Basic Constructs in Dafny

Costel Anghel and Mădălina Erașcu

Objectives

- Understand the basic constructs of Dafny [1, Chapter 1].
- Run some examples.

1 Recalling from the Lecture

- **Methods** in Dafny have input and output parameters, are defined with pre- and post-conditions, and hide their internal implementation. Preconditions and postconditions are used to specify the behavior and requirements of the methods. **Functions** have input parameters, produce a single output (that's an expression), and allow callers to see their implementation. Functions that return booleans are known as **predicates**.
- An **assert** statement in Dafny points out the expected conditions and triggers proof obligations that the programmer expects to hold.
- **Preconditions** in Dafny set conditions that must be met before a method can be invoked. **Postconditions**, on the other hand, specify conditions that the method's behavior must hold after it has executed, ensuring that the method performs as expected.
- **Ghosts** are constructs in Dafny used for program behavior description during verification but are not included in the compiled code. They include conditions and assertions used for validation.

1.1 Methods

Consider the following example:

```
method Triple(x: int) returns (r: int)
{
    var y := 2 * x;
    r := x + y;
}
```

If `x` is 10, what value does the method assign to `r`?

Consider the following example:

```
var t := Triple(18);
```

What is the value of `t`?

1.2 Assert Statements

Run the following example in Dafny IDE

```
method Triple(x: int) returns (r: int)
{
    var y := 2 * x;
    r := x + y;
    assert r == 3 * x;
}
```

What is the result? Change the asserted condition to `3*x+1`. What is now the result?

1.3 Working with the Verifier

Consider the following example:

```
method Triple(x: int) returns (r: int)
{
    var y := 2 * x;
    r := x + y;
    assert r == 10 * x;
    assert r < 5;
    assert false;
}
```

Run it in the Dafny IDE. What do you observe? Change the second assertion to make the verifier complain about the first two assertions but not about the third.

Remark 1 *If the condition of an* `assert` *is false, then the analysis of the program continues with this assumption. However, it might be that the assertion is false only for some values of the variables used in the assertion. In this case, the analysis continues with those values for which the assertion is true. As an example, replace the consition of the second* `assert` *with* `r==0` *and afterwards with any* $r \neq 0$*.*

1.4 Control Paths

Run the following examples in Dafny IDE. Give values to the input variables to show the deterministic/nondeterministic flavor of the `if` statement.

```
method Triple(x: int) returns (r: int)
{
    if x == 0{
            r := 0;
    } else {
        var y := 2 * x;
        r := x + y;
    }
    assert r == 3 * x;
}
```

```
method Triple(x: int) returns (r: int)
{
    if {
        case x < 18 =>
            var a, b := 2 * x, 4 * x;
            r := (a+b) / 2;
        case 0 <= x =>
            var y := 2 * x;
            r := x + y;
        }
    assert r == 3 * x;
}
```

1.5 Method Contracts

Consider the following examples:

```
method Caller()
{
    var result := Triple(18);
    assert result < 100;
}
```

```
method Triple(x: int) returns (r: int)
ensures r == 3 * x
{
    var y := 2 * x;
    r := x + y;
}
```

```
method Triple(x: int) returns (r: int)
requires x % 2 == 0
ensures r == 3 * x
{
    var y := x / 2;
    r := 6 * y;
}
```

1. What is the result of the verifier in each of the 3 examples?
2. Explain the role of the precondition and postcondition.
3. Remove (or comment out) the precondition of Triple. What error does the verifier give you?
4. Write two stronger alternatives to the precondition `x % 2 == 0` that also make the method Triple verify.

1.5.1 Underspecification

Consider the following method specification:

```
method Index(n: int) returns (i: int)
    requires 1 <= n
    ensures 0 <= i < n
```

The precondition says that *Index* can be called only on positive integers. The postcondition says that *Index*(n) will return i as some number between 0 and less than n. So, *which* number from 0 to n will be returned? The specification does not say. Therefore, when we reason about calls to *Index*, we consider all of these values. Viewed by a caller method, it is as if this method were *nondeterministic*. Here is one way to implement the method:

```
method Index(n: int) returns (i: int)
    requires 1 <= n
    ensures 0 <= i < n
{
    i := n / 2;
}
```

Here is another way to implement *Index*:

```
method Index(n: int) returns (i: int)
    requires 1 <= n
    ensures 0 <= i < n
{
    i := 0;
}
```

Both implementations are correct. Because methods are opaque (callers only get to see the specification of a method, not its body), it is fine to change the body of *Index* from `i:=n/2;` to `i:=0;` without affecting the correctness of any callers. Each of the two implementations we just considered is deterministic. That is, for a given input, the method body computes the output in the same way. But since methods are opaque, this is not something a caller can rely on. For example, in

```
var x := Index(50);
var y := Index(50);
assert x == y; //error
```

assertion cannot be proved, because all we know from the specification of *Index* is $0 \leq x < 50 \leq 0 \leq y < 50$ but we know of no connection between x and y. While the specification allows any value, the implementation can be (and most often is) deterministic. This important idea is called *underspecification*. It precisely specifies the freedom entailed by the caller-implementation contract.

1.5.2 Multiple postconditions

Consider the method below that computes the smaller of two given values:

```
method Min(x: int, y: int) returns (m: int)
ensures m <= x && m <= y
{
    if x <= y {
            m := x;
    } else {
            m := y;
    }
}
```

Which of the following postconditions precisely describes that the method computes the minimum?

```
method Min(x: int, y: int) returns (m: int) ensures m <= x && m <= y
```

```
method Min(x: int, y: int) returns (m: int)
ensures m <= x && m <= y
ensures m == x || m == y
```

Run the `Min` method with each of the specifications and check if both are satisfied. Write an implementation for method `Min`, call it `Minfake`, that satisfies one of the postconditions above namely the one which describes that the output is not always the minimum of x and y.

```
method Minfake(x: int, y: int) returns (m: int)
ensures m <= x && m <= y
{
    if x <= y {
        m := x - 10;
    } else {
        m := y - 20;
    }
}
```

Consider the method `Min` and its precise specification below

```
method Min(x: int, y: int) returns (m: int)
ensures m <= x && m <= y
ensures m == x || m == y
```

Write the verification conditions and prove them.
Branch 1:

$$\begin{aligned} x \le y \quad &\Rightarrow \quad \underbrace{x \le x}_{\mathbb{T}} \wedge x \le y \quad \Longleftrightarrow \\ x \le y \quad &\Rightarrow \quad x \le y \quad \checkmark \end{aligned}$$

Branch 2:

$$\begin{aligned} x > y \quad &\Rightarrow \quad y \le x \wedge \underbrace{y \le y}_{\mathbb{T}} \quad \Longleftrightarrow \\ x > y \quad &\Rightarrow \quad x \ge y \quad \Longleftrightarrow \\ \underbrace{x > y}_{K} \quad &\Rightarrow \quad \underbrace{x > y}_{G_2} || \underbrace{x = y}_{G_1} \end{aligned}$$

Next we prove:

$$\begin{aligned} x > y \wedge x \ne y \quad &\Rightarrow \quad x > y \quad \Longleftrightarrow \\ x > y \quad &\Rightarrow \quad x > y \quad \checkmark \end{aligned}$$

Consider the method `Minfake` and its specification

```
method Minfake(x: int, y: int) returns (m: int)
    ensures m <= x && m <= y
```

Write the verifications conditions and prove them.
Branch 1:

$$\begin{aligned} x \le y \quad &\Rightarrow \quad \underbrace{x - 10 \le x}_{\mathbb{T}} \wedge x - 10 \le y \quad \Longleftrightarrow \\ \underbrace{x \le y}_{K} \quad &\Rightarrow \quad \underbrace{x - 10 \le y}_{G} \end{aligned}$$

Proof by contradiction:

$$\begin{aligned} x \le y \quad \wedge \quad \neg(x - 10 \le y) \quad \text{is false} \\ x \le y \quad \wedge \quad x - 10 > y \quad \text{is false} \end{aligned}$$

If $x \leq y$ then $x - 10 << y$, so it cannot be false. ✓
Branch 2:

$$\begin{aligned} x > y &\Rightarrow y - 20 \leq x \wedge y - 20 \leq y \iff \\ x > y &\Rightarrow x \geq y - 20 \end{aligned}$$

Proof by contradiction:

$$x > y \quad \wedge \quad x < y - 20 \quad \text{is false}$$

If $x > y$ then $x >> y - 20 \Rightarrow \neg(x < y - 20)$ ✓

1.6 Functions

As you know from mathematics, a *function* denotes a value computed from given arguments. The key property of a function is that it is *deterministic*, that is, any two invocation of the function with the same arguments result in the same value.

Here is an example function declaration in Dafny:

```
    function Average (a: int, b: int): int{
        (a + b) / 2
    }
```

Functions can be used in expressions, so we can write a specification like this:

```
method Triple'(x: int) returns (r: int)
ensures Average(r, 3 * x) == 3 * x
```

1. The specification of Triple' is not the same as of Triple. Write a correct body for Triple' that does not meet the specification of Triple.
2. How can you strengthen the specification of Triple' with minimal changes to make it equivalent to the specification of Triple?
3. How can you use Dafny to prove that the specification in (b) are indeed equivalent?

The example above highlights another important difference between methods and functions in Dafny: whereas methods are opaque, functions are *transparent*. This is necessary, because if callers had to understand a function only from its specification, then the specification of functions could never make use of functions, which would severely limit what can be said about a function.

A function can have specifications. Of special importance is the function's precondition, which says under which circumstances the function is allowed to be invoked. For example, we can restrict uses of Average to non-negative arguments:

```
function Average(a: int, b: int): int
requires 0 <= a && 0 <= b
    {
        (a + b) / 2
    }
method Triple(x: int) returns (r: int)
ensures r == 3 * x
```

```
{
    if 0 <= x {
        r := Average(2 * x, 4 * x);
    } else {
    r := -Average(-2 * x, -4 * x);
    }
}
```

1.6.1 Predicates

Since assertions, preconditions, and postconditions use boolean conditions, it freequently happens that we declare boolean functions(that is, a function with result type **bool**) for use in specifications. A boolean function is also known as a ***predicate***.

```
predicate IsEven(x: int) {
    x % 2 == 0
}
```

is identical to

```
function IsEven(x: int): bool {
    x % 2 == 0
    }
```

1.7 Compiled versus Ghost

1. Consider the following examples.

```
    function F(): int {
        29
    }

    method M() returns (r: int)
    {
        r := 29;
    }
    method Caller() {
        var a := F();
        var b := M();
        assert a == 29;
        assert b == 29;
    }
```

   What does the verifier have to say about the two assertions in this program? Explain why. How can you change the program to make both assertions verify?

2. The function and method in the following code snippet are both declared as compiled.

```
    function Average (a: int, b: int): int{
        (a + b) / 2
    }
    method Triple (x: int) returns (r: int)
        ensures r == 3 * x
    {
        r := Average(2 * x, 4 * x);
    }

```

Change this example to declare `Average` as ghost. What error message do you get? Declare both `Average` and `Triple` as ghost. Is that legal? Why? Declare `Average` as compiled and `Triple` as ghost. Is that legal? Why?

2 Homework

1. Consider the signature of a method to compute `s` to be the sum of `x` and `y` and `m` to be the maximum of `x` and `y`:

```
method MaxSum (x: int, y: int) returns (s: int, m: int)
```

   (a) Specify the intended postcondition of this method.

   (b) Write an implementation for `MaxSum` and `Max`. `Max` computes the maximum of two integers.

   (c) Write a method that calls `MaxSum` with the input arguments 1928 and 1.

2. Consider a method with the following type signature and postcondition:

```
method ReconstructFromMaxSum(s: int, m: int) returns (x: int, y: int)
ensures s == x + y
ensures (m == x || m == y) && x <= m && y <= m
```

   a) Try to write the body for this method. You will find out you cannot. Write an appropriate precondition for the method that allows you to implement the method.

   b) Write the following test harness to test the method's specification:

```
method TestMaxSum(x: int, y: int)
{
    var s, m := MaxSum(x, y);
    var xx, yy := ReconstructFromMaxSum(s, m);
    assert (xx == x && yy == y) || (xx == y && yy == x);
}
```

How can you change the specification of ReconstructFromMaxSum to allow the assertion in the test harness to succeed?

Solution

```
function Max(a: int, b: int): int
{
    if a > b then a else b
}
```

```
method MaxSum(x: int, y: int) returns (s: int, m: int)
    ensures s == x + y && m == Max(x, y)
{
    s := x + y;
    m := Max(x, y);

}
```

```
method Caller(){
    var x := 1928;
    var y := 1;

    var sum, maximum := MaxSum(x, y);

    assert sum == x + y;
    assert maximum == Max(x, y);
}
```

```
method ReconstructFromMaxSum(s: int, m: int) returns (x: int, y: int)
    requires m <= s && s <= 2*m
    ensures s == x + y
    ensures (m == x || m == y) && x <= m && y <= m
{

    if x >= y{
        x := m;
        y := s-m;
    } else {
        y := m;
        x := s - m;
    }
}
```

```
method TestMaxSum(x: int, y: int){

    var s, m := MaxSum(x, y);
    assume m <= s <= 2*m;
    var xx, yy := ReconstructFromMaxSum(s, m);
    assert (xx == x && yy == y) || (xx == y && yy == x);

}
```

References

[1] K. R. M. Leino. *Program Proofs*. MIT Press, 2023.