Formal Methods in Software Development
Course 10. Program Proofs. Dafny IDE

Mădălina Eraşcu

Content based on the book Leino, K. Rustan M. Program Proofs. MIT Press, 2023

Thanks to Costel Anghel, 3rd Year Bachelor Student, Applied Informatics

June 5, 2024

Contents

Program Proofs and Dafny

https://www.youtube.com/watch?v=oLS_y842fMc

Contents

Contents

Methods

Definition

A method is a program declaration that prescribes some behavior.

Example

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

- input: *in-parameter* `x` of type integer
- output: *out-parameter* `r` of type integer
- the *body* of a method is a list of *statements* that give the method's implementation
- local variable `y` to which it assigns the value `2*x`

Remark

- Methods can have any number of in-parameters and any number of out-parameters.
- In the body of the method, the out-parameters act as local variables and can be assigned and read.
- At the end of the methods body whatever values the out-parameters have will be the values returned to the caller.
- The in-parameters can of course be read, but they cannot be re-assigned in the method body.

Contents

Assert Statements. Control Paths

Example

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

- An assertion in a Dafny program is tried to be proved by the verifier.

Remark

`if` statements and other conditional statements determine many control paths. A program is correct when the traces along *all* control paths are correct.

Assert Statements. Control Paths (cont'd)

Example

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

Remark

- For the example above, the branches are dermined by `x==0` and `x!=0`.
- Control in `if` statement is *deterministic*.

Control Paths (cont'd)

Control flow can also be *nondeterministic*, which means that repeatedly running the program, even on the same input, may result in different traces, see below.

Example

```
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;
}
```

Contents

Method Contracts

Definition

A *client* of a method (or function or type or module) is a piece of code that wants to use the method (or function or type or module).

Consider a client of the Triple method:

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

- An "agreement" (contract) between the caller and the implementation that says what the caller can rely on is used.
- Since the method contract, not the method body, is used at call sites, we say that the methods are *opaque*.

Method Contracts (cont'd)

Definition

A method contract has two fundamental parts: a precondition and a postcondition. The precondition says when it is legal for a caller to invoke the method. It is a proof obligation at every call site, and in exachange it can be assumed to hold at the start of the method body. The postcondition is a proof obligation at every return point from the method body, and in exchange it can be assumed to hold upon return from the invocation at the call site.

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

Exercise. Write two stronger alternatives to the precondition which make the method `Triple` verify.

Definition

Consider the formula $A \Rightarrow B$. We say that A is stronger than B and B is weaker than A.

Contents

Verification Conditions

Consider a method 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;
    }
}
```

We prove the following 2 verification conditions:

Branch 1:

$$x \leq y \Rightarrow \underbrace{x \leq x}_{\mathbb{T}} \wedge x \leq y \quad \Longleftrightarrow \quad x \leq y \Rightarrow x \leq y \quad \checkmark$$

Branch 2:

$$x > y \Rightarrow y \leq x \wedge \underbrace{y \leq y}_{\mathbb{T}} \quad \Longleftrightarrow \quad \underbrace{x > y}_{K} \Rightarrow \underbrace{x > y}_{G_2} || \underbrace{x = y}_{G_1}$$

Further we prove $x > y \wedge x! = y \Rightarrow x > y \checkmark$

Contents

Functions

Definition

A *function* denotes a value computed from given arguments. The key property of a function is that it is *deterministic*, that is, any two invocations of the function with the same arguments result in the same value.

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

Functions can be used in expressions.

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

Remark

Functions in Dafny are *transparent* 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.

Contents

Predicates

Definition

A boolean function is a predicate.

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

is identical to

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

Contents

Compiled versus Ghost

Definition

A declaration, variable, statement, etc., that is used only for specifications purposes is called a ghost.

Remark

The verifier takes all ghosts into account; the compiler erases all ghosts when it generates executable code.

Definition

Program constructs that make it into the executable code are referred to as *compiled* or *non-ghost*.

Which `ghost` constructs we already used?

Remark

To make the erasure of ghosts possible, Dafny checks that compiled code does not rely on ghost constructs.

Example

```
method IllegalAssignment() returns (y: int){
    ghost var x := 10;
    y := 2 * x; //error: cannot assign
                //to compiled variable using a ghost
}
```

Remark

A program is not allowed to use a ghost variable in the right-hand side of an assignment to a compiled variable.

Compiled versus Ghost (cont'd)

Example

```
method Triple(x: int) returns (r: int)
    ensures r == 3 * x
{
    var y := 2 * x;
    r := x + y;
    ghost var a, b := DoubleQuadruple(x);
    assert a <= r <= b || b <= r <= a;
}

ghost method DoubleQuadruple(x: int) returns (a: int, b: int)
    ensures a == 2 * x && b == 4 * x
{
    a := 2 * x;
    b := 2 * a;
}
```

What happens when the code above is compiled?