

Collections

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

Set operations

operator	description	operator	description	expression	description
<code><</code>	proper subset	<code>!!</code>	disjointness	<code> s </code>	set cardinality
<code><=</code>	subset	<code>+</code>	set union	<code>e in s</code>	set membership
<code>>=</code>	superset	<code>-</code>	set difference	<code>e !in s</code>	set non-membership
<code>></code>	proper superset	<code>*</code>	set intersection	multiset(s): set conversion to multiset<T>	

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

$$var S := x_1 : T_1, \dots, x_n T_n \dots \mid P(x_1, \dots, x_n) :: Q(x_1, \dots, x_n)$$

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

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

Slices: $s[i : j : k]$ with $i, j, k \geq 0$ and $i + j + k \leq |s|$, 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

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

The comprehension maps are defined as sets are.

$$\text{map } x : \text{int} \mid 0 \leq x \leq 10 :: x * x$$

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 $\forall x_1, \dots, x_n (\phi \rightarrow \psi)$ 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 theorems with Dafny

Proving a number theory theorem:

Theorem 1. $\forall k > 0, 2^{3k} - 3^k$ 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;
}

```

Assume and Assert

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);
```

Lemmas

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
```

`expPlus3Lemma`

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 * exp(x,e+2) == exp(x,e+3);
    // assert x* exp(x,e) == exp(x,e+1);
}
```

DivBy5Lemma

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 : `==`, `→`, 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;
  }
}

```

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;
DivBy5Lemmma(k);
//assert (exp(2,3*k) - exp(3,k))%5 == 0;
//assert r == 5 * f(k);
}
}

```

More on Lemmas and Induction

```

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);
    }}
}

```