Software Testing, Quality Assurance and Maintenance Winter 2025

Lecture 14 — March 10, 2025

Patrick Lam *version 1*

We motivated Dafny last time by talking about Cedar. In brief, Dafny proves user-specified annotations and verifies the absence of runtime errors—the things that are usually undefined behaviour or crashes, like out of bounds array indexes, null dereferences, etc. We've seen how to use symbolic execution to avoid these as well, but Dafny is a fully-static approach that aims to work on real code (written in the Dafny language).

Other applications of verified software besides Cedar:

- Paris Metro line 14 control system (B)[1]: 110,000 lines of models, 86,000 lines of Ada.
- seL4 (Haskell, Isabelle/HOL, C)[2]: the seL4 microkernel conforms to its specification and maintains integrity and confidentiality.
- CompCert (Coq)[3]: a formally verified optimizing compiler
- IronClad (Dafny)[4]: guarantees about how remote apps behave

The first three applications were carried out by verification experts; the last one was done by systems programmers. In all of these cases, the systems were designed as verified systems from the start.

Dafny

I don't know how long it will take, but in this set of lecture notes, we'll aim to get through the first online guide, which shows how to verify basic imperative code (including loops but not advanced topics like objects, sequences and sets, data structures, lemmas). I'm going to summarize what's there, and in class, we'll work on exercises together.

- `https://dafny.org/dafny/OnlineTutorial/guide`
- `https://dafny.org/blog/2023/12/15/teaching-program-verification-in-dafny-at-amazon/`

Dafny: annotations

This annotation in Dafny should be familiar to you from what we've seen in class:

```
∀ k: int • 0 ≤ k < a.Length ⟹ 0 < a[k]
```

or in ASCII,

[1] `https://www.clearsy.com/wp-content/uploads/2020/03/Formal-methods-for-Railways-brochure-mai-2020.pdf`

[2] `https://sel4.systems/Info/FAQ/proof.html`

[3] https://inria.hal.science/hal-01238879

[4] https://www.microsoft.com/en-us/research/project/ironclad/

```
forall k: int :: 0 <= k < a.Length ==> 0 < a[k]
```

It says that all elements of array a are positive.

Let's start Dafny and write a method that ensures this as a precondition. Can you fill this in on your computer, experimenting with how to make it verify?

```
method all_positive() returns (rv: array<int>)
ensures ∀ k: int • 0 ≤ k < rv.Length ⟹ 0 < rv[k]
{
  var arr := new int[2];
  //...
  return arr;
}
```

Dafny: methods

We saw a method above. A *method* is a piece of imperative, executable code. It's like a Java method, or a procedure or function in other languages. Dafny uses "function" to mean something else. Here's another method declaration:

```
method Abs(x: int) returns (y: int)
{
  //...
}
```

Dafny methods can return multiple values:

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

Note the use of ":=" for assignment (and "==" for equality).

Return variables are just like local variables and can be assigned multiple times. Parameters are read-only.

Let's fill in the body of Abs.

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

Preconditions and postconditions

What's special about Dafny, of course, is that you can write requires and ensures clauses on methods, and Dafny will try to verify them, or complain. Let's add a postcondition to the Abs method.

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

Note that we use the name of the return value, y. Now, why doesn't this verify?

We can write multiple postconditions as well.

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

OK, what goes wrong here? How can we verify this method?

This doesn't help, but we can also write:

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

There are two reasons why Dafny can't prove something. (1) it's "too hard" for Dafny and Boogie/Z3 to prove; (2) it's actually not true. Test cases can help with (2). The other thing you will need, in general, is invariants.

So, of course less $<$ x isn't true if y is negative. Let's make sure that it's impossible to call MultipleReturns in that case.

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

You can also write more than one requires clause. OK, here's an exercise.

Exercise 0. Write a method Max that takes two integer parameters and returns their maximum. Add appropriate annotations and make sure your code verifies.

```
method Max(a: int, b: int) returns (c: int)
  // write a postcondition
{
  // write code
}
```

Assertions

We can put assertions anywhere, for instance:

```
method TryingOutAssertions() {
  assert 2 < 3;
  // what about asserting something not true?
}
```

Assertions are especially helpful for debugging Dafny, because it tries to prove that they hold on all executions. So if you think that Dafny knows something, write it as an assertion and see if it does or not.

Local variables are useful, including for assertions.

```
method m()
{
  var x, y, z: bool := 1, 2, true;
}
```

Dafny infers types for local variables, or you can explicitly specify them. Using variables:

```
method TestAbsWithPostcondition(){
  var v := AbsWithPostcondition(3);
  assert 0 ≤ v;
}
```

Exercise 1. Write a test method that calls your Max method from Exercise 0 and then asserts something about the result.

But what about:

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

method TestAbs()
{
  var v := AbsWithPostcondition(3);
  assert 0 ≤ v;
  assert v = 3; // oops, can't prove this
}
```

Looking at AbsWithPostcondition, clearly v is 3 if we use the body of that method. But Dafny doesn't know that. It just looks at the postcondition. From the postcondition, it knows that v is non-negative, but not exactly what it is.

Indeed, Dafny doesn't know the difference between AbsWithPostcondition and:

```
method NotAbs(x: int) returns (y: int)
ensures 0 ≤ y
{
  return 5;
}
```

and it also can't prove this:

```
method TestNotAbs(x: int) returns (y: int)
ensures 0 ≤ y
{
  var v := NotAbs(3);
  assert v = 3; // actually not true
}
```

So what postcondition do we want? How about this?

```
method AbsBetterPostcondition(x: int) returns (y: int)
ensures 0 ≤ y
ensures 0 ≤ x ⟹ y = x
{
  if x < 0 {
    return −x;
  } else {
    return x;
  }
}
```

Well...

```
method TestAbsBetter(x: int) returns (y: int)
{
  var v := AbsBetterPostcondition(5);
  assert v = 5;
  var w := AbsBetterPostcondition(−2);
  assert w = 2;
}
```

OK, one way to write the postcondition we really want is:

```
method AbsFullPostcondition(x: int) returns (y: int)
ensures 0 ≤ y
ensures 0 ≤ x ⟹ y = x
ensures x < 0 ⟹ y = −x
{
  if x < 0 {
    return −x;
  } else {
    return x;
  }
}
```

Or we can write:

```
ensures 0 ≤ y ∧ (y = x ∨ y = −x)
```

There can be more than one way to write a postcondition.

Now, we want to eliminate the redundancy between the postcondition and the implementation. Functions can help here. But, first, some exercises.

Exercise 2. Using a precondition, change Abs such that it can only be called on negative values. Simplify the body of Abs into just one return statement and make sure the method still verifies.

Exercise 3. Keeping the postconditions of Abs the same as for AbsFullPostcondition, change the body of the method to just y := x + 2. What precondition do you need to impose on the method

so that it verifies? What about body y := x + 1? What does that precondition say about when you can call the method?

Functions

Here is a function. A function body must consist of exactly one expression with the correct type.

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

Why functions? They can be used directly in specifications (e.g. asserts, requires, ensures). Here, look.

```
method m()
{
  assert abs(3) = 3;
}
```

We don't need to store the result of function abs in a temporary local variable, nor did we have to write a postcondition for it. Dafny doesn't forget about the function body. It is allowed to write preconditions and postconditions for functions, but not required.

Exercise 4. Write a function max that returns the larger of two integer parameters. Write a test method using an assert that checks that your function is correct (at least on one case).

Exercise 5. Change the postcondition of method Abs to use function abs, make sure that Abs still verifies, and then change the body of Abs to also use the function.

Let's continue using functions. Here is a naïve implementation of the Fibonacci function.

```
function fib(n: nat): nat
{
  if n = 0 then 0
  else if n = 1 then 1
  else fib(n-1) + fib(n-2)
}
```

Some notes. (1) We had ints before, but now we have nats, i.e. natural numbers, which can't be negative; these are a subset type of int. (By the way, Dafny ints and nats are unbounded.) (2) You wouldn't actually want to *calculate* Fibonacci numbers this way, but it obviously matches the definition, and we can ask Dafny to prove that a (faster) implementation also computes the same thing as this function.

```
method ComputeFib(n: nat) returns (b: nat)
  ensures b = fib(n)
{
  //...
}
```

Loops and Loop Invariants

The usual more efficient way to compute Fibonacci numbers is using a loop. We've talked about loops in the context of Hoare logic, and Dafny is similar, except that it does a lot of proving for you. But it doesn't supply invariants.

Here's a loop, with an invariant.

```
method FirstLoop(n: nat)
{
  var i := 0;
  while i < n
    invariant 0 ≤ i
  {
    i := i + 1;
  }
}
```

We can see that i == n at the end of the loop, but does Dafny know that? We can ask it, using assert. (It doesn't). We need to strengthen the loop invariant. If we try $0 \leq i < n$, then Dafny complains that the invariant might not hold on entry and it might not be maintained by the loop body. Why is that?

Well, in any case, if we use the invariant $i \leq i \leq n$, then Dafny is satisfied with everything.