

# INTRODUCTION TO DAFNY

---

Lin Tzu-Chi

October 7, 2020

Dafny is an imperative programming language with built-in annotations to prove correctness of code.

# BASIC SYNTAX

---

methods are functions in typical imperative languages.

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

The input parameters are read only, and an implicit **return** is added automatically at the end of a method, where the current values of return parameters are returned as-is.

There can be multiple return values.

```
method MultipleReturns(x: int, y: int)
  returns (more: int, less: int)
{
  more := x + y;
  less := x - y;
  // comments.
}
```

**ensures** annotates postconditions of a method for Dafny to check its correctness.

```
method MultipleReturns(x: int, y: int)
  returns (more: int, less: int)
    ensures less < x
    ensures x < more
{
  more := x + y;
  less := x - y;
}
```

Dafny rejects this program.

**requires** annotates preconditions. It is the programmer's job to establish them.

```
method MultipleReturns(x: int, y: int)
  returns (more: int, less: int)
    requires 0 < y
    ensures less < x < more
{
  more := x + y;
  less := x - y;
}
```

Dafny verifies this program successfully.

`assert` is a keyword to place assertions in the middle of a method.

```
// use definition of Abs() from before.  
method Testing()  
{  
    var v := Abs(3);  
    assert 0 <= v;  
}
```



The program:

```
var v := Abs(3);  
assert v == 3;
```

would not be verified, because Dafny only knows the postconditions of **Abs**, but nothing more.

To prove the assertion above, we can modify **Abs** to provide more information.

```
method Abs(x: int) returns (y: int)
  ensures 0 <= y
  ensures 0 <= x ==> x == y
{
  // body as before
}
```

```
function abs(x: int): int
{
    if x < 0 then -x else x
}
```

Unlike a method, which can have all sorts of statements in its body, a function body must consist of exactly one expression, with the correct return type.

The power of functions comes from the fact that they can be used directly in specifications. So we can write:

```
assert abs(3) == 3;
```

without explicitly writing pre- and postconditions.

# LOOP INVARIANTS

```
var i := 0;
while i < n
  invariant 0 <= i
{
  i := i + 1;
}
```

When you specify an invariant, Dafny proves two things: the invariant holds upon entering the loop, and it is preserved by the loop. Assuming the invariant holds at the beginning of the loop, we must show that executing the loop body once makes the invariant hold again.

# LOOP INVARIANTS

```
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;
  }
}
```

A decreases annotation, as its name suggests, gives Dafny and expression that decreases with every loop iteration or recursive call.

```
while 0 < i
  invariant 0 <= i
  decreases i
{
  i := i - 1;
}
```

`array<T>` is the type of array of `T`. Arrays can be `null`, and have a built in length field, `a.Length`. All array accesses must be proven to be within bounds. This program can be successfully verified:

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



# QUANTIFIERS

```
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;
}
```

A predicate is a function which returns a boolean.

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

Note that there is no return type, because predicates always return a boolean. Dafny rejects this code as given, claiming that the predicate cannot read `a`. Fixing this issue requires another annotation, the `reads` annotation.

The sorted predicate is not able to access the array because the array was not included in the function's reading frame. The reading frame of a function (or predicate) is all the memory locations that the function is allowed to read. Dafny will check that you do not read any memory location that is not stated in the reading frame.

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

# BINARY SEARCH

```
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;
}
```

# TERMINATION

---

There are two potential sources of non-terminating (divergent) behavior: loops, and recursive functions and methods. Dafny employs **decreases** annotations for handling either case.

A decreases annotation specifies a value, called the termination measure, that becomes strictly smaller each time, and it is bounded.

Each kind of termination measure comes with a built-in lower bound, Dafny proves that the termination measure gets smaller on each iteration.

```
while i < n
  invariant 0 <= i <= n
{
  // do something interesting
  i := i + 1;
}
```

Dafny verifies this program immediately. Because it is a common loop form and Dafny tries to guess the termination measure **decreases**  $n - i$ .

Dafny requires that the loop does not execute again when the termination measure is negative.

Dafny proves the termination of the whole program, not just loops. For each function/method that is possibly recursive, it requires either an explicit or implicit decreases annotation on it.

Most recursive functions just call themselves with smaller values of the parameters, so the parameters decreasing is the default guess:

```
function fac(n: nat): nat
{
  if n == 0 then 1 else n * fac(n-1)
}
```



Dafny provides a special annotation that instructs Dafny not to attempt to prove termination:

```
method hail(N: nat)
  decreases *
{
  var n := N;
  while 1 < n
    decreases *
    {
      n := if n % 2 == 0 then n / 2 else n * 3 + 1;
    }
}
```

The final termination measure is a tuple of the other kinds of measures. For example, the following implementation of the Ackermann function uses a pair of integers to prove termination:

```
function Ack(m: nat, n: nat): nat
  decreases m, n
{
  if m == 0 then n + 1
  else if n == 0 then Ack(m - 1, 1)
  else Ack(m - 1, Ack(m, n - 1))
}
```

Eventually one of the elements must decrease.

Termination applies not just to single functions/methods, but also to multiple mutually recursive functions/methods:

```
predicate even(n: nat)
  ensures even(n) <==> n % 2 == 0
{
  if n == 0 then true else odd(n-1)
}
predicate odd(n: nat)
  ensures odd(n) <==> n % 2 != 0
{
  if n == 0 then false else even(n-1)
}
```

# LEMMAS

---

Sometimes there are steps of logic required to prove a program correct, but they are too complex for Dafny to discover and use on its own:

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

A `lemma` is really just a ghost method. The desired property stated by the lemma is declared as the postcondition:

```
lemma Lemma(...)
  ensures (desirable property)
{
  ...
}
```

For the zero search problem:

```
lemma SkippingLemma(a : array<int>, j : int)
  requires a != null
  requires forall i :: 0 <= i < a.Length ==> 0 <= a[i]
  requires forall i :: 0 < i < a.Length ==> a[i-1]-1 <= a[i]
  requires 0 <= j < a.Length
  ensures forall i :: j <= i < j + a[j] && i < a.Length ==> a[i] != 0
{
  ...
}
```

Checking that this lemma is what we want:

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



Finishing the lemma:

```
lemma SkippingLemma(a : array<int>, j : int)
  requires a != null
  requires forall i :: 0 <= i < a.Length ==> 0 <= a[i]
  requires forall i :: 0 < i < a.Length ==> a[i-1]-1 <= a[i]
  requires 0 <= j < a.Length
  ensures forall i :: j <= i < j + a[j] && i < a.Length ==> a[i] != 0
{
  var i := j;
  while i < j + a[j] && i < a.Length
    invariant i < a.Length ==> a[j] - (i-j) <= a[i]
    invariant forall k :: j <= k < i && k < a.Length ==> a[k] != 0
  {
    i := i + 1;
  }
}
```