Software Testing, Quality Assurance and Maintenance Winter 2025

Lecture 15 — March 24, 2025

Patrick Lam *version 3*

We'll work through some more examples from the stqam-1251-dafny repo, which you can find at `https://git.uwaterloo.ca/stqam-1251/dafny`.

First, consider the greatest common divisor. We will take the textual specification to be this:

```
/*
  Define a recursive function, called gcd, that computes a GCD of two numbers.
  Use the following properties of gcd.
  (a) gcd(n, n) = n
  (b) gcd(m, n) = gcd(m-n, n) whenever m > n
  (c) gcd(m, n) = gcd(m, n-m) whenever n > m
*/
```

We can then write this specification as a (recursive) Dafny function:

```
function gcd(m: nat, n: nat): nat
  requires m > 0 ∧ n > 0
{
  if (m = n) then m
  else if (m > n) then gcd(m − n, n)
  else gcd(m, n − m)
}
```

For our purposes, we take this to be the definition of gcd. I don't think Dafny can prove that the declarative definition of gcd is equal to this recursive definition. Since Dafny's natural numbers start at 0, we specifically define gcd to start at 1 to ensure termination: we are subtracting numbers and we need to subtract at least 1 to $m + n$ on each recursive call if we are to make progress.

The contract of GcdCalc can be as follows:

```
method GcdCalc(m: nat, n: nat) returns (res: nat)
  requires m > 0 ∧ n > 0;
  ensures res = gcd(m, n)
```

That is, we use the gcd function we've defined. And we mirror the precondition of the function.

The implementation is straightforward:

```
{
  var n1, m1 := n, m;
  while (n1 ≠ m1)
  {
    if (m1 < n1) {
     n1 := n1 − m1;
    } else {
     m1 := m1 − n1;
    }
  }
  assert(m1 = n1);
  return n1;
}
```

We don't absolutely need the final assert, but it also doesn't hurt to put it there as documentation. What we do need, though, is an invariant. Without an invariant, Dafny doesn't know what to make

of the loop. So, here, you can read along, and agree with what I'm writing, but really, you need to practice writing some invariants to be able to actually do this. Go do some invariant writing. More than on the assignment.

Anyway, the sensible first invariant to put in is mirroring the precondition. (Indeed, Dafny complains that it can't show the function precondition for function gcd with no precondition). We need both m and n to be positive.

```
invariant m > 0 ∧ n > 0
```

That does help, but Dafny still can't prove termination, nor can it prove the postcondition. Of course. We don't know anything about what the loop is actually computing at this point, just that m and n are strictly positive.

A good guess for the additional invariant is

```
invariant gcd(m, n) = gcd(m1, n1)
```

I don't really know how to give you intuition about this. The invariant has to somehow gcd and you should have some idea that this gcd algorithm is preserving the gcd. For you, it can be based on something you vaguely remember from MATH 135.

Under the hood with Dafny

Some of the things that Dafny does are a form of syntactic sugar which increase its usability. Let's look at this method again and desugar it, using what we know about Hoare Logic.

```
method add_by_inc(n: nat, m: nat) returns (r: nat)
  requires true
  ensures r = m + n
{
  r := n;
  var i := 0;
  while i < m
    invariant i ≤ m
    invariant r = n + i
  {
    i := i + 1;
    r := r + 1;
  }
  return r;
}
```

Dafny converts this into a bunch of verification conditions which it passes along to z3. So, first, let's convert the requires and ensures clauses into assumes/asserts. Also, to make our life easier, we'll combine the two invariants.

```
method add_by_inc(n: nat, m: nat) returns (r: nat)
{
  assume {:axiom} true;
  r := n;
  var i := 0;
  while i < m
    invariant i ≤ m ∧ r = n + i
  {
    i := i + 1;
    r := r + 1;
  }
```

```
  assert r = m + n;
  return r;
}
```

Remember that we had to show that the invariant held upon entry, that it was preserved by the loop; and we exit the loop with the invariant and not the loop condition. Let's do that manually here. First, that the invariant holds upon entry.

```
method add_by_inc(n: nat, m: nat) returns (r: nat)
{
  assume {:axiom} true;
  r := n;
  var i := 0;
  assert i ≤ m ∧ r = n + i;
  while i < m
    invariant i ≤ m ∧ r = n + i
  {
    i := i + 1;
    r := r + 1;
  }
  assert r = m + n;
  return r;
}
```

Next, we want to show that the invariant holds at the end of the loop body, if we know nothing but the invariant upon entry (at least about i and r). We stop the verification after the end of the loop body by putting assume false; no paths escape that assumption, so the verifier just does nothing further there.

```
method add_by_inc(n: nat, m: nat) returns (r: nat)
{
  assume {:axiom} true;
  r := n;
  var i := 0;
  assert i ≤ m ∧ r = n + i;
  // havoc i, r;
  i, r := *, *;
  assume {:axiom} i ≤ m ∧ r = n + i
  if i < m
  {
    i := i + 1;
    r := r + 1;
    assert i ≤ m ∧ r = n + i
    assume {:axiom} false;
  }
  assert r = m + n;
  return r;
}
```

Let's think about what we've done here. In fact, we've made Dafny check the three properties of inductive loop invariants.

1. Clearly, the first assert checks that it holds upon entry to the loop.
2. The havoc and assume erase knowledge of everything (havoc) but the loop invariant (assume) before the body of the loop executes. The loop body executes if the loop condition is true (and Dafny tracks the fact that the loop condition is true), and keeps the loop invariant. Then, Dafny checks that the loop invariant is preserved by the loop body. That's all that it checks for the loop body; it must not propagate facts further.
3. Finally, the do-nothing else branch of the if takes the invariant once again, and conjoins it with the negation of the loop condition, and continues from there; Dafny goes on and checks

what happens after the loop exits—in particular, whether the postcondition exiting the loop is strong enough.

There was also an inferred decreases clause. We can use $\mathsf{m} - \mathsf{i}$ as the thing that decreases. It's easy to desugar that. We're going to use ghost variables; ghost means that a variable is specification-only—it doesn't need to be included in the executable. Nothing bad happens if you make them normal variables, but it's not as neat.

We check that $\mathsf{m} - \mathsf{i}$ is smaller at the end of the loop than it was at the beginning of the loop, and also that it is nonnegative (the integers have infinite decreasing chains while the naturals don't; no infinite decreasing chain and strict decreasing together imply termination).

```
// as before..
{
  ghost var rank0 := m - i;
  i := i + 1;
  r := r + 1;
  ghost var rank1 := m - i;
  assert rank1 < rank0;
  assert rank1 ≥ 0;
  assert i ≤ m ∧ r = n + i
  assume {:axiom} false;
}
```

You can use this desugaring to help prove things. When Dafny is complaining about something, you can explicitly put in the assertions that it's trying to show, look at the code, and understand why the assertion fails. You can also put other assertions to figure out what else Dafny knows. And you can put in assumes to see which case Dafny can't show (we'll talk more about that).

In the dafny repository, you can find the above example as well as one with nested loops (file vcc.sol.dfy).

Fast exponentiation

This example is in fast_exp.prob.dfy and solved in fast_exp.sol.dfy in the Dafny repository. We will work through how we specify and verify the fast exponentiation algorithm, which uses repeated squaring to compute exponents; it uses the fact that

$$e^{2n} = (e^n)^2$$

i.e. to compute e^{2n} you can compute e^n and then square the result.

First, let's define a function exp. This part is straightforward; nothing to see here. The comments in the provided Dafny file provide the definition of exp that we're working from. This is a recursive definition.

```
function exp(x: real, n:nat) : real
{
  if n = 0 then 1.0
  else x * exp(x, n-1)
}
```

Next, we'd like to provide a specification for the provided implementation. Let's first look at the implementation and convince ourselves that it works.

```
method exp_by_sqr (x0: real, n0: nat) returns (r: real)
{
  if (n0 = 0) { return 1.0; }
  if (x0 = 0.0) { return 0.0; }
  var x, n, y := x0, n0, 1.0;
  while (n > 1)
  {
    if (n % 2 = 0)
    {
      x := x * x;
      n := n / 2;
    }
    else
    {
      y := x * y;
      x := x * x;
      n := (n-1)/2;
    }
  }

  return x * y;
}
```

We put aside the base cases, which are clear. In the even case, we see that we are repeatedly squaring x and halving n, which is exactly in line with the identity we saw above. In the odd case, we now have a variable y, which stores extra xs. Here, let's look.

$$x^{2n+1} = x \times x^{2n}$$

So we're assigning $x \times y$ to y, i.e. storing an extra factor of x here. And, in the end, we return x times the leftover ys that we've stored.

Dafny now has test cases, so let's also run a few test cases to convince ourselves that this implementation works. It can be useful to do that before we try to prove something that turns out to be not true. Instead of assert, which is statically verified, we use expect to write what we expect (shout-out to Kieran for finding this; the documentation on Dafny tests is hard to find.)

```
method {:test} TestExpBaseCases()
{
  var res1 := exp_by_sqr(17.0, 0);
  expect res1 = 1.0;
  var res2 := exp_by_sqr(2.2, 1);
  expect res2 = 2.2;
  var res3 := exp_by_sqr(0.0, 11);
  expect res3 = 0.0;
}

method {:test} TestExpEven()
{
  var res1 := exp_by_sqr(1.0, 4);
  expect res1 = 1.0;
  var res2 := exp_by_sqr(2.0, 4);
  expect res2 = 16.0;
  var res3 := exp_by_sqr(3.0, 2);
  expect res3 = 9.0;
}
```

```
method {:test} TestExpOdd()
{
  var res1 := exp_by_sqr(1.0, 5);
  expect res1 = 1.0;
  var res2 := exp_by_sqr(2.0, 5);
  expect res2 = 32.0;
  var res3 := exp_by_sqr(3.0, 3);
  expect res3 = 27.0;
}
```

We can run these test cases. Here I run them on a version of the program that I've verified, but you can also run tests on not-yet-verified code if you annotate relevant methods with {:verify false} and call dafny test --allow-warnings --no-verify.

```
$ dafny test fast_exp.dfy

Dafny program verifier finished with 9 verified, 0 errors
TestExpBaseCases: PASSED
TestExpEven: PASSED
TestExpOdd: PASSED
```

What's the specification? Well, we are returning r and it had better be equal to exp(x0, n0), so here we are:

```
ensures r = exp(x0, n0)
```

Onwards to verification. As always, we at least need loop invariants (if not more). What could they be? Once again, we look at the postcondition, which essentially says x * y == exp(x0, n0). Then, Dafny gives two errors: (1) this invariant is not strong enough to prove the postcondition; and (2) the invariant is also not maintained by the loop. So now we have to debug verification issues.

It seems like it's a better bet to make sure the invariant is strong enough—might as well do that before we try to make it go through. Even if we expect it to be true, let's explicitly write out what we have upon exit to the loop, i.e. the negation of the loop condition plus the invariant. We can also explicitly write the postcondition as an assertion.

```
assert n ≤ 1 ∧ exp(x0, n0) = exp(x, n) * y;
assert x * y = exp(x0, n0)
```

Dafny can of course prove the first assertion but not the second one.

If we look at that first assertion, and we remember that Dafny starts its nat type at 0, then we might have $n = 0$. Is that indeed the case that is causing trouble? We can add an assume n == 1 to see. If we do that, then we see that Dafny is fine with the postcondition. Or, we can add assert n == 1 and see that Dafny can't prove that. We don't ever actually want to have $n = 0$ here, and we can realize that the loop invariant did not constrain n. This reminds us to add the usual kind of invariant on the loop variable:

```
invariant 1 ≤ n ≤ n0
```

and we can see that Dafny then only complains about the invariant being maintained by the loop body, but not about the postcondition.

Recall that what Dafny is effectively doing is putting an assume at the beginning of the loop body and an assert at the end. We can be explicit and put these, and indeed, Dafny complains at the end.

In the loop body, there is the n even case and the n odd case. We can focus on just one case by putting assume false on the other case, so that Dafny doesn't go on and try to verify that the invariant is maintained on that case. Let's handle the even case and put the assume on the odd case. (If we put it on both cases, then Dafny thinks it can verify the thing.) We can move our assert to the then case as well.

```
    assume exp(x0, n0) = exp(x, n) * y;
    if (n % 2 = 0)
```

```
    {
      x := x * x;
      n := n / 2;
      assert exp(x0, n0) = exp(x, n) * y;
    }
    else
    {
      assume false;
      y := x * y;
      x := x * x;
      n := (n-1)/2;
    }
```

What about at the beginning of the then-clause? Of course we can put the loop invariant there. That will work, but it doesn't get us anywhere. But, do we know that exp(x0, n0) == exp(x*x, n/2) * y? We should! (This is what the loop body does). If we assert that, then Dafny says no. What if we move the assume into the body of the then-clause? Dafny still says no.

We can simplify this even more and just assert

```
  assert exp(x, n) = exp(x*x, n/2);
```

and see that Dafny doesn't know about the key property of the exp function that we started this verification with.

We can go and remove extra asserts and assumes; we continue the verification of exp_by_sqr with just one assume:

```
  if n % 2 = 0 {
    assume exp(x, n) = exp(x*x, n/2)
    //...
```

We can do the same process on the else branch to get the assume:

```
  } else {
    assume exp(x, n) = exp(x*x, (n-1)/2) * x;
    //...
```

Using lemmas. Our proof so far has some assumes, which mean that we haven't quite proven correctness yet. We can look at the assume and say, well, of course it's true, but we don't really know. I'm going to deus ex machina some lemmas which we are going to use. Lemmas are functions that tell Dafny some additional facts about their parameters.

```
lemma exp_even (x: real, n: nat)
  requires n % 2 = 0
  ensures exp (x, n) = exp (x*x, n/2)
{
  // needs some code here rather than assume false
  assume false;
}

lemma exp_odd(x: real, n: nat)
  requires n % 2 = 1
  ensures exp (x, n) = exp (x*x, (n-1)/2)*x
{
  // needs some code here rather than assume false
  assume false;
}
```

Let's look at exp_even. If we call it from some other Dafny code, then, just like with a function, the precondition has to hold (i.e. n has to be even). Upon return from this lemma, we know the postcondition—the fact that exp (x, n) == exp (x*x, n/2). Which is just the property that we started with.

Lemmas are ghost code and don't get compiled into the executable. They exist only for their effects on the proof. Dafny takes them into account when it's verifying code. You also have to prove that the postcondition is actually true, but we will first use the lemma and then prove it. Here's what that looks like.

```
if (n % 2 = 0)
{
  exp_even(x,n);
  x := x * x;
  n := n / 2;
}
else
{
  exp_odd(x,n);
  y := x * y;
  x := x * x;
  n := (n-1)/2;
}
```

The precondition of the lemmas holds because it is just the immediately-preceding if condition. Dafny can therefore know that the postcondition is true. It happens that these postconditions are exactly the same as the assumes that we had previously inserted, so Dafny can verify the whole exp_by_sqr method (modulo the still-missing proofs of the lemmas).

Proving lemmas. Before we prove the lemmas for exp_by_sqr, we can learn how to prove some simpler lemmas. The file lemmas.prob.inclass.dfy contains some example lemmas which we'll work through.

We usually need lemmas when we have some function which Dafny doesn't know the properties of. Dafny is pretty good with properties of its + operator, but let's provide a add function which does the same thing. Dafny doesn't know that function.

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

/* Lemma: x + 0 = 0 */
lemma add_zero_lemma (x: nat)
  ensures add (x, 0) = x
{
  // REPLACE BY PROOF
  assume(false);
}
```

It turns out that Dafny is actually smart enough to prove add_zero_lemma on its own, but let's provide our own proof of it, as a pedagogical exercise.

Dafny supports calc blocks, where we put a sequence of lines that are each equal to the next one. Using these blocks, we can supply calculational proofs.

```
lemma add_zero_lemma (x: nat)
```

```
  ensures add (x, 0) = x
{
  calc {
    add(x,0);
    x;
  }
}
```

This calc block shows that add(x,0) == x, which we knew already. If we make it wrong on purpose (e.g. add(x,1) == x), then Dafny complains.

This next lemma is actually harder to prove—it doesn't follow immediately from the definition of add. Dafny is still smart enough to prove it automatically, but we'll do it manually again.

```
lemma zero_add_lemma (x: nat)
  ensures add (0, x) = x
{
  assume(false);
}
```

We can split the proof into cases, e.g.

```
lemma zero_add_lemma (x: nat)
  ensures add (0, x) = x
{
  if x = 0 {
  } else if x = 1 {
  } else {
    assume false;
  }
}
```

Dafny verifies this, so we know that it can handle the 0 and 1 cases automatically. (These are add(0,0) == 0 and add(0, 1) == 1 respectively). Let's write a calc block for the other cases.

```
  calc {
    add(0,x);
    add(0, x-1)+1;
  }
```

What we really want to do now is to invoke induction, using 0 and 1 as the base cases. We can write the inductive hypothesis as an assume for calc:

```
  calc {
    add(0,x);
    add(0, x-1)+1;
    {zero_add_lemma(x-1);}
    (x-1)+1;
    x;
  }
```

Dafny checks that the preconditions to zero_add_lemma are met. Since there are none, those hold. Also, because it is a recursive call inside the braces, Dafny checks that the recursion is well-founded: the parameter x-1 is smaller than x, and there are base cases. We can then use the postcondition to replace add(0, x-1) with (x-1) in the calc block, and the rest of the calculation is easy.

There are a few more lemmas that Dafny can prove automatically. It's good to try to prove them manually as well, and the solutions are in lemmas.sol.dfy. I will put this one here, though.

```
lemma one_plus_add (x: nat, y: nat)
  ensures add(x, y) + 1 = add (x+1, y)
{ }
```

Here is one that Dafny cannot prove automatically.

```
/** Lemma: (x + y) = (y + x)
*/
lemma add_comm_lemma (x: nat, y: nat)
  ensures add(x, y) = add(y, x)
{
}
```

We get the usual error that there is a postcondition that could not be proved.

We can try some base cases:

```
/** Lemma: (x + y) = (y + x) */
lemma add_comm_lemma (x: nat, y: nat)
  ensures add(x, y) = add(y, x)
{
  if x = 0 { }
  else if y = 0 { }
  else {
  }
}
```

and see that those cases verify, but not the general case. Time to put a calc block.

```
  calc {
    add(x, y);
    add(x, y−1) + 1; // from the definition of add
```

This is straightforward so far. Then, we want to use induction again. The obvious thing to use induction on is what we have on the previous line, arguments (x, y-1). And we can then deploy the postcondition of the present lemma. Note that if you get the inductive arguments wrong, Dafny will indeed complain.

```
    {add_comm_lemma(x,y−1);}
    add(y−1,x) + 1;
```

Dafny can actually do it pretty much automatically from here, but let's explicitly also bring the +1 back into the add() parameters using induction again. The remaining steps after that are straightforward. We'll be pedantic about -1+1, though Dafny doesn't need that either.

```
    {one_plus_add(y−1,x);}
    add(y−1 + 1, x);
    add(y,x);
  }
```

Proving fast_exp lemmas. You don't actually need to use calc to prove these—you can iterate on n and write some invariants—but the solutions in fast_exp.sol.dfy do use calc and mutual recursion, and I guess that's nicer? You decide.

```
lemma exp_even (x:real, n:nat)
  requires n % 2 = 0
  ensures exp (x, n) = exp (x*x, n/2)
{
  if (n = 0) { }
  else
  {
    calc
    {
      exp (x, n);
      x * exp (x, n−1);
```

```
      { exp_odd (x, n-1); }
      x * x * exp (x*x, (n-2)/2);
      exp (x*x, n/2);
    }
  }
}

lemma exp_odd(x:real, n:nat)
  requires n % 2 = 1
  ensures exp (x, n) = exp (x*x, (n-1)/2)*x
{
  if (n = 0) { }
  else
  {
    calc
    {
      exp (x, n);
      x * exp (x, n-1);
      { exp_even (x, n-1); }
      x * exp (x*x, (n-1)/2);
    }
  }
}
```

Here are my iterative proofs. exp_odd just calls exp_even since, if n is odd, then n-1 is even; and I didn't use exp_odd in my exp_even proof.

```
lemma exp_even (x:real, n:nat)
  requires n % 2 = 0
  ensures exp (x, n) = exp (x*x, n/2)
{
  var i := 0;
  var r, rr := 1.0, 1.0;
  while i < n/2
    invariant 0 ≤ i ≤ n/2
    invariant r * r = rr
    invariant exp(x, i) = r
    invariant exp(x*x, i) = rr
    invariant exp(x, i*2) = rr
  {
    r := r * x;
    rr := rr * x * x;
    i := i + 1;
  }
}

lemma exp_odd(x:real, n:nat)
  requires n % 2 = 1
  ensures (exp (x, n) = exp (x*x, (n-1)/2)*x)
{
  exp_even(x, n-1);
}
```