The Use and Anatomy of an Auto-Active Verifier

Rustan Leino

Lecture 2 - *July 1, 2026*

1 More Hands-On in Dafny

One can run Dafny programs in the terminal with the following command: `dafny run Hello.dfy`.

To better understand how to verify programs in Dafny, we will implement another sorting program with the same specification as the previous one but using a different algorithm.

The program `BruteSort` first counts the number of `false` and `true` values in the array, and then rewrites the array based on these counts. Although this algorithm is not efficient in practice and would not be useful if the booleans were tied to some payload, it is simple and it is a good example for understanding the verification process.

An implementation without the loop invariants is shown in Listing 1. However, this version does not verify because Dafny lacks several pieces of information that are necessary for the proof.

Listing 1: Initial implementation of `BruteSort`

```
method BruteSort(a: array<bool>)
    modifies a
    ensures forall i, j :: 0 <= i < j < a.Length ==> a[i] ==> a[j]
    ensures multiset(a[..]) == multiset(old(a[..]))
{
    // number of false's and true's, respectively
    var c0 := 0;
    var c1 := 0;
    var m := multiset{};
    var n := 0;
    while n < a.Length
    {
        var x := a[n];
        if x {
```

Compiled By:
Namratha Gangamreddypalli, Jasper Geer, Max Gross, Abigail Pribisova,
Hans Sardar, and Konami Shu

```
            c1 := c1 + 1;
        } else {
            c0 := c0 + 1;
        }
        m := m + multiset{x};
        n := n + 1;
    }
    n := 0;
    while n < a.Length
    {
        var x := c0 == 0;
        if x {
            c1 := c1 - 1;
        } else {
            c0 := c0 - 1;
        }
        m := m - multiset{x};
        a[n] := x;
        n := n + 1;
    }
}
```

Let us add invariants for the while statements to prove the specifications.

1.1 The First Loop

Listing 2 shows the invariants for the first loop. The first three invariants are straightforward. They state that `n` is within the range of the array, `m` is equal to the multiset of the elements up to `n`, and the counters `c0` and `c1` are equal to the number of `false` and `true` values in `m`, respectively. The last invariant in the first loop is necessary for the last invariant in the second loop. As a reminder `old` always refers to the intial heap at the beginning of the method - no matter whether it appears in the postcondition, an assertion, or loop invariant. Nested methods, which might cause confusion about the heap to which `old` refers, are not permitted.

Listing 2: The first loop's invariants of `BruteSort`

```
while n < a.Length
    invariant 0 <= n <= a.Length
    invariant m == multiset(a[..n])
    invariant c0 == m[false] && c1 == m[true]
    invariant c0 + c1 == n
```

1.2 The Second Loop and Debugging

Listing 3 shows the final program of the second loop with the comments used for debugging. The first invariant states that `n` is within the range of the array. The second invariant states that the array is sorted up to `n`. This invariant does not hold initially, because we have not proved that while we are writing `false`'s, the prefix are all `false`. This property is expressed by the third invariant.

The fourth invariant means that the left to `n` is updated correctly in terms of the number of `false`'s and `true`'s. This invariant does not hold initially. The fifth invariant is required to link `c0` and `c1` to `m`. The sixth invariant states that the total number of the processed and unprocessed elements remains the same. This invariant follows from the last invariant of the first loop.

To debug the invariant proof, it is common to temporarily insert assertions representing properties that we expect to hold, and once the proof succeeds, remove all unnecessary assertions, leaving only the minimal set that helps the verifier. These assertions help identify which facts are missing from the proof. In this example, we insert the assertions as in line 9 to line 1. The statement `a[..n] == a[..];` remains at the end. This fact appears obvious, but Dafny does not know it by default because adding this as an axiom creates performance problems for the underlying SMT solver. This assertion is required after the while loop terminates as well. When we insert this assertion, Dafny is able to establish the fourth invariant.

Listing 3: The second loop of `BruteSort` with debugging comments

```
1    // assert n == a.Length;
2    assert a[..] == a[..n]; // we introduce a fact that the verifier doesn't already know
3    // assert old(a[..]) == a[..];
4    // assert multiset(old(a[..])) == m;
5    // assert m + multiset{} == m;
6    // assert multiset(a[..0]) == multiset{};
7    // assert multiset(old(a[..])) == m + multiset(a[..0]);
8    n := 0;
9    // assert multiset(old(a[..])) == m + multiset(a[..n]);
10   while n < a.Length
11       invariant 0 <= n <= a.Length
12       invariant forall i, j :: 0 <= i < j < n ==> a[i] ==> a[j]
13       invariant c0 != 0 ==> forall i :: 0 <= i < n ==> !a[i]
14       invariant multiset(old(a[..])) == m + multiset(a[..n])
15       invariant c0 == m[false] && c1 == m[true]
16       invariant c0 + c1 + n == a.Length
17   {
18       ...
19   }
20   assert a[..] == a[..n];
```

1.3 `ghost` prefix

Ghost variables are used only for verification and are erased from the generated executable program. In this example, we can set `c1` and `m` as `ghost` variables. This allows us to introduce proof-only state without affecting runtime behavior.

```
    ghost var c1 := 0;
    ghost var m := multiset{};
```

Dafny also keeps track of aliasing ghost variables with a non-ghost variable, like for a global variable `d`, assigning `d = c1`. This will produce an error.

2 Lemmas and Inductive Proofs

Consider the function `TriangleSum` in Listing 4, which computes the `n`th triangular number. We would like to prove that for all `n`, the number computed by this function is equal to `n * (n + 1) / 2`. How might we do this? One thing we could do is define a method which modifies no variables and whose postcondition is what we seek to prove. It turns out that this is a common enough pattern that Dafny has special syntax for such methods; we declare them as `lemma`s, like `Gauss` on Line 7.

In the body of `Gauss`, we must provide a program (a proof) that satisfies the postcondition (our goal). Our strategy will be to proceed by induction on `n`. In Dafny, there are two ways to do so. First, we could write a recursive procedure that matches on the form of `n`. Second, we could write a loop; this is how we will proceed for this proof.

Listing 4: A function that computes triangular numbers and a lemma

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

// Gauss is a method we call that modifies nothing and returns nothing
// It is an inductive proof of a lemma!
lemma /* method */ Gauss(n: nat)
    ensures TriangleSum(n) == n * (n + 1) / 2
{
    var i := 0;
    while i < n
        invariant 0 <= i <= n
        invariant TriangleSum(i) == i * (i + 1) / 2
    {
        i := i + 1;
    }
}
```

We are obligated to provide a loop invariant for our loop; in the context of a proof this is effectively our inductive hypothesis. The usual trick of looking at our postcondition works well here; the invariant on Line 13 is exactly the same modulo renaming. Dafny is satisfied, and so our lemma is proven!

3 Calculational Proofs

In Listing 5, we can find the usual inductive definition of a `List` datatype, along with recursive helper functions `Length`, `Take`, `Append`, and `Drop`. Note that the latter three functions are declared as *opaque*, meaning that Dafny will not automatically reveal their definitions. Sometimes this is useful; we do this here to demonstrate manually revealing definitions during calculational proofs. Additionally, we turn off automatic induction with {`:induction false`} after `lemma` (without this, Dafny doesn't need our help). Dafny also has co-inductive datatypes, allowing one to reason about infinite streams.

Listing 5: Inductive definition of lists and useful recursive list operations

```
1  datatype List<X> = Nil | Cons(head: X, tail: List<X>)
2
3  function Length<X>(xs: List<X>): nat {
4      match xs
5      case Nil => 0
6      case Cons(_, tail) => 1 + Length(tail)
7  }
8
9  opaque function Append<X>(xs: List<X>, ys: List<X>): List<X>
10 {
11     match xs
12     case Nil => ys
13     case Cons(head, tail) => Cons(head, Append(tail, ys))
14 }
15
16 opaque function Take<X>(xs: List<X>, n: nat): List<X>
17     requires n <= Length(xs)
18 {
19     if n == 0 then Nil else Cons(xs.head, Take(xs.tail, n - 1))
20 }
21
22 opaque function Drop<X>(xs: List<X>, n: nat): List<X>
23     requires n <= Length(xs)
24 {
25     if n == 0 then xs else Drop(xs.tail, n - 1)
26 }
```

We would now like to prove the lemma `AppendTakeDrop` in Listing 6 which relates the three list

functions. As in the previous example, we will proceed by induction on the argument `n`. However, the proof in this scenario is not so simple; extra work is required to convince Dafny that the postcondition is satisfied in both branches.

As a note, it is not immediately obvious when running Dafny in VSCode that both the base and inductive cases need help. Dafny verifies backwards, so a verification error first appears on the inductive case. One technique to pinpoint verification errors is to `assume false` in both branches, trivially discharging the verification condition, and commenting out the `assume false` in the branch of interest, as follows:

Listing 6: Lemma relating our three list functions and debugging statements

```
lemma {:induction false} AppendTakeDrop<X>(xs: List<X>, n: nat)
    requires n <= Length(xs)
    ensures Append(Take(xs, n), Drop(xs, n)) == xs
{
    if n == 0 { // Verification error: Dafny needs help to see the postcondition here.
        // assume false;
    } else {
        assume false;
    }
}
```

To address this, we provide a calculational proof in the `calc` block beginning on Line 6. Our proof here is a series of equalities, but other transitive relations may be used. The first equality is known to the verifier, but the second one is not, since `Take` was declared as an opaque function and is thus not automatically unfolded. In order to direct the verifier to reveal `Take`'s body we provide the `reveal` hint on Line 10. We proceed in a similar fashion for the remainder of the first branch and for most of the second branch, until Line 26. At this point, our hint to Dafny is to use the inductive hypothesis in the form of a recursive call to `AppendTakeDrop` passing `xs.tail` and `n - 1`. This is enough to convince Dafny that our expression can be rewritten to `xs` after a couple more steps.

In the recursive call, Dafny automatically infers that `xs` decreases in each call (since we pass in the tail). In non-obvious cases, this decreasing element can be specified using the `decreases` clause in loop invariants and along with the pre- and postconditions of specifications.

Listing 7: Inductive proof of our lemma containing calculational sub-proofs

```
lemma {:induction false} AppendTakeDrop<X>(xs: List<X>, n: nat)
    requires n <= Length(xs)
    ensures Append(Take(xs, n), Drop(xs, n)) == xs
{
    if n == 0 {
        calc {
            Append(Take(xs, n), Drop(xs, n));
        ==  // what n is
            Append(Take(xs, 0), Drop(xs, 0));
```

```
        ==  { reveal Take; } // proof hint: unfold opaque Take
            Append(Nil, Drop(xs, 0));
        ==  { reveal Drop; }
            Append(Nil, xs);
        ==  { reveal Append; }
            xs;
        }
    } else {
        calc {
            Append(Take(xs, n), Drop(xs, n));
        ==  { reveal Take; }
            Append(Cons(xs.head, Take(xs.tail, n - 1)), Drop(xs, n));
        ==  { reveal Append; }
            Cons(xs.head, Append(Take(xs.tail, n - 1), Drop(xs, n)));
        ==  { reveal Drop; }
            Cons(xs.head, Append(Take(xs.tail, n - 1), Drop(xs.tail, n - 1)));
        ==  { AppendTakeDrop(xs.tail, n - 1); }
            Cons(xs.head, xs.tail);
        ==
            xs;
        }
    }
}
```

If we reveal all the functions above, we can simplify our proof as follows:

```
lemma {:induction false} AppendTakeDrop<X>(xs: List<X>, n: nat)
    requires n <= Length(xs)
    ensures Append(Take(xs, n), Drop(xs, n)) == xs
{
    if n != 0 {
        AppendTakeDrop(xs.tail, n-1);
    }
}
```

Additionally, if we turn induction back on, Dafny can figure out the whole proof itself:

```
lemma AppendTakeDrop<X>(xs: List<X>, n: nat)
    requires n <= Length(xs)
    ensures Append(Take(xs, n), Drop(xs, n)) == xs
{ }
```