Getting Started with Dafny: A Guide

Jason KOENIG [a] and K. Rustan M. LEINO [b]
[a] *Carnegie Mellon University, Pittsburgh, PA, USA*
[b] *Microsoft Research, Redmond, WA, USA*

Abstract. Common program specification and verification build on concepts like method pre- and postconditions and loop invariants. These lectures notes teach those concepts in the context of the language and verifier Dafny.

Keywords. Automatic program verification, program specification, Dafny, precondition, postcondition, loop invariant, termination specification

0. Introduction

Dafny is a language that is designed to make it easy to write correct code. This means correct in the sense of not having any runtime errors, but also correct in actually doing what the programmer intended it to do. To accomplish this, Dafny relies on high-level annotations to reason about and prove correctness of code. The effect of a piece of code can be given abstractly, using a natural, high-level expression of the desired behavior, which is easier and less error prone to write. Dafny then generates a proof that the code matches the annotations (assuming they are correct, of course!). Dafny lifts the burden of writing bug-free *code* into that of writing bug-free *annotations*. This is often easier than writing the code, because annotations are shorter and more direct. For example, the following fragment of annotation in Dafny says that every element of the array is strictly positive:

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

This says that for all integers k that are indices into the array, the value at that index is greater than zero. By writing these annotations, one is confident that the code is correct. Further, the very act of writing the annotations can help one understand what the code is doing at a deeper level.

In addition to proving a correspondence to user supplied annotations, Dafny proves that there are no run time errors, such as index out of bounds, null dereferences, division by zero, etc. This guarantee is a powerful one, and is a strong case in and of itself for the use of Dafny and tools like it. Dafny also proves the termination of code, except in specially designated loops.

These lecture notes give a guide to Dafny, explaining the specification features that get used routinely in all verification tasks. Although the lecture notes focus on Dafny, the concepts explained apply to program specification and verificaton more generally.

The appendix of these lecture notes contains a reference guide to a large subset of the Dafny language. Turn to it if you need to know more about the syntax or constructs available.

The best way to learn from these notes is to try things out yourself. There are three ways to run the Dafny verifier. The slickest way is to run it in the Microsoft Visual Studio integrated development environment, since this continuously runs the verifier in the background and highlights errors in your program as you develop it. Another way is to run Dafny from the command line, which requires a .NET virtual machine (built in on Windows platforms, and provided by Mono on non-Windows platforms). The third way to run Dafny uses any web browser and requires no installation.[0] More information about these options and installation are available from the Dafny homepage.[1]

Let's get started writing some Dafny programs.

1. Methods

In many ways, Dafny is a typical imperative programming language. There are methods, variables, types, loops, if statements, arrays, integers, and more. One of the basic units of any Dafny program is the *method*. A method is a piece of imperative, executable code. In other languages, they might be called procedures, or functions, but in Dafny the term "function" is reserved for a different concept that we will cover later. A method is declared in the following way:

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

This declares a method called "`Abs`" which takes a single integer parameter, called "`x`", and returns a single integer, called "`r`". Note that the types are required for each parameter and return value, and follow each name after a colon ("`:`"). Also, the return values (out parameters) are named, and there can be multiple return values, as in below:

```
method MultipleReturns(x: int, y: int) returns (more: int, less: int)
{
   ...
}
```

The method body is the code contained within the braces, which until now has been cleverly represented as "..." (which is *not* Dafny syntax). The body consists of a series of *statements*, such as the familiar imperative assignments, **`if`** statements, loops, other method calls, return statements, etc. For example, the `MultipleReturns` method may be implemented as:

[0] `rise4fun.com`
[1] research.microsoft.com/dafny

```
method MultipleReturns(x: int, y: int) returns (more: int, less: int)
{
   more := x + y;
   less := x - y;
   // comments: are not strictly necessary.
   /* unless you want to keep your sanity. */
}
```

Assignments do not use "=", but rather ":=". (In fact, as Dafny uses "==" for equality, there is no use of a single equals sign in Dafny statements and expressions.) Simple statements must be followed by a semicolon, and whitespace and comments are ignored. To return a value from a method, the value is assigned to one of the named return values sometime before the method returns. In fact, the return values act very much like local variables, and can be assigned to more than once. The input parameters, however, are read only. Return statements are used when one wants to return before reaching the end of the body block of the method. Return statements can be just the **return** keyword (where the current value of the out parameters are used), or they can take a list of values to return. There are also compound statements, such as **if** statements. **if** statements require parentheses around the boolean condition, and act as one would expect:

```
method Abs(x: int) returns (r: int)
{
   if (x < 0)
      { return -x; }
   else
      { return x; }
}
```

One caveat is that they always need braces around the branches, even if the branch only contains a single statement (compound or otherwise). Here the **if** statement checks whether `x` is less than zero, using the familiar comparison operator syntax, and sets `r` to the absolute value as appropriate. (Other comparison operators are `<=`, `>`, `>=`, `!=`, and `==`, with the expected meaning.) Here we use the implicit return at the end of the function. Regardless of the branch taken, the correct value is put into `r`, so when we get to the end of the function, the result is returned properly.

2. Pre- and Postconditions

None of what we have seen so far has any annotations: the code could be written in virtually any imperative language (with appropriate considerations for multiple return values). The real power of Dafny comes from the ability to annotate these methods to specify their behavior. For example, one property that we observe with the `Abs` method is that the result is always greater than or equal to zero, regardless of the input. We could put this observation in a comment, but then we would have no way to know whether the method actually had this property. Further, if someone came along and changed the method, we wouldn't be guaranteed that the comment was changed to match. With annotations, we can have Dafny prove that the property we claim of the method is true. There are several ways to give annotations, but some of the most common, and most basic, are method *pre-* and *postconditions* (see also, *e.g.*, [10,19,20,9,3,1]).

This property of the Abs method, that the result is always non-negative, is an example of a postcondition: it is something that is true after the method returns. Postconditions, declared with the **ensures** keyword, are given as part of the method's declaration, after the return values (if present) and before the method body. The keyword is followed by the boolean expression and a semicolon, which are both required. Like an **if** or **while** condition and most annotations, a postcondition is always a boolean expression: something that can be *false* or *true*. In the case of the Abs method, a reasonable postcondition is the following:

```
method Abs(x: int) returns (y: int)
   ensures 0 <= y;
{
   ...
}
```

You can see here why return values are given names. This makes them easy to refer to in the postcondition of a method. When the expression is true, we say that the postcondition *holds*. The postcondition must hold for every possible invocation of the function, and for every possible return point (including the implicit one at the end of the function body). In this case, the only property we are expressing is that the return value is always at least zero.

Sometimes there are multiple properties that we would like to establish about our code. In this case, we have two options. We can either join the two conditions together with the boolean *and* operator (&&), or we can write multiple **ensures** annotations. The latter is the same as the former, but it helps make distinct properties clearer. For example, the return value names from the MultipleReturns method might lead one to guess the following postconditions:

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

The postcondition can also be written:

```
   ensures less < x && x < more;
```

or even:

```
   ensures less < x < more;
```

because of the *chaining* comparison operator syntax in Dafny. (In general, most of the comparison operators can be chained, but only "in one direction", *i.e.*, not mixing greater than and less than; see Section A.8.)

The first way of expressing the postconditions separates the "less" part from the "more" part, which may be desirable. Another thing to note is that we have included one of the input parameters in the postcondition. This is useful because it allows us to relate the input and output of the method to one another (this works because input parameters are read only, and so are the same at the end as they were at the beginning).

Dafny actually rejects this program, claiming that the first postcondition does not hold (*i.e.*, is not true). This means that Dafny wasn't able to prove that this annotation holds every time the method returns. In general, there are two main causes for Dafny verification errors: annotations that are inconsistent with the code, and situations where it is not "clever" enough to prove the required properties. Differentiating between these two possibilities can be a difficult task, but fortunately, Dafny and the Boogie/Z3 system [0,18,5] on which it is based [14] are pretty smart, and will prove matching code and annotations with a minimum of fuss.

In this situation, Dafny is correct in saying there is an error with the code. The key to the problem is that `y` is an integer, so it can be negative. If `y` is negative (or zero), then `more` can actually be smaller than or equal to `x`. Our method will not work as intended unless `y` is strictly larger than zero. This is precisely the idea of a *precondition*. A precondition is similar to a postcondition, except that it is something that must be true *before* a method is called. When you call a method, it is your job to establish (make true) the preconditions, something Dafny will enforce using a proof. Likewise, when you write a method, you get to assume the preconditions, but you must establish the postconditions. The caller of the method then gets to assume that the postconditions hold after the method returns.

Preconditions have their own keyword, **`requires`**. We can give the necessary precondition to `MultipleReturns` as below:

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

Like postconditions, multiple preconditions can be written either with the boolean *and* operator (`&&`), or by multiple **`requires`** keywords. Traditionally, **`requires`** annotations precede **`ensures`** annotations in the source code, though this is not strictly necessary. (The order of the **`requires`** and **`ensures`** annotations with respect to others of the same type can sometimes matter, as we will see later.) With the addition of this condition, Dafny now verifies the code as correct, because this assumption is all that is needed to guarantee the code in the method body is correct.

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

Not all methods necessarily have preconditions. For example, the `Abs` method we have already seen is defined for all integers, and so has no preconditions (other than the trivial requirement that its argument is an integer, which is enforced by the type system). Even though it has no need of preconditions, the `Abs` function as it stands now is not very useful. To investigate why, we need to make use of another kind of annotation, the *assertion*.

3. Assertions

Unlike pre- and postconditions, an assertion is placed somewhere in the middle of a method. Like the previous two annotations, an assertion has a keyword, **`assert`**, followed by the boolean expression and a semicolon. An assertion says that a particular expression always holds when control reaches that part of the code. For example, the following is a trivial use of an assertion inside a dummy method:

```
method Testing()
{
   assert 2 < 3;
}
```

Dafny proves this method correct, as `2` is always less than `3`. Asserts have several uses, but chief among them is checking whether your expectations of what is true at various points is actually true. You can use this to check basic arithmetical facts, as above, but they can also be used in more complex situations. Assertions are a powerful tool for debugging annotations, by checking what Dafny is able to prove about your code. For example, we can use it to investigate what Dafny knows about the `Abs` function.

To do this, we need one more concept: local variables. Local variables behave exactly as you would expect, except maybe for a few issues with shadowing. Local variables are declared with the **`var`** keyword, and can optionally have type declarations. Unlike method parameters, where types are required, Dafny can infer the types of local variables in almost all situations. This is an example of an initialized, explicitly typed variable declaration:

```
   var x: int := 5;
```

The type annotation can be dropped in this case:

```
   var x := 5;
```

Multiple variables can be declared at once:

```
   var x, y, z: bool := 1, 2, true;
```

Explicit type declarations apply only to the immediately preceding variable, so here the **`bool`** declaration applies only to `z`, and not `x` or `y`, which are both inferred to be of type **`int`**. We needed variables because we want to talk about the return value of the `Abs` method. We cannot put `Abs` inside a annotations directly, as the method could change memory state, among other problems. So we capture the return value of a call to Abs as follows:

```
// use definition of Abs() from before.
method Testing()
{
   var v := Abs(3);
   assert 0 <= v;
}
```

This is an example of a situation where we can ask Dafny what it knows about the values in the code, in this case v. We do this by adding assertions, like the one above. Every time Dafny encounters an assertion, it tries to prove that the condition holds for all executions of the code. In this example, there is only one control path through the method, and Dafny is able to prove the annotation easily because it is exactly the postcondition of the Abs method. Abs guarantees that the return value is non-negative, so it trivially follows that v, which is this value, is non-negative after the call to Abs.

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

But we know something stronger about the Abs method. In particular, for non-negative x, `Abs(x) == x`. Specifically, in the above program, the value of v is 3. If we try adding an assertion (or changing the existing one) to say:

```
   assert v == 3;
```

we find that Dafny cannot prove our assertion, and gives an error. The reason this happens is that Dafny "forgets" about the body of every method except the one it is currently working on. This simplifies Dafny's job tremendously, and is one of the reasons it is able to operate at reasonable speeds. It also helps us reason about our programs by breaking them apart and so we can analyze each method in isolation (given the annotations for the other methods). We don't care at all what happens inside each method when we call it, as long as it satisfies its annotations. This works because Dafny will prove that all the methods satisfy their annotations, and refuse to compile our code until they do.

For the Abs method, this means that the only thing Dafny knows in the Testing method about the value returned from Abs is what the postconditions say about it, and *nothing more*. This means that Dafny won't know the nice property about Abs and non-negative integers unless we tell it by putting this in the postcondition of the Abs method. Another way to look at it is to consider the method annotations (along with the type of the parameters and return values) as fixing the behavior of the method. Everywhere the method is used, we assume that it is any conceivable method that satisfies the pre- and postconditions. In the Abs case, we might have written:

```
method Abs(x: int) returns (y: int)
   ensures 0 <= y;
{
   y := 0;
}
```

This method satisfies the postconditions, but clearly the program fragment:

```
var v := Abs(3);
assert v == 3;
```

would not be true in this case. Dafny is considering, in an abstract way, all methods with those annotations. The mathematical absolute value certainly is such a method, but so are all methods that return a positive constant, for example. We need stronger postconditions to eliminate these other possibilities, and “fix” the method down to exactly the one we want. We can partially do this with the following:

```
method Abs(x: int) returns (y: int)
   ensures 0 <= y;
   ensures 0 <= x ==> x == y;
{
   if (x < 0) { y := -x; }
         else { y := x; }
}
```

This expresses exactly the property we discussed before, that the absolute value is the same for non-negative integers. The second **ensures** uses implication operator, `==>`, which basically says that the left-hand side implies the right in the mathematical sense (it binds more weakly than boolean *and* and comparisons, so the above says `0 <= x` implies `x == y`). The left and right sides must both be boolean expressions.

The postcondition says that after `Abs` is called, if the value of `x` was non-negative, then `y` is equal to `x`. One caveat of the implication is that it is still true if the left part (the antecedent) is false. So the second postcondition is trivially true when `x` is negative. In fact, the only thing that the annotations say when `x` is negative is that the result, `y`, is non-negative. But this is still not enough to fix the function, so we must add another postcondition, to make the following complete annotation covering all cases:

```
method Abs(x: int) returns (y: int)
   ensures 0 <= y;
   ensures 0 <= x ==> y == x;
   ensures x < 0 ==> y == -x;
{
   if (x < 0) { y := -x; }
         else { y := x; }
}
```

These annotations are enough to require that our method actually computes the absolute value of `x`. But we still have an issue: there seems to be a lot of duplication. The body of the method is reflected very closely in the annotations. While this is correct code, we want to eliminate this redundancy. As you might guess, Dafny provides a means of doing this: functions.

Exercise 2 *Using a precondition, change `Abs` to say it can only be called on negative values. Simplify the body of `Abs` into just one assignment statement and make sure the method still verifies.*

Exercise 3 *Keeping the postconditions of* `Abs` *the same as above, change the body of* `Abs` *to* `y := x + 2;`*. What precondition do you need to annotate the method with in order for the verification to go through? And what precondition do you need if the body is* `y := x + 1;`*? What does that precondition say about calling the method?*

4. Functions

A function in Dafny closely follows the concept of a mathematical function. Unlike methods, a Dafny function cannot write to memory, and consists solely of one expression. They are required to have a single return value, which is unnamed. The declaration looks similar to that of a method:

```
function abs(x: int): int
{
   ...
}
```

This declares a function called `abs` which takes a single integer, and returns an integer (the second **`int`**). 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 type. Here our body must be an integer expression. In order to implement the absolute value function, we need to use an *if expression*. An if expression behaves like the ternary operator in other languages.

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

Obviously, the condition must be a boolean expression, and the two branches must have the same type. You might wonder why anyone would bother with functions, if they are so limited compared to methods. The power of functions comes from the fact that they can be *used directly in annotations*. So we can write:

```
   assert abs(3) == 3;
```

In fact, not only can we write this statement directly without capturing to a local variable, we didn't even need to write all the postconditions that we did with the method (though functions can and do have pre- and postconditions in general). The limitations of functions are precisely what let Dafny do this. Unlike methods, Dafny does not forget the body of a function when considering other functions. So it can expand the definition of `abs` in the above assertion and determine that the result is actually 3.

Exercise 4 *Write a* function `max` *that returns the larger of two given integer parameters. Write a test method that uses an* **`assert`** *statement to check some property of the value of* `max` *on some arguments (for example,* `34 < max(21,55)` *or* `max(21,55) == max(55,21)`*).*

One caveat of functions is that not only can they appear in annotations, they can only appear in annotations. One cannot write:

```
var v := abs(3);
```

as this is not an annotation. Functions are never part of the final compiled program, they are just tools to help us verify our code. Nevertheless, sometimes it is convenient to use a function in real code too, so one can define a **function method**, which *can* be called from real code. Note that there are restrictions on what functions can be function methods.

Exercise 5 *Change the test method in Exercise 4 to put the result of* `max` *into a local variable and then use an* **assert** *statement to check some property of that local variable. Dafny will reject this program, because you're calling* `max` *from real code. Fix this problem using a* **function method***.*

Exercise 6 *Now that we have an* `abs` *function, change the postcondition of method* `Abs` *(at the end of Section 3) to make use of* `abs`*. After you make sure the method still verifies, change the body of* `Abs` *to also use* `abs`*. (After doing this, you will also realize there's not much point in having a method that does exactly the same thing as a function method.)*

Unlike methods, functions can appear in expressions. Thus we can do something like implement the mathematical 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)
}
```

Here we use **nat**, the type of natural numbers (non-negative integers), which is often more convenient than annotating everything to be non-negative. It turns out that we could make this function a function method if we wanted to. But executing that function method would be extremely slow, as this version of calculating the Fibonacci numbers has exponential complexity. There are much better ways to calculate the Fibonacci function. But this function is still useful, as we can have Dafny prove that our fast version really matches the mathematical definition. We can get the best of both worlds: the guarantee of correctness and the performance we want.

We can start by defining a method like the following:

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

We haven't written the body yet, so Dafny will complain that our post condition doesn't hold. We need an algorithm to calculate the nth Fibonacci number. The basic idea is to keep a counter, and repeatedly calculate adjacent pairs of Fibonacci numbers until the desired number is reached. To do this, we need a loop. In Dafny, this is done via a **while** loop:

```
var i := 0;
while (i < n)
{
   i := i + 1;
}
```

This is a trivial loop that just increments `i` until it reaches `n`. This will form the core of our loop to calculate Fibonacci numbers.

5. Loop Invariants

Loops present a problem for Dafny. There is no way for Dafny to know in advance how many times the code will go around the loop. But Dafny needs to consider all paths through a program, which could include going around the loop any number of times. To make it possible for Dafny to work with loops, you need to provide *loop invariants*, another kind of annotation.

A loop invariant is an expression that holds upon entering a loop, and after every execution of the loop body. It captures something that is invariant, *i.e.*, does not change, about every step of the loop. Now, obviously we are going to want to change variables, etc. each time around the loop, or we wouldn't need the loop. Like pre- and postconditions, an invariant is a *property* that is preserved for each execution of the loop, expressed using the same boolean expressions we have seen. For example, we see in the above loop that if `i` starts off positive, then it stays positive. So we can add the invariant, using its own keyword, to the loop:

```
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. By preserved, we mean that assuming that the invariant holds at the beginning of the loop, we must show that executing the loop body once makes the invariant hold again. Dafny can only know upon analyzing the loop body what the invariants say, in addition to the loop guard (the loop condition). Just as Dafny will not discover properties of a method on its own, it will not know any but the most basic properties of a loop are preserved unless it is told via an invariant.

In our example, the point of the loop is to build up the Fibonacci numbers one (well, two) at a time until we reach the desired number. After we exit the loop, we will have that `i == n`, because `i` will stop being incremented when it reaches `n`. We can use our assertion trick to check to see if Dafny sees this fact as well:

```
var i := 0;
while (i < n)
   invariant 0 <= i;
{
   i := i + 1;
}
assert i == n;
```

We find that this assertion fails. As far as Dafny knows, it is possible that `i` somehow became much larger than `n` at some point during the loop. All it knows after the loop exits (*i.e.*, in the code after the loop) is that the loop guard failed, and the invariants hold. In this case, this amounts to `n <= i` and `0 <= i`. But this is not enough to guarantee that `i == n`, just that `n <= i`. Somehow we need to eliminate the possibility of `i` exceeding `n`. One first guess for solving this problem might be the following:

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

This does not verify, as Dafny complains that the invariant is not preserved (also known as maintained) by the loop. We want to be able to say that after the loop exits, then all the invariants hold. Our invariant holds for every execution of the loop *except* for the very last one. Because the loop body is executed only when the loop guard holds, in the last iteration `i` goes from `n - 1` to `n`, but does not increase further, at the loop exits. Thus, we have only omitted exactly one case from our invariant, and repairing it relatively easy:

```
...
   invariant 0 <= i <= n;
...
```

Now we can say both that `n <= i` from the loop guard and `0 <= i <= n` from the invariant, which allows Dafny to prove the assertion `i == n`. The challenge is finding a loop invariant that is preserved by the loop, but also that lets you prove what you need after the loop has executed.

Exercise 7 *Change the loop invariant to `0 <= i <= n+2`. Does the loop still verify? Does the assertion `i == n` after the loop still verify?*

Exercise 8 *Once again using the invariant `0 <= i <= n` for the loop above, change the loop guard from `i < n` to `i != n`. Do the loop and the assertion after the loop still verify?*

In addition to the counter, our algorithm called for a pair of numbers which represent adjacent Fibonacci numbers in the sequence. Unsurprisingly, we will have another invariant or two to relate these numbers to each other and the counter. To find these invariants, we employ a common Dafny trick: working backwards from the postconditions.

Our postcondition for the Fibonacci method is that the return value `b` is equal to `fib(n)`. But after the loop, we have that `i == n`, so we need `b == fib(i)` at the end of the loop. This might make a good invariant, as it relates something to the loop counter. This observation is surprisingly common throughout Dafny programs. Many times a method is just a loop that, when it ends, makes the postcondition true by having a counter reach another number, often an argument or the length of an array or sequence. So we have that the variable `b`, which is conveniently our out parameter, will be the current Fibonacci number:

```
invariant b == fib(i);
```

We also note that in our algorithm, we can compute any Fibonacci number by keeping track of a pair of numbers, and summing them to get the next number. So we want a way of tracking the previous Fibonacci number, which we will call `a`. Another invariant will express that number's relation to the loop counter. The invariants are:

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

At each step of the loop, the two values are summed to get the next leading number, while the trailing number is the old leading number. Using a parallel assignment, we can write a loop that performs this operation:

```
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;
}
```

Here `a` is the trailing number, and `b` is the leading number. The parallel assignment means that the entire right-hand side is calculated before the assignments to the variables are made. Thus `a` will get the old value of `b`, and `b` will get the sum of the two old values, which is precisely the behavior we want.

We also have made a change to the loop counter `i`. Because we also want to track the trailing number, we can't start the counter at `0`, as otherwise we would have to calculate a negative Fibonacci number. The problem with doing this is that the loop counter invariant may not hold when we enter the loop. The only problem is when `n` is `0`. This can be eliminated as a special case, by testing for this condition at the beginning of the loop. The completed Fibonacci method becomes:

```
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;
   }
}
```

Dafny no longer complains about the loop invariant not holding, because if `n` were `0`, it would return before reaching the loop. Dafny is also able to use the loop invariants to prove that after the loop, `i == n` and `b == fib(i)`, which together imply the postcondition, `b == fib(n)`.

Exercise 9 *The `ComputeFib` method above is more complicated than necessary. Write a simpler program by not introducing `a` as the Fibonacci number that precedes `b`, but instead introducing a variable `c` that that succeeds `b`. Verify that your program is correct.*

Exercise 10 *Starting with the final `ComputeFib` method above, delete the **`if`** statement and change the initializations to start `i` at `0`, `a` at `1`, and `b` at `0`. Verify this new program by adjusting the loop invariants to match the new behavior.*

One of the problems with using invariants is that it is easy to forget to have the loop *make progress*, *i.e.*, do work at each step. For example, we could have omitted the entire body of the loop in the previous program. The invariants would be correct, because they are still true upon entering the loop, and since the loop doesn't change anything, they would be preserved by the loop. But the crucial step from the loop to the postcondition wouldn't hold. We know that *if* we exit the loop, then we can assume the negation of the guard and the invariants, but this says nothing about what happens if we never exit the loop. Thus we would like to make sure the loop ends at some point, which gives us a much stronger correctness guarantee.

6. Termination

Dafny proves code terminates, *i.e.*, does not loop forever, by using **`decreases`** annotations. For many things, Dafny is able to guess the right annotations, but sometimes it needs to be made explicit. In fact, for all of the code we have seen so far, Dafny has been able to do this proof on its own, which is why we haven't needed the **`decreases`** annotation explicitly yet. There are two places Dafny proves termination: loops and recursion. Both of these situations require either an explicit annotation or a correct guess by Dafny.

A **decreases** annotation, as its name suggests, gives Dafny an expression that decreases with every loop iteration or recursive call. There are two conditions that Dafny needs to verify when using a **decreases** expression: that the expression actually gets smaller, and that it is bounded. Many times, an integral value (natural or plain integer) is the quantity that decreases, but other things can be used as well. In the case of integers, the bound is assumed to be zero. For example, the following is a proper use of a **decreases** annotation on a loop:

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

Here Dafny has all the ingredients it needs to prove termination. The variable `i` gets smaller each loop iteration, and is bounded below by zero. This is fine, except the loop is backwards from most loops, which tend to count up instead of down. In this case, what decreases is not the counter itself, but rather the distance between the counter and the upper bound. A simple trick for dealing with this situation is given below:

```
while (i < n)
   invariant 0 <= i <= n;
   decreases n - i;
{
   i := i + 1;
}
```

This is actually Dafny's guess for this situation, as it sees `i < n` and assumes that `n - i` is the quantity that decreases. The upper bound of the loop invariant implies that `0 <= n - i`, and gives Dafny a lower bound on the quantity. This also works when the bound `n` is not constant, such as in the binary search algorithm, where two quantities approach each other.

Exercise 11 *In the loop above, the invariant* `i <= n` *and the negation of the loop guard allow us to conclude* `i == n` *after the loop (as we checked in Section 5 by writing an* **`assert`** *statement). Note that if the loop guard were instead written as* `i != n` *(as in Exercise 8), then the negation of the guard immediately gives* `i == n` *after the loop, regardless of the loop invariant. Change the loop guard to* `i != n` *and delete the invariant annotation. Does the program verify?*

The other situation that requires a termination proof is when methods or functions are recursive. Similarly to looping forever, these methods could potentially call themselves forever, never returning to their original caller. When Dafny is not able to guess the termination condition, an explicit **decreases** clause can be given along with pre- and postconditions, as in the unnecessary annotation for the `fib` function:

```
function fib(n: nat): nat
   decreases n;
{
   ...
}
```

As before, Dafny can guess this condition on its own, but sometimes the decreasing condition is hidden within a field of an object where Dafny cannot find it on its own, and then it requires an explicit annotation.

7. Arrays

All that we have considered is fine for toy functions and little mathematical exercises, but it really isn't helpful for real programs. So far we have only considered a handful of values at a time in local variables. Now we turn our attention to arrays of data. Arrays are a built in part of the language, with their own type, **array**<T>, where T is another type. For now we only consider arrays of integers, **array**<**int**>. Arrays can be **null**, and have a built in length field called Length. Element access uses the standard bracket syntax and are indexed from zero, so a[3] is preceded by the 3 elements a[0], a[1], and a[2], in that order. All array accesses must be proven to be within bounds, which is part of the no-runtime-errors safety guaranteed by Dafny. Because bounds checks are proven at verification time, no runtime checks need to be made. To create a new array, it must be allocated with the **new** keyword, but for now we will only work with methods that take a previously allocated array as an argument.

One of the most basic things we might want to do with an array is search through it for a particular key, and return the index of a place where we can find the key if it exists. We have two outcomes for a search, with a different correctness condition for each. If the algorithm returns an index (*i.e.*, non-negative integer), then the key should be present at that index. This might be expressed as follows:

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

The array index here is safe because the implication operator is *short circuiting*. Short circuiting means if the left part is false, then the implication is already true regardless of the truth value of the second part, and thus it does not need to be evaluated. Using the short circuiting property of the implication operator, along with the boolean *and* (&&), which is also short circuiting, is a common Dafny practice. The condition index < a.Length is necessary because otherwise the method could return a large integer which is not an index into the array. Together, the short circuiting behavior means that by the time control reaches the array access, index must be a valid index.

If the key is not in the array, then we would like the method to return a negative number. In this case, we want to say that the method did not miss an occurrence of the key; in other words, that the key is not in the array. To express this property, we turn to another common Dafny tool: quantifiers.

8. Quantifiers

A quantifier in Dafny most often takes the form of a **forall** expression, also called a universal quantifier. As its name suggests, this expression is true if some property holds for all elements of some set. For now, we will consider the set of integers. An example universal quantifier, wrapped in an assertion, is given below:

```
assert forall k :: k < k + 1;
```

A quantifier introduces a temporary name for each element of the set it is considering. This is called the bound variable, in this case `k`. The identifier has a type, which is almost always inferred rather than given explicitly and is usually **int** anyway. (In general, one can have any number of bound variables; we will encounter an example in Section 9.) A pair of colons (`::`) separates the bound variable and its optional type from the quantified property (which must be of type **bool**). In this case, the property is that adding `1` to any integer makes a strictly larger integer. Dafny is able to prove this simple property automatically. Generally it is not very useful to quantify over infinite sets, such as all the integers. Instead, quantifiers are typically used to quantify over all elements in an array or data structure. We do this for arrays by using the implication operator to make the quantified property trivially true for values which are not indices:

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

This says that some property holds for each element of the array. The implication makes sure that `k` is actually a valid index into the array before evaluating the second part of the expression. Dafny can use this fact not only to prove that the array is accessed safely, but also to reduce the set of integers it must consider to only those that are indices into the array.

With a quantifier, saying the key is not in the array is straightforward:

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

Thus our method becomes (with the addition of the non-nullity requirement on `a`):

```
method Find(a: array<int>, key: int) returns (index: int)
   requires a != null;
   ensures 0 <= index ==> index < a.Length && a[index] == key;
   ensures index < 0 ==> forall k :: 0 <= k < a.Length ==> a[k] != key;
{
   ...
}
```

We can fill in the body of this method in a number of ways, but perhaps the easiest is a linear search, implemented below:

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

As you can see, we have omitted the loop invariants on the while loop, so Dafny gives us a verification error on one of the postconditions. The reason we get an error is that Dafny does not know that the loop actually covers all the elements. In order to convince Dafny of this, we have to write an invariant that says that everything before the current index has already been looked at (and is not the key). Just like the postcondition, we can use a quantifier to express this property:

```
invariant forall k :: 0 <= k < index ==> a[k] != key;
```

This says that everything before, but excluding, the current index is not the key. Notice that upon entering the loop, `i` is `0`, so the first part of the implication is always false, and thus the quantified property is always true. This common situation is known as *vacuous truth*: the quantifier holds because it is quantifying over an empty set of objects. This means that it is true when entering the loop. We test the value of the array before we extend the non-key part of the array, so Dafny can prove that this invariant is preserved. One problem arises when we try to add this invariant: Dafny complains about the index being out of range for the array access within the invariant.

This is erroneous because there is no invariant on `index`, so it could be greater than the length of the array. Then the bound variable, `k`, could exceed the length of the array. To fix this, we put the standard bounds on `index`, `0 <= index <= a.Length`. Note that because we say `k < index`, the array access is still protected from error even when `index == a.Length`. The use of a variable that is one past the end of a growing range is a common pattern when working with arrays, where it is often used to build a property up one element at a time. The complete method is given below:

```
method Find(a: array<int>, key: int) returns (index: int)
   requires a != null;
   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;
}
```

Exercise 12 *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 loop invariants to verify it.*

A linear search is not very efficient, especially when many queries are made of the same data. If the array is sorted, then we can use the efficient binary search procedure to find the key. But in order for us to be able to prove our implementation correct, we need some way to require that the input array actually is sorted. We could do this directly with a quantifier inside a **`requires`** clause of our method, but a more modular way to express this is through a *predicate*.

9. Predicates

A predicate is a function that returns a boolean. It is a simple but powerful idea that occurs throughout Dafny programs. For example, preparing the way for the binary search method in Section 11, we define the sorted predicate over arrays of integers as a function that takes an array as an argument, and returns true if that array is sorted in increasing order. The use of predicates makes our code shorter, as we do not need to write out a long property over and over. It can also make our code easier to read by giving a common property a name.

There are a number of ways we could write the sorted predicate, but the easiest is to use a quantifier over the indices of the array. We can write a quantifier that expresses the property, "if `x` is before `y` in the array, then `x <= y`", as a quantifier over two bound variables:

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

Here we have two bound variables, `j` and `k`, which are both integers. The comparisons between the two guarantee that they are both valid indices into the array, and that `j` is before `k`. Then the second part says that they are ordered properly with respect to one another. The function, along with the standard non-null precondition required for using the array, is thus:

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

Dafny rejects this code as given, claiming that the function cannot read the elements of `a`. Dealing with this complaint requires another annotation, the *reads annotation*.

Exercise 13 *To get practice with quantifiers, modify the definition of function* `sorted` *so that it returns true only when the array is sorted and all its elements are distinct.*

10. 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 is all the memory locations that the function is allowed to read. The reason we might limit what a function can read is so that when we write to memory, we can be sure that functions that do not read that part of memory yield the same value after the write as they did before the write. For example, we might have two arrays, one of which we know is sorted. If we did not put a reads annotation on the sorted predicate, then when we modify the unsorted array, we cannot immediately determine whether the other array stopped being sorted. While we might be able to give invariants to preserve it in this case, it gets even more complex when manipulating data structures and dealing with recursive functions. In those cases, framing is essential to making the verification process feasible.

To specify a reading frame, we use a reads annotation, again with its own keyword:

```
function sorted(a: array<int>): bool
   ...
   reads a;
   ...
```

A **reads** annotation can appear anywhere along with the pre- and postconditions, but it does not give a boolean expression like the other annotations we have seen; instead, it specifies a set of memory locations that the function is allowed to access. The name of an array, like `a` in the above example, stands for all the elements of that array. One can also specify object fields and sets of objects, but we will not concern ourselves with those topics here. Dafny will check that you do not read any memory location that is not stated in the reading frame. This means that function calls within a function must have reading frames that are a subset of the calling function's reading frame. One thing to note is that parameters to the function are not memory locations, so do not need to be declared.

Frames also affect methods. As you might have guessed, methods are not required to list the things they read, as we have written a method which accesses an array with no reads annotation. Methods are allowed to read whatever memory they like, but they are required to list which parts of memory they modify, with a *modifies annotation*. They are almost identical to their reads cousins, except they say what can be changed. In combination with reads, modification restrictions allow Dafny to prove properties of code that would otherwise be very difficult or impossible. Reads and modifies are two of the tools that allow Dafny to work on one method at a time, because they restrict what would otherwise be arbitrary modifications of memory to something that Dafny can reason about.

11. Binary Search

Predicates are used to make other annotations clearer:

```
method BinarySearch(a: array<int>, key: int) returns (index: int)
   requires a != null && sorted(a);
   ensures 0 <= index ==> index < a.Length && a[index] == key;
```

```
    ensures index < 0 ==> forall k :: 0 <= k < a.Length ==> a[k] != key;
{
    ...
}
```

Exercise 14 *What happens if you remove the precondition* `a != null`*? Change the definition of* `sorted` *so that it allows its argument to be null but returns false if it is.*

We have the same postconditions that we did for the linear search, as the goal is the same. The difference is that now we know the array is sorted. Because Dafny can unwrap functions, inside the body of the method it knows this too. We can then use that property to prove the correctness of the search. The method body is given below:

```
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] != key;
{
   var mid := (low + high) / 2;
   if (a[mid] < key) {
      low := mid + 1;
   } else if (key < a[mid]) {
      high := mid;
   } else {
      index := mid;
      return;
   }
}
index := -1;
```

First we declare our range to search over. This can be thought of as the remaining space where the key could possibly be. The first invariant expresses the fact that this range is within the array. The second says that the key is not anywhere outside of this range. In the first two branches of the **if** chain, we find the element in the middle of our range is not the key, and so we move the range to exclude that element and all the other elements on the appropriate side of it. We need the addition of `1` when moving the lower end of the range because it is inclusive. If we do not add this, then the loop may continue forever when `mid == low`, which happens when `low + 1 == high`. We could change this to say that the loop exits when `low` and `high` are one apart, but this would mean we would need an extra check after the loop to determine if the key was found at the one remaining index. In the above formulation, this is unnecessary because when `low == high`, the loop exits. But this means that no elements are left in the search range, so the key was not present. This can be deduced from the loop invariant:

```
invariant forall i :: 0 <= i < a.Length && !(low <= i < high)
                   ==> a[i] != key;
```

When `low == high`, the negated condition in the first part of the implication is always true (because no `i` can be both at least and strictly smaller than the same value). Thus the invariant says that all elements in the array are not the key, and the second postcondition holds. As you can see, it is easy to introduce subtle off-by-one errors in this code. With the invariants, not only can Dafny prove the code correct, but we can understand the operation of the code easier ourselves.

Exercise 15 *Change the assignments in the body of `BinarySearch` to set `low` to `mid` or to set `high` to `mid - 1`. In each case, what goes wrong?*

12. Further Study

For advanced uses of specification features in Dafny or other tools, we recommend studying textbooks on program correctness (such as [6,8,7,11]), collections of verified programs (such as [23,4]), or, as have become more prevalent lately, the solutions submitted to verification challenges ([22,16,17]) and verification competitions ([13]).

There are several techniques for specifying programs with dynamic data structures (for a survey, see [10]). For this purpose, Dafny supports a specification idiom based on *dynamic frames* [12,21]. That idiom goes beyond what we have covered in these lecture notes, but see [15,14].

13. Conclusion

We've seen a whirlwind tour of the major features of Dafny, and used it for some interesting, if a little on the small side, examples of what Dafny can do. Even if you do not use Dafny regularly, writing down what it is that your code does, in a precise way, and using this to prove code correct is a useful skill to have. Invariants, pre- and postconditions, and annotations are useful in debugging code, and also as documentation for future developers. When modifying or adding to a codebase, they confirm that the guarantees of existing code are not broken. If annotations are checked at run time, they act as test oracles injected in the code. They also ensure that APIs are used correctly, by formalizing behavior and requirements and enforcing correct usage. Reasoning from invariants, considering pre- and postconditions, and writing assertions to check assumptions are all general computer science skills that will benefit you no matter what language you work in.

References

[0] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, September 2006.

[1] Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. Specification and verification: the Spec# experience. *Communications of the ACM*, 54(6):81–91, June 2011.

[2] Raymond T. Boute. The Euclidian definition of the functions div and mod. *ACM Transactions on Programming Languages and Systems*, 14(2):127–144, April 1992.

[3] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joeseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, June 2005.

[4] Dafny test suite. Files with extension `.dfy` in the subdirectories of `Test` at `boogie.codeplex.com`.

[5] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, March–April 2008.

[6] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.

[7] Edsger W. Dijkstra and W. H. J. Feijen. *A Method of Programming*. Addison-Wesley, July 1988.

[8] David Gries. *The Science of Programming*. Texts and Monographs in Computer Science. Springer-Verlag, 1981.

[9] John V. Guttag and James J. Horning, editors. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer, 1993. With Stephen J. Garland, Kevin D. Jones, Andrés Modet, and Jeannette M. Wing.

[10] John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew Parkinson. Behavioral interface specification languages. Technical Report CS-TR-09-01a, Department of Electrical Engineering and Computer Science, University of Central Florida, October 2010. To appear in *ACM Computing Surveys*.

[11] Anne Kaldewaij. *Programming: The Derivation of Algorithms*. Series in Computer Science. Prentice-Hall International, 1990.

[12] Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods, 14th International Symposium on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 268–283. Springer, August 2006.

[13] Vladimir Klebanov, Peter Müller, Natarajan Shankar, Gary T. Leavens, Valentin Wüstholz, Eyad Alkassar, Rob Arthan, Derek Bronish, Rod Chapman, Ernie Cohen, Mark Hillebrand, Bart Jacobs, K. Rustan M. Leino, Rosemary Monahan, Frank Piessens, Nadia Polikarpova, Tom Ridge, Jan Smans, Stephan Tobies, Thomas Tuerk, Mattias Ulbrich, and Benjamin Weiß. The 1st verified software competition: Experience report. In Michael Butler and Wolfram Schulte, editors, *FM 2011: Formal Methods - 17th International Symposium on Formal Methods*, volume 6664 of *Lecture Notes in Computer Science*, pages 154–168. Springer, June 2011.

[14] K. Rustan M. Leino. Specification and verification of object-oriented software. In Manfred Broy, Wassiou Sitou, and Tony Hoare, editors, *Engineering Methods and Tools for Software Safety and Security*, volume 22 of *NATO Science for Peace and Security Series D: Information and Communication Security*, pages 231–266. IOS Press, 2009. Summer School Marktoberdorf 2008 lecture notes.

[15] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *LPAR-16*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, April 2010.

[16] K. Rustan M. Leino and Rosemary Monahan. Dafny meets the verification benchmarks challenge. In Gary T. Leavens, Peter W. O'Hearn, and Sriram K. Rajamani, editors, *Verified Software: Theories, Tools, Experiments, Third International Conference, VSTTE 2010*, volume 6217 of *Lecture Notes in Computer Science*, pages 112–126. Springer, August 2010.

[17] K. Rustan M. Leino and Michał Moskal. VACID-0: Verification of ample correctness of invariants of data-structures, edition 0. In *Tools and Experiments workshop at VSTTE 2010*, August 2010.

[18] K. Rustan M. Leino and Philipp Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010*, volume 6015 of *Lecture Notes in Computer Science*, pages 312–327. Springer, March 2010.

[19] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. MIT Electrical Engineering and Computer Science Series. MIT Press, 1986.

[20] Bertrand Meyer. *Object-oriented Software Construction*. Series in Computer Science. Prentice-Hall International, 1988.

[21] Jan Smans, Bart Jacobs, Frank Piessens, and Wolfram Schulte. Automatic verifier for Java-like programs

based on dynamic frames. In José Luiz Fiadeiro and Paola Inverardi, editors, *Fundamental Approaches to Software Engineering, 11th International Conference, FASE 2008*, volume 4961 of *Lecture Notes in Computer Science*, pages 261–275. Springer, March–April 2008.

[22] Bruce W. Weide, Murali Sitaraman, Heather K. Harton, Bruce Adcock, Paolo Bucci, Derek Bronish, Wayne D. Heym, Jason Kirschenbaum, and David Frazier. Incremental benchmarks for software verification tools and techniques. In Natarajan Shankar and Jim Woodcock, editors, *Verified Software: Theories, Tools, Experiments, Second International Conference, VSTTE 2008*, volume 5295 of *Lecture Notes in Computer Science*, pages 84–98. Springer, October 2008.

[23] Why3 gallery of certified programs. `proval.lri.fr/gallery/why3.en.html`.

A. Dafny Quick Reference

This appendix illustrates many of the most common language features in Dafny.

A.0. Programs

At the top level, a Dafny program (stored as a file with extension `.dfy`) is a set of declarations. The declarations introduce fields, methods, and functions, as well as classes and inductive datatypes, where the order of introduction is irrelevant. A class also contains a set of declarations, introducing fields, methods, and functions. Fields, methods, and functions declared outside a class go into an implicit class called `_default`, giving the appearance of the program having global variables, procedures, and functions. If the program contains a unique parameter-less method called `Main`, then program execution starts there, but it is not necessary to have a main method to do verification.

Comments start with `//` and go to the end of the line, or start with `/*` and end with `*/` and can be nested.

A.1. Fields

A field `x` of some type `T` is declared as:

```
var x: T;
```

Unlike for local variables and bound variables, the type is required and will not be inferred. The field can be declared to be a *ghost* field by preceding the declaration with the keyword **`ghost`**. Ghost entities (fields, variables, functions, methods, code) are not represented at run time; they are only used by the verifier, which treats them as it treats non-ghost entities.

Dafny's types include **`bool`** for booleans, **`int`** for mathematical (that is, unbounded) integers, user-defined classes and inductive datatypes, **`set`**<T> for finite mathematical (immutable) sets of `T` values (where `T` is any type), and **`seq`**<T> for mathematical (immutable) sequences of `T` values. In addition, there are array types (which are like predefined "class" types) of one and more dimensions, written **`array`**<T>, **`array2`**<T>, **`array3`**<T>, ... The type **`object`** is a supertype of all class types, that is, an **`object`** denotes any reference, including **`null`**. Finally, the type **`nat`** denotes a subrange of **`int`**, namely the non-negative integers.

A.2. Methods

A method declaration has the form:

```
method M(a: A, b: B, c: C) returns (x: X, y: Y, z: Y)
   requires Pre;
   modifies Frame;
   ensures Post;
   decreases Rank;
{
   Body
}
```

where a, b, c are the method's in-parameters, x, y, z are the method's out-parameters, Pre is a boolean expression denoting the method's precondition, Frame denotes a set of objects whose fields may be updated by the method, Post is a boolean expression denoting the method's postcondition, Rank is the method's variant function, and Body is a statement that implements the method. Frame can be a list of expressions, each of which is a set of objects or a single object, the latter standing for the singleton set consisting of that one object. The method's frame is the union of these sets, plus the set of objects allocated by the method body. If omitted, the pre- and postconditions default to **true** and Frame defaults to the empty set. The variant function is a list of expressions, denoting the unending lexicographic tuple consisting of the given expressions followed implicitly by "top" elements. If omitted, Dafny will guess a variant function for the method, namely the lexicographic tuple that starts with the list of the method's in-parameters.

A method can be declared as ghost by preceding the declaration with the keyword **ghost**. By default, a method has an implicit receiver parameter, **this**. This parameter can be removed by preceding the method declaration with the keyword **static**. A static method M in a class C can be invoked by C.M(...).

In a class, a method can be declared to be a constructor method by replacing the keyword **method** with the keyword **constructor**. A constructor can only be called at the time an object is allocated (see object-creation examples below), and for a class that contains one or more constructors, object creation must be done in conjunction with a call to a constructor.

A.3. Functions

A function declaration has the form:

```
function F(a: A, b: B, c: C): T
   requires Pre;
   reads Frame;
   ensures Post;
   decreases Rank;
{
   Body
}
```

where a, b, c are the method's parameters, T is the type of the function's result, Pre is a boolean expression denoting the function's precondition, Frame denotes a set of objects whose fields the function body may depend on, Post is a boolean expression denoting the function's postcondition, Rank is the function's variant function, and Body is an expression that defines the function. The precondition allows a function to be partial, that is, the precondition says when the function is defined (and Dafny will verify that every use of the function meets the precondition). The postcondition is usually not needed, since the body of the function gives the full definition. However, the postcondition can be a convenient place to declare properties of the function that may require an inductive proof to establish. For example:

```
function Factorial(n: int): int
   requires 0 <= n;
   ensures 1 <= Factorial(n);
{
   if n == 0 then 1 else Factorial(n-1) * n
}
```

says that the result of Factorial is always positive, which Dafny verifies inductively from the function body. To refer to the function's result in the postcondition, use the function itself, as shown in the example.

By default, a function is ghost, and cannot be called from non-ghost code. To make it non-ghost, replace the keyword **function** with the two keywords **function method**.

By default, a function has an implicit receiver parameter, **this**. This parameter can be removed by preceding the function declaration with the keyword **static**. A static function F in a class C can be invoked by C.F(...). This can give a convenient way to declare a number of helper functions in a separate class.

A.4. Classes

A class is defined as follows:

```
class C {
   // member declarations go here
}
```

where the members of the class (fields, methods, and functions) are defined (as described above) inside the curly braces.

A.5. Datatypes

An inductive datatype is a type whose values are created using a fixed set of constructors. A datatype Tree with constructors Empty and Node is declared as follows:

```
datatype Tree = Empty | Node(Tree, int, Tree);
```

The constructors are separated by vertical bars. Parameter-less constructors need not use parentheses, as is shown here for `Empty`.

For each constructor `Ct`, the datatype implicitly declares a boolean member `Ct?`, which returns **`true`** for those values that have been constructed using `Ct`. For example, after the code snippet:

```
var t0 := Empty;
var t1 := Node(t0, 5, t0);
```

the expression `t1.Node?` evaluates to **`true`** and `t0.Node?` evaluates to **`false`**. Two datatype values are equal if they have been created using the same constructor and equal parameters to that constructor. Therefore, for parameter-less constructors like `Empty`, `t.Empty?` gives the same result as `t == Empty`.

A constructor can optionally declare a destructor for any of its parameters, which is done by introducing a name for the parameter. For example, if `Tree` were declared as:

```
datatype Tree = Empty | Node(left: Tree, data: int, right: Tree);
```

then `t1.data == 5` and `t1.left == t0` hold after the code snippet above.

A.6. Generics

Dafny supports generic types. That is, any class, inductive datatype, method, and function can have type parameters. These are declared in angle brackets after the name of what is being declared. For example:

```
class Multiset<T> { /*...*/ }
datatype Tree<T> = Empty | Node(Tree<T>, T, Tree<T>);
method Find<T>(key: T, collection: Tree<T>) { /*...*/ }
function IfThenElse<T>(b: bool, x: T, y: T): T { /*...*/ }
```

A.7. Statements

Here are examples of the most common statements in Dafny.

```
var LocalVariables := ExprList;
Lvalues := ExprList;
assert BoolExpr;
print PrintList;
if (BoolExpr0) {
   Stmts0
} else if (BoolExpr1) {
   Stmts1
} else {
   Stmts2
}
while (BoolExpr)
```

```
   invariant Inv;
   modifies Frame;
   decreases Rank;
{
   Stmts
}
match (Expr) {
   case Empty =>  Stmts0
   case Node(l, d, r) =>  Stmts1
}
break;
return;
```

The **var** statement introduces local variables (which are not allowed to shadow other variables declared inside the same set of most tightly enclosing curly braces). Each variable can optionally be followed by `:T` for any type `T`, which explicitly gives the preceding variable the type `T` (rather than being inferred). The `ExprList` with initial values is optional. To declare the variables as ghost variables, precede the declaration with the keyword **ghost**.

The assignment statement assigns each right-hand side in `ExprList` to the corresponding left-hand side in `Lvalues`. These assignments are performed in parallel, so the left-hand sides must denote distinct L-values. Each right-hand side can be an expression or an object creation of one of the following forms:

```
new T
new T.Init(ExprList)
new T[SizeExpr]
new T[SizeExpr0, SizeExpr1]
```

The first form allocates an object of type `T`. The second form additionally invokes an initialization method or constructor `Init` on the newly allocated object. The other forms show examples of array allocations, in particular a one- and a two-dimensional array of `T` values, respectively.

The entire right-hand side of an assignment can also be a method call, in which case the left-hand sides are the actual out-parameters (omitting the "`:=`" if there are no out-parameters).

The **assert** statement claims that the given expression evaluates to **true** (which is checked by the verifier).

The **print** statement outputs to standard output the values of the given print expressions. A print expression is either an expression or a string literal (where `\n` is used to denote a newline character).

The **if** statement is the usual one. The example shows stringing together alternatives using **else if**. The **else** branch is optional, as usual.

The **while** statement is the usual loop, where the **invariant** declaration gives a loop invariant, the **modifies** clause restricts the modification frame of the loop, and the **decreases** clause introduces a variant function for the loop. By default, the loop invariant is **true**, the modification frame is the same as in the enclosing context (usually the **modifies** clause of the enclosing method), and the variant function is guessed from the loop guard.

The **match** statement evaluates the source `Expr`, an expression whose type is an inductive datatype, and then executes the case corresponding to which constructor was used to create the source datatype value, binding the constructor parameters to the given names.

The **break** statement can be used to exit loops, and the **return** statement can be used to exit a method.

A.8. Expressions

The expressions in Dafny are quite similar to those in Java-like languages. Here are some noteworthy differences.

In addition to the short-circuiting boolean operators `&&` (and) and `||` (or), Dafny has a short-circuiting implication operator `==>` and an if-and-only-if operator `<==>`. As suggested by their widths, `<==>` has lower binding power than `==>`, which in turn has lower binding power than `&&` and `||`. Implication associates to the right.

Dafny comparison expressions can be chaining, which means that comparisons "in the same direction" can be strung together. For example,

```
0 <= i < j <= a.Length == N
```

has the same meaning as:

```
0 <= i && i < j && j <= a.Length && a.Length == N
```

Note that boolean equality can be expressed using both `==` and `<==>`. There are two differences between these. First, `==` has a stronger binding power than `<==>`. Second, `==` is chaining while `<==>` is associative. That is, `a == b == c` is the same as `a == b && b == c`, whereas `a <==> b <==> c` is the same as `a <==> (b <==> c)`, which is also the same as `(a <==> b) <==> c`.

Operations on integers are the usual ones, except that `/` (integer division) and `%` (integer modulo) follow the Euclidean definition, which means that `%` always results in a non-negative number [2]. (Hence, when the first argument to `/` or `%` is negative, the result is different than what you get in C, Java, or C#.)

Dafny expressions include universal and existential quantifiers, which have the form:

```
forall BoundVariables :: Expr
```

and likewise for **exists**, where each bound variable can optionally be followed by an explicit type, as in `x: T`, and `Expr` is a boolean expression.

Operations on sets include `+` (union), `*` (intersection), and `-` (set difference), as well as the set comparison operators `<` (proper subset), `<=` (subset), their duals `>` and `>=`, and `!!` (disjointness). The expression `x in S` says that `x` is a member of set `S`, and `x !in S` is a convenient way of writing `!(x in S)`. To make a set from some elements, enclose them in curly braces. For example, `{x,y}` is the set consisting of `x` and `y` (which is a singleton set if `x == y`), `{x}` is the singleton set containing `x`, and `{}` is the empty set.

Operations on sequences include `+` (concatenation) and the comparison operators `<` (proper prefix) and `<=` (prefix). Membership can be checked like for sets: `x in S` and `x !in S`. The length of a sequence `S` is denoted `|S|`, and the elements of such a sequence have indices from `0` to less than `|S|`. The expression `S[j]` denotes the element at index `j` of sequence `S`. The expression `S[m..n]`, where `0 <= m <= n <= |S|`, returns a sequence whose elements are the `n-m` elements of `S` starting at index `m` (that is, from `S[m]`, `S[m+1]`, ... to but not including `S[n]`). The expression `S[m..]` (often called "drop `m`") is the same as `S[m..|S|]`, that is, it returns the sequence whose elements are all but the first `m` elements of `S`. The expression `S[..n]` (often called "take `n`") is the same as `S[0..n]`, that is, it returns the sequence that consists of the first `n` elements of `S`. If `j` is a valid index into sequence `S`, then the expression `S[j := x]` is the sequence that is like `S` except that it has `x` at index `j`. Finally, to make a sequence from some elements, enclose them in square brackets. For example, `[x,y]` is the sequence consisting of the two elements `x` and `y` such that `[x,y][0] == x` and `[x,y][1] == y`, `[x]` is the singleton sequence whose only element is `x`, and `[]` is the empty sequence.

The if-then-else expression has the form:

```
if BoolExpr then Expr0 else Expr1
```

where `Expr0` and `Expr1` are any expressions of the same type. Unlike the **`if`** statement, the if-then-else expression does not require parentheses around the guard expression, uses the **`then`** keyword, and must include an explicit **`else`** branch.

The **`match`** statement also has an analogous match expression, which has a form like:

```
match Expr
case Empty =>  Expr0
case Node(l, d, r) =>  Expr1
```

As with the **`if`** statement versus the if-then-else expression, note that the **`match`** expression does not require parentheses around the source expression and does not surround the cases with curly braces. A **`match`** expression can only be used in the body of function definitions, where it must either be the entire body or be the entire expression for a case in an enclosing **`match`** expression; furthermore, the source expression must be a parameter of the enclosing function.