

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)<sup>1</sup>: 110,000 lines of models, 86,000 lines of Ada.
- seL4 (Haskell, Isabelle/HOL, C)<sup>2</sup>: the seL4 microkernel conforms to its specification and maintains integrity and confidentiality.
- CompCert (Coq)<sup>3</sup>: a formally verified optimizing compiler
- IronClad (Dafny)<sup>4</sup>: 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.

We're going to talk more about formal verification and software development in Lecture 12.

## 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:

$\forall k: \text{int} \bullet 0 \leq k < a.\text{Length} \implies 0 < a[k]$

---

<sup>1</sup><https://www.clearsy.com/wp-content/uploads/2020/03/Formal-methods-for-Railways-brochure-mai-2020.pdf>

<sup>2</sup><https://sel4.systems/Info/FAQ/proof.html>

<sup>3</sup><https://inria.hal.science/hal-01238879>

<sup>4</sup><https://www.microsoft.com/en-us/research/project/ironclad/>

or in ASCII,

```
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 postcondition. 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  $\text{Abs}$ .

```
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  $\text{Abs}$  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 (under the hood) 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  $\geq$  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, which must have 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), e.g.

```
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. It can serve as an oracle, more obviously matching the definition of Fibonacci numbers than the iterative implementation.

```
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. You've seen 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;
    }
    assert i = n;
}
```

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  $0 \leq i \leq n$ , then Dafny is satisfied with everything.

**Exercise 6.** Change the loop invariant to  $0 \leq i \leq n + 2$ . Does the loop still verify? Does the assertion  $i == n$  after the loop still verify?

**Exercise 7.** With the original loop invariant, change the loop guard from  $i < n$  to  $i \neq n$ . Do the loop and assertion after the loop still verify? Why or why not?

Here is another loop, which implements multiplication using addition.

```
method AddByInc(n: nat, m: nat) returns (r: nat)
    ensures r = n + m
{
    r := n;
    var i := 0;
    while (i < m)
    {
        i := i + 1;
        r := r - 1;
    }
}
```

Turns out that we need to add two invariants for the while:

```
invariant 0 ≤ i ≤ m;
invariant r = n + i;
```

These are similar invariants to what we've seen in Hoare logic, but it's easier to experiment with them here and see exactly what we need and when Dafny complains. The second invariant really is an invariant. We can express it as  $n = r - i$ . The loop doesn't modify  $n$ . So, as  $i$  goes up,  $r$  goes down by the same amount.

We can also check some assertions at the end to see what Dafny actually knows:

```

assert (r = n + i  $\wedge$  i = m);
assert (r = n + m);

```

Here's a method that multiplies two numbers by addition.

```

method Product(m: nat, n: nat) returns (res: nat)
  ensures res = m*n
{
  var m1 := m;
  res := 0;
  while m1  $\neq$  0
  {
    var n1 := n;
    while n1  $\neq$  0
    {
      res := res + 1;
      n1 := n1 - 1;
    }
    m1 := m1 - 1;
  }
}

```

We need invariants for both while loops. We can start with the outer loop. We could write an invariant about the range of **m1** but it turns out that this is not mandatory. The invariant that actually is mandatory is about the value of **res**. It looks like magic, but you can examine the postcondition and the fact that **m1** is the induction variable here, and conclude that your invariant should be:

```

invariant res = (m-m1) * n

```

When the loop starts, it has gone around the loop zero times, so  $m == m1$  and their difference is 0, as is **res**. In terms of maintaining the invariant, each time around the outer loop, the program is adding one **n** to **res** and subtracting 1 from **m1**, and the invariant does that.

As for the second invariant, we are now doing the adding of **n** to **res** one at a time. Through this loop, **m** and **m1** are both constant, and we are changing **res** from  $(m - m1) \times n$  to  $(m - m1) + 1 \times n$ . We have that **n** stays constant and **n1** goes down. Every time that **n1** goes down by 1, **res** goes up by 1, so putting  $-n1$  is a good bet. The invariant is thus

```
1   invariant res == (m-m1)*n + (n-n1)
```

and Dafny will verify this.

Let's implement the Fibonacci method.

```

method ComputeFib(n: nat) returns (b: nat)
  ensures b = fib(n)
{
  var i := 1;
  var a := 0;
  b := 1;
  while i < n
  {
    a, b := b, a + b;
    i := i + 1;
  }
}

```

(Note that Dafny allows assignment to **a** and **b** simultaneously).

We have postcondition  $b = \text{fib}(n)$ , and we also know that  $i == n$ , so it must be that  $b == \text{fib}(i)$  which we can have as a loop invariant. This is promising as part of the loop invariant: it talks about the loop counter. Hence, we have what we had before, plus this.

```
invariant 0 ≤ i ≤ n
invariant b = fib(i)
```

We also know that  $a$  is the previous Fibonacci number. So,

```
invariant a = fib(i - 1)
```

Finally, since Dafny `nat` starts at 0, we include:

```
if n = 0 { return 0; }
```

All together:

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

method ComputeFib(n: nat) returns (b: nat)
    ensures b = fib(n)
{
    if n = 0 { return 0; }
    var i := 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;
    }
}
```

**Exercise 8.** It is possible to simplify `ComputeFib`. Write a simpler program by not introducing `a` as the Fibonacci number that precedes `b`, but instead introducing `c` as the variable that succeeds `b`. Verify that your program is correct according to the mathematical definition of Fibonacci.

(I'm omitting Exercise 9 from the tutorial because it's not really that useful.)

As you know, an issue with just verifying partial correctness is that it doesn't tell you anything at all if the program doesn't terminate. When you run the program, you can see it not terminating, but if you're just proving it, you might write non-terminating code.

## Termination

Dafny does prove termination for loops and recursion. It does so by proving that there is something that decreases and that is bounded. Usually Dafny manages to guess what is decreasing. When that is something over natural numbers, the lower bound is often 0. You can hover over the loop

to see the inferred decreases clause. Sometimes Dafny can't guess, and you have to provide an annotation.

```
method Decreases()
{
  var i := 20;
  while 0 < i
    invariant 0 ≤ i
    decreases i // could be inferred
    {
      i := i - 1;
    }
}
```

Note that the thing that is decreasing might be the negation of something that's increasing.

```
method Decreases2()
{
  var i, n := 0, 20;
  while i < n
    invariant 0 ≤ i ≤ n
    decreases n - i
    {
      i := i + 1;
    }
}
```

Dafny can guess this one too, from the loop condition  $i < n$ . The lower bound is from  $i \leq n$  in the invariant, which implies  $n - i \geq 0$ . The bound doesn't have to be constant, as in binary search.

**Exercise 10.** In the loop above, the invariant  $i \leq n$  and the negation of the loop guard allow us to conclude  $i == n$  after the loop, as we've previously checked with an **assert**. If the loop guard was instead  $i \neq n$  (as in Exercise 7), then the negation of the guard immediately gives  $i == n$ , without needing the loop invariant. Change the loop guard to  $i \neq n$  and delete the invariant. What happens?

We have also seen the recursive function **fib**. Dafny also guesses the decreases annotation here. We could specify it explicitly:

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

There is a longer Dafny tutorial on termination, and we'll probably go through that one as well.

## Arrays

In Dafny we have one-dimensional arrays **array** $\langle T \rangle$  for any  $T$  (though here we'll only consider **array** $\langle \text{int} \rangle$ ), as well as **array?** $\langle T \rangle$  which includes possibly-null arrays.

Array access is as you might expect,  $a[i]$ , and there is a built-in length field: **a.Length**. Part of Dafny's compile-time verification includes showing that all array accesses are in-bounds, i.e. between 0 and **Length** – 1 inclusive.

We saw array allocation right at the beginning of this document:

```
var arr := new int[2];
```

Let's express the easy part of the contract for the `Find` method.

```
method FindPartialContract(a: array<int>, key: int) returns (index: int)
  // partial contract
  ensures 0 ≤ index ⇒ index < a.Length ∧ a[index] = key
{
  // can you write code that satisfies this postcondition?
  // it can be done with one statement.
}
```

This is the case where the key is in the array. It's at index `index`. The conjunct `index < a.Length` guards against out-of-bounds array accesses that might be caused by `a[index]`, and works because Dafny has short-circuit logical ands/ors (like C and many other languages).

For the other case, we need to say that `key` does not exist at *any* index in the array, so we need quantifiers.

**Quantifiers.** Dafny supports quantifiers in expressions. Use a double-colon (::) after the variable name; it shows up as a dark · in Emacs and VS Code.

```
method WithQuantifier()
{
  assert ∀ k • k < k + 1;
}
```

(This verifies, but there's a warning about triggers. We won't talk about this here.)

Typically we quantify over some more limited set than “all integers” (Dafny infers the type of `k`, though you can specify it if you want). For instance, all integers that are valid indices into an array, i.e. between 0 and `a.Length`.

```
∀ k • 0 ≤ k ≤ a.Length ⇒ a[k] ≠ key
```

We can thus write the whole contract:

```
method Find(a: array<int>, key: int) returns (index: int)
  ensures 0 ≤ index ⇒ index < a.Length ∧ a[index] = key
  ensures index < 0 ⇒ ∀ k • 0 ≤ k < a.Length ⇒ a[k] ≠ key
{
  // TODO implement this
}
```

Linear search is the easiest way to implement `Find`:

```
index := 0;
while index < a.Length {
  if a[index] = key {
    return;
  }
  index := index + 1;
}
index := -1;
```

and when we plug that into the contract, we see that Dafny can't prove the postcondition saying that if `index` is negative, then `key` is not in the array. Makes sense: all we know upon leaving the loop is that `index` has reached the end of the array. We need to add an invariant showing that the part of the array we've visited so far does not contain `key`.

```
invariant ∀ k • 0 ≤ k < index ⇒ a[k] ≠ key
```

Dafny complains: it doesn't know that `index` doesn't go off the end of the array. This invariant works.

```
invariant 0 ≤ index ≤ a.Length
```

We often see loop invariants where we build the property that we need one element at a time, as we do here. In this case, the property is that the visited part of the array doesn't contain the key that we're looking for.

**Exercise 11.** Write a method that takes an integer array, which it requires to have at least one element, and returns an index to the maximum of the array's elements. Annotate the method with pre- and postconditions that state the intent of the method, and annotate its body with a loop invariant to verify it.

```
method FindMax(a: array<int>) returns (index: int)
  // TODO put preconditions and postconditions here
{
  // TODO implement this
}
```

Linear search is fine, but if the array is sorted, then binary search is faster. We can't prove that binary search is correct unless we can somehow express a property about the array being sorted. Which we can. But let's see if we can refactor this property, using predicates.

**Predicates.** Dafny has functions, which can return anything. Dafny also has predicates, which return booleans. We are going to define the `sorted` predicate which will return true if and only iff its array argument is sorted. Using this predicate makes our expressions shorter and more readable (as long as the name of the predicate corresponds to its meaning).

```
predicate sorted(a: array<int>)
{
  ∀ j, k • 0 ≤ j < k < a.Length ⇒ a[j] ≤ a[k]
}
```

A predicate is like a function, but the return type must be boolean, and so you don't have to write this.

Dafny won't compile this, and says that there is an insufficient `reads` clause.

**Framing.** Dafny needs to know when heap locations might change.

We know that functions and predicates can't change the heap. But Dafny isn't a purely functional language—methods can change the heap.

So, Dafny wants to know when the value of `sorted` might change between calls. In a world where there are two arrays, and a method writes to one of the arrays, a `reads` annotation tells Dafny *which* array `sorted` is reading from, and whether that array might have been changed by the method. If not, then the array has the same sortedness before and after the method was called.

```
predicate sorted(a: array<int>)
  reads a
```

```
{
   $\forall j, k \bullet 0 \leq j < k < a.Length \implies a[j] \leq a[k]$ 
}
```

The `reads` annotation specifies a set of memory locations that the function is allowed to access. Here, it is an array, which means that `sorted` can access all of the elements of the array. It could also be object fields and sets of objects. We'll probably check that later.

As we've seen, Dafny checks that the `reads` clause specifies everything that the function might read. This includes any functions that the function transitively calls.

Conversely, as we've seen, methods can read anything they want. But they must declare any changes to memory using a `modifies` clause.

These clauses allow Dafny to reason about methods and functions in isolation, and hence they make verification much more tractable.

Dafny only requires framing information for memory on the heap, which includes arrays and objects, but not sets, sequences, or multisets (which we'll talk about later).

**Exercise 12.** Create a `sortedAndDistinct` predicate that returns true exactly when the array is sorted and all its elements are distinct.

**Exercise 13.** Create a `sortedAndNotNull` predicate that accepts references to a possibly-null array and that returns true exactly when the array is sorted and not null.

**Binary Search.** Predicates: software engineering principles for annotations. Here we use the `sorted` predicate in the precondition for `BinarySearch`. Dafny thus knows that the array is sorted when verifying the method body.

```
method BinarySearch(a: array<int>, value: int) returns (index: int)
  requires 0 ≤ a.Length ∧ sorted(a)
  ensures 0 ≤ index ⇒ index < a.Length ∧ a[index] = value
  ensures index < 0 ⇒ ∀ k • 0 ≤ k < a.Length ⇒ a[k] ≠ value
{
  //...
}
```

Here's an implementation of binary search.

```
var low, high := 0, a.Length;
while low < high
  invariant 0 ≤ low ≤ high ≤ a.Length
  invariant ∀ 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;
```

Let's look at the implementation and the invariants.

```
invariant 0 ≤ low ≤ high ≤ a.Length
invariant ∀ i •
  0 ≤ i < a.Length ∧ ¬(low ≤ i < high) ⇒ a[i] ≠ value
```

The first invariant says that the search is looking between  $[low, high]$ , and the second invariant says that the searched-for value is not anywhere outside that range.

Reasoning about how the first invariant continues to hold is straightforward.

For the second invariant, Dafny needs to show that changing `low` or `high` doesn't miss the value that we are searching for. It can use the fact that predicate `sorted` holds. Taking the first if case, we are bumping up `low` to above `mid`, based on sortedness and the fact that  $a[mid] < value$ , which ensure that the value isn't in that part of the range.

Off-by-one errors are easy to write. Dafny catches them in this case.

**Exercise 14.** Change the assignments in the body of `BinarySearch` to set `low` to `mid` or to set `high` to `mid - 1`. What goes wrong?

## Conclusion

We've seen a bunch of the major features of Dafny and used it to verify some simple code, up to a binary search implementation. There are more features: objects, sequences and sets, data structures, lemmas, modules, etc. We'll probably talk about them in the next few lectures. You don't really need them to do the assignment this year. Maybe next year.

There is a lot of information about Dafny online, including other tutorials and the reference.

SE 465 is not actually the course on specifications. That's SE 463. But we use formal specifications in this part of SE 465 to prove that implementations conform to these specifications, and that they are using other code consistently with that code's specifications. Both tests and verification are tools that you can use to verify that your code does what it's supposed to do. Tests are more practical, while verification is conceptually more powerful.