# INTRODUCTION TO DAFNY

Lin Tzu-Chi

November 29, 2021

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

- Dafny is built on Boogie, a intermediate verification language.
- Boogie generates verification conditions (logical formulas), which are passed to an SMT solver (Z3 by default).

# BASIC SYNTAX

`method`s 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.

We can check by ourselves that postcondition above holds under given precondition. But how does Dafny know that?

Pre- and Postconditions are translated to the following formula, ($\rightarrow$ represents implication.)

$$\begin{aligned}(0 < y) \rightarrow ((\text{more} = x + y) \wedge (\text{more} > x)) \\ \wedge((\text{less} = x - y) \wedge (\text{less} < x))\end{aligned}$$

We can check by hand that the annotated pre- and postconditions hold if and only if this formula always holds.

While an SMT solver cannot directly prove that a proposition always holds, the problem can be translated to verifying the satisfiability of **negation** of the given formula,

such that if it is unsatisfiable, there is no program state that could violates given pre- and postconditions:

$$\neg((0 < y) \rightarrow ((x + y > x) \wedge (x - y < x)))$$

`assert` is a keyword to place assertions in the moddle 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

To pre- and postconditions of a loop, We would want to konw what always holds for each loop iteration.

```
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, Dafny checks that executing the loop body once makes the invariant hold again:

$$(0 \leq i_0) \rightarrow (i_1 = i_0 + 1) \wedge (0 \leq i_1)$$

, as mentioned, to verify above formula, the problem is equivalent to unsatisfiability of the negation of the formula.

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

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

Combining with the requirement that it must be an integer, the measure will reach zero in finite number of iterations, thus ensures termination.

The `decreases` in this case would be translated to:

$$(i_1 = i_0 - 1) \land (i_1 < i_0)$$

It is crucial that strict decrementation of the measure does entail termination (the measure has not reached zero yet, or the loop terminates, i.e. the while condition does not hold):

$$(0 < i_1) \lor \neg(0 < i_1)$$

A `decreases` is automatically added if not given by the user, so an addition formula is added to verification condition whenerver a loop is present.

Excercise: Write down the `fib` function.

```
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;
   }
}
```

Hint: it should be defined recursively.

- `array<T>` is the type of array of type `T`.
- `array<T>` is always non-empty, while `array?<T>` can be `null`.
- An array have a built-in length field, `a.Length`.
- All array accesses must be proven to be within bounds.

This program can be successfully verified, but what does it mean?

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

# QUANTIFIERS

A `forall` quantifier can be added in a proposition.

```
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;
}
```

Excercise: write down the verification condition(s) of the program above.

# PREDICATES

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 the `reads` annotation.

# FRAMING

- 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.

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

# EXAMPLE: 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;
}
```