { return; }
      index := index + 1;
   }
   index := -1;
}
```

Find a value in a array

One needs to provide the invariant

```
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;
  }
}
```

Predicates

- 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]
}
```

Binary search

```
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

   {
      var mid := (low + high) / 2;
      if a[mid] < value
      {
         low := mid + 1;
      }
      else if value < a[mid]
      {
         high := mid;
      }
      else
      {
         return mid;
      }
   }
   return -1;
}
```

Binary search

```
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;
}
```

Framing, reads and modifies

- 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.
- If a variable is declared `ghost` in a method, it will not be compiled
- A method or a function can also be `ghost`

Bubble sort

```
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[..]))
```

Bubble sort

```
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;
    }
}
```

$$7, 2, 6, 3, 4$$

$$\begin{aligned}
i = 5 &\rightarrow 2, 6, 3, 4, 7 \\
i = 4 &\rightarrow 2, 3, 4, 6, 7 \\
i = 3 &\rightarrow 2, 3, 4, 6, 7 \\
i = 2 &\rightarrow 2, 3, 4, 6, 7 \\
i = 1 &\rightarrow 2, 3, 4, 6, 7
\end{aligned}$$

Bubble sort - outer loop

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 omited (because of the variant)

Bubble sort - inner loop

Invariant: for indexes $\leq j$ the values 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;}
```

Execution of partition

$lo = 0,\ hi = 4$

$$7, 2, 6, 3, 4$$

$$\begin{aligned}
i = j = 0 &\rightarrow 7, 2, 6, 3, 4 \\
i = 0, j = 1 &\rightarrow 7, 2, 6, 3, 4 \\
i = 1, j = 2 &\rightarrow 2, 7, 6, 3, 4 \\
i = 1, j = 3 &\rightarrow 2, 7, 6, 3, 4 \\
i = 2, j = 4 &\rightarrow 2, 3, 6, 7, 4 \\
i = 2, j = 5 &\rightarrow 2, 3, 6, 7, 4 \\
i = 2, j = 5 &\rightarrow 2, 3, 4, 7, 6
\end{aligned}$$

$pivot = 2$

```
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])
}
```

Preconditions

- quicksort2:

  ```
  requires 0 <= lo <= hi + 1 <= a.Length
  ```

- partition:

  ```
  requires 0 <= lo <= hi < a.Length
  requires partitioned(a, lo, hi)
  ```

Postconditions

- 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)
```

Invariant

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)
```

Sum of an array

```
function Sum(v: seq<int>) :int
decreases v
{ if v == [] then 0 else v[0] + Sum(v[1..]) }

method sum(v:array<int>) returns (x: int)
 ensures x == Sum(v[..])
  {
    var n:= v.Length;
    x:= 0;
    while n != 0
        invariant 0 <= n <= v.Length
        invariant x == Sum(v[n..])
     {
        x, n := x + v[n-1], n - 1;
         }
     }
```

Sum of an array (2)

But if

```
function Sum(v: seq<int>) :int
decreases v
{ if v == [] then 0 else v[0] + Sum(v[1..]) }

method sum1(v:array<int>) returns (x: int)
 ensures x == Sum(v[..])
  {
    var n:= 0;
    x:= 0;
    while n != v.Length
        invariant 0 <= n <= v.Length
        invariant x == Sum(v[..n])
     {
        x, n := x + v[n], n + 1;
         }
     }
```

Lemmas

```
lemma Ex_Lemma (x1: T1,...,xn: Tn)
requires phi
ensures psi
{ body }
```

It means $\forall x_1, \ldots, x_n(\phi \rightarrow \psi)$ and the body is the proof.

A call to `lemma(a)` corresponds to variables instantiation.

In general a `lemma` is like a `method` but is not executed in runtime.

Lemmas are ghost methods. They allow proofs by induction.

Sum of an array (2)

But if

```
function Sum(v:seq<int>) :int
decreases v
{ if v == [] then 0 else v[0] + Sum(v[1..]) }

method sum1(v:array<int>) returns (x: int)
 ensures x == Sum(v[..])
  {
    var n:= 0;
    x:= 0;
    while n != v.Length
        invariant 0 <= n <= v.Length
        invariant x == Sum(v[..n])
     {
        x, n := x + v[n], n + 1;
         }
     }
```

```
lemma  SumLemma(s:seq<int>, i:int)
    requires 0 <= i < |s|
    ensures Sum(s[..i]) + s[i] == Sum(s[..i+1])
{
    if i > 0
    { SumLemma(s[1..],i-1);
    assert Sum(s[1..][..i-1]) + s[1..][i-1]
              == Sum(s[1..][..i]);
    assert s[1..][..i-1] == s[1..i];
    assert s[1..][..i] == s[1..i+1];
    assert Sum(s[1..i]) + s[i] == Sum(s[1..i+1]);
    assert Sum(s[..i]) + s[i] == Sum(s[..i+1]);
    }
 }
```

```
function Sum(v:seq<int>) :int
decreases v
{ if v == [] then 0 else v[0] + Sum(v[1..]) }

method sum1(v:array<int>) returns (x: int)
 ensures x == Sum(v[..])
  {
    var n:= 0;
    x:= 0;
    while n != v.Length
        invariant 0 <= n <= v.Length
        invariant x == Sum(v[..n])
     {
        x, n := x + v[n], n + 1;
        SumLemma(v[..], n-1);
         }
     assert v[..v.Length] == v[..];
     }
```

Proving a number theory theorem:

Theorem
$\forall k > 0, 2^{3k} - 3^k$ *is divisible by* 5.

The theorem can be proven by induction on k. We can simulate that with a program and ensure that the program validates the theorem.

The proof although using assertions is similar to the one that can be done with an interactive theorem prover.

$2^{3k} - 3^k$ is divisible by 5

```
function f(k: int): int
requires k >= 1;
{ (exp(2,3*k) - exp(3,k)) / 5 }

function exp(x: int ,e: int): int
 requires e >= 0
 decreases e
{ if e==0 then 1 else x * exp(x,e - 1) }
```

$2^{3k} - 3^k$ is divisible by 5

```
method compute5f (k: int) returns (r: int)
requires k >= 1
ensures r == 5*f(k)
{
var i, t1, t2:= 0, 1, 1;
while i < k
decreases k-i
invariant 0 <= i <= k;
invariant t1 == exp(2,3*i);
invariant t2 == exp(3,i);
{
i, t1, t2 := i+1, 8*t1, 3*t2;
}
r := t1 - t2;
}
```

We use the directive `assume`.

```
assume t1 ==  exp(2,3*i)
```

and assert the post condition

```
assert r == exp(2,3*k) - exp(3,k);
```

To verify the program one needs to change `assume` to `assert`.

But Dafny cannot prove the assertion.

Solution: try to construct a tableaux using the weakest precondition technique

```
assume 8*t1 == exp(2,3*(i+1));
i, t1, t2 := i+1, 8*t1, 3*t2;
assert t1 == exp(2,3*i);
```

But the assert still does not hold

```
assert 8*t1 == 8*exp(2,3*i)== exp(2,3*(i+1))==exp(2,3*i+3)
```

Some lemmas are needed and then we use them instead of asserts.

```
lemma expPlus3_Lemma (x: int , e: int )
requires e >= 0;
ensures x * x * x * exp(x,e) == exp(x,e+3)
// to be proved

lemma DivBy5_Lemma (k: int )
requires k >= 1
ensures (exp(2,3*k) - exp(3,k)) % 5 == 0
// to be proved
```

For the first lemma we just use an iterative computation

```
lemma expPlus3_Lemma (x: int , e: int )
requires e >= 0;
ensures x * x * x * exp(x,e) == exp(x,e+3);
{
    assert x * x * x * exp(x,e) == x * x * exp(x,e+1) == x
   // assert x* exp(x,e) == exp(x,e+1);
}
```

For the second one it is needed to use `calc` that allows to perform algebraic calculations where one can use also `assert, lemma`, etc to justify each step (*hints*, that are written with $\{\}$).

Each step is separated by a logic operator : $==$, $\rightarrow$, etc

```
Lemma {:induction k} DivBy5_Lemma (k: int )
requires k >= 1
decreases k
ensures (exp(2,3*k) - exp(3,k)) % 5 == 0
{
if k==1 {
} else {
calc {
(exp(2,3*k)- exp(3,k)) % 5;
==
(exp(2,3*(k-1)+3) - exp(3,(k-1)+1)) % 5;
==
expPlus3_Lemma(2,3*(k-1));
}
(8*exp(2,3*(k-1)) - exp(3,(k-1))*3) % 5;
==
 (3 *( exp(2,3*(k-1)) - exp(3,k-1) ) + 5*exp(2,3*(k-1))) % 5;
  ==
 { DivBy5_Lemma (k-1);
 }
// assert(exp(2,3*(k-1))- exp(3,k-1)) % 5 == 0;
0;}
}
}
```

The previous lemma can be simplified just stating what is needed to the proof.

```
lemma DivBy5LemmaS (k: int )
requires k >= 1
ensures (exp(2,3*k) - exp(3,k)) % 5 == 0
{ if k > 1
{expPlus3_Lemma(2,3*(k-1));
     DivBy5LemmaS(k-1);
}
}
```

And the general form is

```
lemma Lemma1 ()
ensures forall n:: n >= 1 ==> (exp(2,3*n) - exp(3,n)) % 5 == 0
{
forall n | n >= 1 { DivBy5_Lemma(n); }
}
```

Then the annotated program is

```
method compute5f (k: int) returns (r: int)
requires k >= 1
ensures r == 5*f(k)
{
var i, t1, t2:= 0, 1, 1;
while i < k
decreases k-i;
invariant 0 <= i <= k;
invariant t1 == exp(2,3*i);
invariant t2 == exp(3,i);
{
    expPlus3_Lemma(2,3*i) ;
   // assert 8*t1 == 8*exp(2,3*i)== exp(2,3*(i+1))==exp(2,3*i+3);
    i, t1, t2 := i+1, 8*t1, 3*t2;
    assert t1 == exp(2,3*i);
}
r := t1 - t2;
DivBy5Lemma(k);
//assert (exp(2,3*k) - exp(3,k))%5 == 0;
//assert r == 5 * f(k);
}
}
```

```
predicate OddPre(n: nat) {
    n%2 == 1
}

function Odd(n:nat): bool
decreases n
{ if n == 1 then true
else if  n == 0 then false
else !Odd(n-1)
}
```

```
function OddTail(n:nat,o:bool): bool
decreases n
{ if n == 1 then o
else if  n == 0 then ! o
else OddTail(n-1, !o )
}

lemma {:induction} TestOdd(n: nat)
decreases n
ensures Odd(n) == OddPre(n)
{}
```

Multiplication

```
function Mult(x: nat, y: nat): nat
    decreases x,y
    {
  if y == 0 then 0 else x + Mult(x, y - 1)
}
```

We use lexicographic order for pairs (in the variant)

Proof that

$$Mult(x, y) = Mult(y, x)$$

i.e.

```
Lemma MultCommutative(x: nat, y: nat)
  ensures Mult(x, y) == Mult(y, x)
  {}
```

```
Lemma {:induction false} MultCommutative(x: nat, y: nat)
  ensures Mult(x, y) == Mult(y, x)
  { if x==y {}
  else if x == 0 {
    MultCommutative(x, y-1);}
  else if (y < x) { MultCommutative(y, x);}
  else {
    calc {
    Mult(x,y);
    == x + Mult (x, y-1);
    == {MultCommutative(x, y-1);}
    x + Mult (y-1,x);
    ==
    x + y-1 + Mult(y-1,x-1);
    == {MultCommutative(x-1, y-1);}
     x + y-1 + Mult(x-1,y-1);
     ==
     y + Mult(x-1,y);
     == {MultCommutative(x-1, y);}
     y + Mult(y,x-1);
     ==
     Mult(y, x);
         }}
 }
```