Formal Methods in Software Engineering

Specification and Verification, Spring 2026

Konstantin Chukharev

Program Verification

Motivation

Is this program *correct*?

```
x = 0;
y = a;
while (y > 0) {
    x = x + b;
    y = y - 1;
}
```

Program Correctness

Note: A program can be *correct* only with respect to a *specification*.

Is this program correct with respect to the following specification? ✗

"Given integers a and b, the program computes and stores in x the product of a and b."

Program Correctness [2]

Note: A program can be *correct* only with respect to a *specification*.

Is this program correct with respect to the following specification? ✓

*"Given **positive** integers a and b, the program computes and stores in x the product of a and b."*

```
x = 0;
y = a;
while (y > 0) {
    x = x + b;
    y = y - 1;
}
```

Design by Contract

Specification of a program can be seen as a *contract*:

- *Pre-conditions* define what is *required* to get a meaningful result.
- *Post-conditions* define what is *guaranteed* to return when the pre-condition is met.

requires a and b to be positive integers
ensures x is the product of a and b

Formal Verification

To formally verify a program you need:

- A formal specification (mathematical description) of the program.
- A formal proof that the specification is correct.
- Automated tools for verification and reasoning.
- Domain-specific expertise.

Historical context: Floyd-Hoare logic (1969) laid the theoretical foundations. Dijkstra's weakest precondition calculus (1975) made it practical. Modern tools like Dafny (2009, Microsoft Research) combine:

- Programming language design
- SMT solvers (Z3) for automation
- Boogie intermediate verification language
- User-friendly syntax inspired by C# and Java

Formal Verification [2]

There are many tools and even specific languages for writing specs and verifying them.

One of them is *Dafny*, both a specification language and a program verifier.

Next, we are going to learn how to:

- *specify* precisely what a program is supposed to do
- *prove* that the specification is correct
- *verify* that the program behaves as specified
- *derive* a program from a specification
- use the *Dafny* programming language and verifier

Dafny

Introduction to Dafny

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

Note: The *caller* does not need to know anything about the *implementation* of the method, only its *specification*, which abstracts the method's behavior. The method is *opaque* to the caller.

Introduction to Dafny [2]

Completing the example:

```
method Triple(x: int) returns (r: int)
  requires x >= 0
  ensures r == 3 * x
{
  var y := Double(x);
  r := x + y;
}

method Double(x: int) returns (r: int)
  requires x >= 0
  ensures r == 2 * x
```

Exercise: Fix the above code/spec to avoid `requires x >= 0` in the `Triple` method.

Logic in Dafny

Dafny expression	Description
`true`, `false`	constants
`!A`	"not A"
`A && B`	"A and B"
`A || B`	"A or B"
`A ==> B`	"A implies B" or "A only if B"
`A <==> B`	"A iff B"
`forall x :: A`	"for all x, A is true"
`exists x :: A`	"there exists x such that A is true"

Precedence order: `!`, `&&`, `||`, `==>`, `<==>`

Verifying the Imperative Procedure

Below is the Dafny program for computing the maximum segment sum of an array. Source: [1]

```
// find the index range [k..m) that gives the
largest sum of any index range
method MaxSegSum(a: array<int>)
  returns (k: int, m: int)
  ensures 0 ≤ k ≤ m ≤ a.Length
  ensures forall p, q ::
          0 ≤ p ≤ q ≤ a.Length ==>
          Sum(a, p, q) ≤ Sum(a, k, m)
{
  k, m := 0, 0;
  var s, n, c, t := 0, 0, 0, 0;
  while n < a.Length
    invariant 0 ≤ k ≤ m ≤ n ≤ a.Length &&
              s == Sum(a, k, m)
    invariant forall p, q ::
              0 ≤ p ≤ q ≤ n ==> Sum(a, p, q) ≤ s
    invariant 0 ≤ c ≤ n && t == Sum(a, c, n)
    invariant forall b ::
              0 ≤ b ≤ n ==> Sum(a, b, n) ≤ t
```

```
  {
    t, n := t + a[n], n + 1;
    if t < 0 {
      c, t := n, 0;
    } else if s < t {
      k, m, s := c, n, t;
    }
  }
}

// sum of the elements in the index range [m..n)
function Sum(a: array<int>, m: int, n: int): int
  requires 0 ≤ m ≤ n ≤ a.Length
  reads a
{
  if m == n then 0
  else Sum(a, m, n-1) + a[n-1]
}
```

Program State

```
method MyMethod(x: int) returns (y: int)
  requires x >= 10
  ensures y >= 25
{
  var a := x + 3;
  var b := 12;
  y := a + b;
}
```

The program variables `x`, `y`, `a`, and `b` together form the method's *state*.

Note: Not all program variables are in scope the whole time.

Floyd Logic

Let's propagate the pre-condition *forward*:

```
method MyMethod(x: int) returns (y: int)
  requires x >= 10
  ensures y >= 25
{
  // here, we know x >= 10
  var a := x + 3;
  // here, x >= 10 && a == x+3
  var b := 12;
  // here, x >= 10 && a == x+3 && b == 12
  y := a + b;
  // here, x >= 10 && a == x+3 && b == 12 && y == a + b
}
```

The last constructed condition *implies* the required post-condition:

$$(x \geq 10) \wedge (a = x + 3) \wedge (b = 12) \wedge (y = a + b) \rightarrow (y \geq 25)$$

Floyd Logic [2]

Now, let's go *backward* starting with a post-condition at the last statement:

```
method MyMethod(x: int) returns (y: int)
  requires x >= 10
  ensures y >= 25
{
  // here, we want x + 3 + 12 >= 25
  var a := x + 3;
  // here, we want a + 12 >= 25
  var b := 12;
  // here, we want a + b >= 25
  y := a + b;
  // here, we want y >= 25
}
```

The last calculated condition is *implied* by the given pre-condition:

$$(x + 3 + 12 \geq 25) \leftarrow (x \geq 10)$$

Exercise #1

Consider a method with the type signature below which returns in `s` the sum of `x` and `y`, and in `m` the maximum of `x` and `y`:

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

Write the post-condition specification for this method.

Exercise #2

Consider a method that attempts to reconstruct the arguments `x` and `y` from the return values of `MaxSum`.
In other words, in other words, consider a method with the following type signature and *the same post-condition* as in Exercise 1:

```
method ReconstructFromMaxSum(s: int, m: int)
  returns (x: int, y: int)
  requires ...
  ensures ...
```

This method cannot be implemented as is.
Write an appropriate pre-condition for the method that allows you to implement it.

Floyd-Hoare Logic

From Contracts to Floyd-Hoare Logic

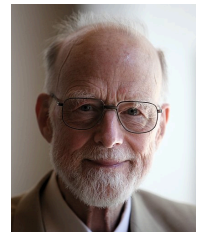
In the design-by-contract methodology, contracts are usually assigned to procedures or modules. In general, it is possible to assign contracts to each statement of a program.

A formal framework for doing this was developed by Tony Hoare [2], formalizing a reasoning technique introduced by Robert Floyd [3].

It is based on the notion of a *Hoare triple*.

Dafny is based on Floyd-Hoare Logic.

Robert Floyd



Tony Hoare

Hoare Triples

Definition 1: For predicates P and Q, and a problem S, the Hoare triple $\{P\}\ S\ \{Q\}$ describes how the execution of a piece of code changes the state of the computation.

It can be read as "if S is started in any state that satisfies P, then S will terminate (and does not crash) in a state that satisfies Q".

Examples:

$$\{x = 1\} \quad x := 20 \quad \{x = 20\}$$
$$\{x < 18\} \quad y := 18 - x \quad \{y \geq 0\}$$
$$\{x < 18\} \quad y := 5 \quad \{y \geq 0\}$$

Non-examples:

$$\{x < 18\} \quad x := y \quad \{y \geq 0\}$$

Forward Reasoning

Definition 2: *Forward reasoning* is a construction of a *post-condition* from a given pre-condition.

Note: In general, there are *many* possible post-conditions.

Examples:

$\{x = 0\} \quad y := x + 3 \quad \{y < 100\}$

$\{x = 0\} \quad y := x + 3 \quad \{x = 0\}$

$\{x = 0\} \quad y := x + 3 \quad \{0 \leq x, y = 3\}$

$\{x = 0\} \quad y := x + 3 \quad \{3 \leq y\}$

$\{x = 0\} \quad y := x + 3 \quad \{\texttt{true}\}$

Strongest Post-condition

Forward reasoning constructs the *strongest* (*i.e.*, *the most specific*) post-condition.

$$\{x = 0\} \quad y := x + 3 \quad \{0 \leq x \wedge y = 3\}$$

Definition 3: A is *stronger* than B if $A \rightarrow B$ is a valid formula.

Definition 4: A formula is *valid* if it is true for any valuation of its free variables.

Backward Reasoning

Definition 5: *Backward reasoning* is a construction of a *pre-condition* for a given post-condition.

Note: Again, there are *many* possible pre-conditions.

Examples:

$$\begin{aligned}
\{x \leq 70\} \quad & y := x + 3 \quad \{y \leq 80\} \\
\{x = 65, y < 21\} \quad & y := x + 3 \quad \{y \leq 80\} \\
\{x \leq 77\} \quad & y := x + 3 \quad \{y \leq 80\} \\
\{x \cdot x + y \cdot y \leq 2500\} \quad & y := x + 3 \quad \{y \leq 80\} \\
\{\texttt{false}\} \quad & y := x + 3 \quad \{y \leq 80\}
\end{aligned}$$

Weakest Pre-condition

Backward reasoning constructs the *weakest* (*i.e.*, *the most general*) pre-condition.

$$\{x \leq 77\} \quad y := x + 3 \quad \{y \leq 80\}$$

Definition 6: A is *weaker* than B if $B \rightarrow A$ is a valid formula.

Weakest Pre-condition for Assignment

Definition 7: The weakest pre-condition for an *assignment* statement $x := E$ with a post-condition Q, is constructed by replacing each x in Q with E, denoted $Q[x := E]$.

$$\{Q[x := E]\} \quad x := E \quad \{Q\}$$

Example: Given a Hoare triple $\{?\}\ y := a + b\ \{25 \leq y\}$, we construct a pre-condition $\{25 \leq a + b\}$.

Examples:

$$\begin{array}{rll} \{25 \leq x + 3 + 12\} & a := x + 3 & \{25 \leq a + 12\} \\ \{x + 1 \leq y\} & x := x + 1 & \{x \leq y\} \\ \{6x + 5y < 100\} & x := 2 \cdot x & \{3x + 5y < 100\} \end{array}$$

Exercises

1. Explain rigorously why each of these Hoare triples holds:
   1. $\{x = y\} \quad z := x - y \quad \{z = 0\}$
   2. $\{\text{true}\} \quad x := 100 \quad \{x = 100\}$
   3. $\{\text{true}\} \quad x := 2y \quad \{x \text{ is even}\}$
   4. $\{x = 89\} \quad y := x - 34 \quad \{x = 89\}$
   5. $\{x = 3\} \quad x := x + 1 \quad \{x = 4\}$
   6. $\{0 \leq x < 100\} \quad x := x + 1 \quad \{0 < x \leq 100\}$
2. For each of the following Hoare triples, find the *strongest post-condition*:
   1. $\{0 \leq x < 100\} \quad x := 2x \quad \{?\}$
   2. $\{0 \leq x \leq y < 100\} \quad z := y - x \quad \{?\}$
   3. $\{0 \leq x < N\} \quad x := x + 1 \quad \{?\}$
3. For each of the following Hoare triples, find the *weakest pre-condition*:
   1. $\{?\} \quad b := (y < 10) \quad \{b \to (x < y)\}$
   2. $\{?\} \quad x, y := 2x, x + y \quad \{0 \leq x \leq 100y \leq x\}$
   3. $\{?\} \quad x := 2y \quad \{10 \leq x \leq y\}$

Swap Example

Consider the following program that swaps the values of x and y using a temporary variable.

```
var tmp := x;
x := y;
y := tmp;
```

Let's prove that it indeed swaps the values, by performing the backward reasoning on it. First, we need a way to refer to the initial values of x and y in the post-condition. For this, we use *logical variables* that stand for some values (initially, $x = X$ and $y = Y$) in our proof, yet cannot be used in the program itself.

```
// { x == X, y == Y }
// { ? }
var tmp := x;
// { ? }
x := y;
// { ? }
y := tmp
// { y == Y, x == X }
```

Simultaneous Assignment

Dafny allows simultaneous assignment of multiple variables in a single statement.

Examples:

$x, y := 3, 10$ sets x to 3 and y to 10

$x, y = x + y, x - y$ sets x to the sum of x, and y and y to their difference

All right-hand sides are evaluated *before* any variables are assigned.

Note: The last example is *different* from the two statements: `x = x + y; y = x - y;`

Weakest Pre-condition for Simultaneous Assignment

Definition 8: The weakest pre-condition for a *simultaneous assignment* $x_1, x_2 := E_1, E_2$ is constructed by replacing each x_1 with E_1 and each x_2 with E_2 in post-condition Q.

$$Q[x_1 := E_1, x_2 := E_2] \quad x_1, x_2 := E_1, E_2 \quad \{Q\}$$

Example: Going *backward* in the following "swap" program:

```
// { x == X, y == Y } -- initial state
// { y == Y, x == X } -- weakest pre-condition
x, y = y, x
// { x == Y, y == X } -- final "swapped" state
```

Weakest Pre-condition for Variable Introduction

Note: The statement `var x := tmp;` is actually *two* statements: `var x; x := tmp`.

What is true about x in the post-condition, must have been true for all x before the variable introduction.

$$\{\forall x.\, Q\} \quad \texttt{var } x \quad \{Q\}$$

Examples:

- $\{\forall x.\, 0 \leq x\} \quad \texttt{var x} \quad \{0 \leq x\}$
- $\{\forall x.\, 0 \leq x \cdot x\} \quad \texttt{var x} \quad \{0 \leq x \cdot x\}$

Strongest Post-condition for Assignment

Consider the Hoare triple

$$\{w < x, x < y\} \quad x := 100 \quad \{?\}$$

Obviously, $x = 100$ is a post-condition, however it is *not the strongest*.

Something *more* is implied by the pre-condition: there exists an n such that $(w < n) \wedge (n < y)$, which is equivalent to $w + 1 < y$.

In general:

$$\{P\} \quad x := E \quad \{\exists n.\, P[x := n] \wedge x = E[x := n]\}$$

Exercises

Replace the "?" in the following Hoare triples by computing *strongest post-conditions*.

1. $\{y = 10\} \quad x := 12 \quad \{?\}$
2. $\{98 \leq y\} \quad x := x + 1 \quad \{?\}$
3. $\{98 \leq x\} \quad x := x + 1 \quad \{?\}$
4. $\{98 \leq y < x\} \quad x := 3y + x \quad \{?\}$

$\mathcal{WP}$ and $\mathcal{SP}$

Let P be a predicate on the *pre-state* of a program S, and let Q be a predicate on the *post-state* of S.

$\mathcal{WP}[\![S, Q]\!]$ denotes the *weakest pre-condition* of S w.r.t. Q.

- $\mathcal{WP}[\![\text{var } x, Q]\!] = \forall x.\, Q$
- $\mathcal{WP}[\![x := E, Q]\!] = Q[x := E]$
- $\mathcal{WP}[\![(x_1, x_2 := E_1, E_2), Q]\!] = Q[x_1 := E_1, x_2 := E_2]$

$\mathcal{SP}[\![S, P]\!]$ denotes the *strongest post-condition* of S w.r.t. P.

- $\mathcal{SP}[\![\text{var } x, P]\!] = \exists x.\, P$
- $\mathcal{SP}[\![x := E, P]\!] = \exists n.\, P[x := n] \wedge x = E[x := n]$

Exercise: Compute the following pre- and post-conditions:

- $\mathcal{WP}[\![x := y, x + y \leq 100]\!]$
- $\mathcal{WP}[\![x := -x, x + y \leq 100]\!]$
- $\mathcal{WP}[\![x := x + y, x + y \leq 100]\!]$
- $\mathcal{WP}[\![z := x + y, x + y \leq 100]\!]$
- $\mathcal{WP}[\![\text{var } x, x \leq 100]\!]$
- $\mathcal{SP}[\![x := 5, x + y \leq 100]\!]$
- $\mathcal{SP}[\![x := x + 1, x + y \leq 100]\!]$
- $\mathcal{SP}[\![x := 2y, x + y \leq 100]\!]$
- $\mathcal{SP}[\![z := x + y, x + y \leq 100]\!]$
- $\mathcal{SP}[\![\text{var } x, x \leq 100]\!]$

Control Flow

Statement	Program
Assignment	$x := E$
Local variable	var x
Composition	$S; T$
Condition	if B then $\{S\}$ else $\{T\}$
Assumption	assume P
Assertion	assert P
Method call	$r := M(E)$
Loop	while B do $\{S\}$

Sequential Composition

$$S;T$$

$$\{P\}\ S\ \{Q\}\ T\ \{R\}$$

$$\{P\}\ S\ \{Q\} \quad \text{and} \quad \{Q\}\ T\ \{R\}$$

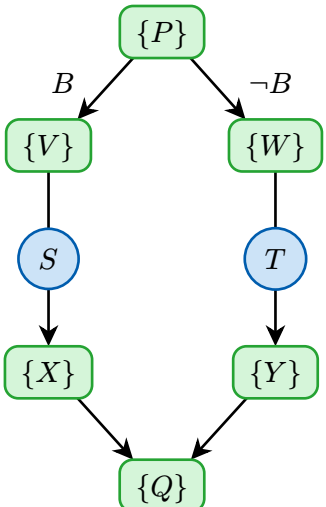
Strongest post-condition:

- Let $Q = \mathcal{SP}[\![S, P]\!]$
- $\mathcal{SP}[\![(S;T), P]\!] = \mathcal{SP}[\![T, Q]\!] = \mathcal{SP}[\![T, \mathcal{SP}[\![S, P]\!]]\!]$

Weakest pre-condition:

- Let $Q = \mathcal{WP}[\![T, R]\!]$
- $\mathcal{WP}[\![(S;T), R]\!] = \mathcal{WP}[\![S, Q]\!] = \mathcal{WP}[\![S, \mathcal{WP}[\![T, R]\!]]\!]$

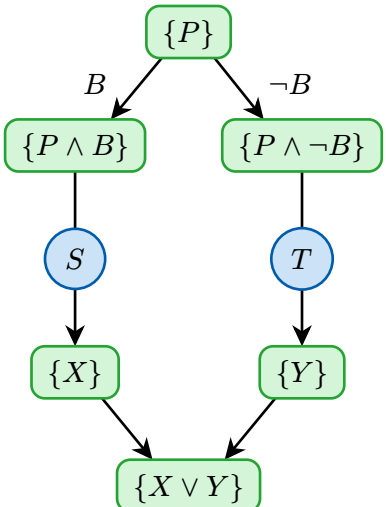
Conditional Control Flow



$\{P\}$ if B then $\{S\}$ else $\{T\}$ $\{Q\}$

1. $(P \wedge B) \to V$
2. $(P \wedge \neg B) \to W$
3. $\{V\}\ S\ \{X\}$
4. $\{W\}\ T\ \{Y\}$
5. $X \to Q$
6. $Y \to Q$

Strongest Post-condition for Condition



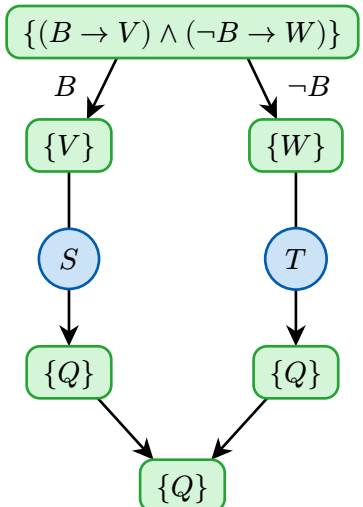
$\{P\}$ if B then $\{S\}$ else $\{T\}$ $\{Q\}$

$V = P \wedge B$
$W = P \wedge \neg B$

$X = \mathcal{SP}[\![S, P \wedge B]\!]$
$Y = \mathcal{SP}[\![T, P \wedge \neg B]\!]$

$\mathcal{SP}[\![\text{if } B \text{ then } \{S\} \text{ else } \{T\}, P]\!] =$
$= X \vee Y =$
$= \mathcal{SP}[\![S, P \wedge B]\!] \vee \mathcal{SP}[\![T, P \wedge \neg B]\!]$

Weakest Pre-condition for Condition



$\{P\}$ if B then $\{S\}$ else $\{T\}$ $\{Q\}$

$\mathcal{WP}[\![\text{if } B \text{ then } \{S\} \text{ else } \{T\}, Q]\!] =$

$= (B \to V) \wedge (\neg B \to W) =$

$= (B \to \mathcal{WP}[\![S, Q]\!]) \wedge (\neg B \to \mathcal{WP}[\![T, Q]\!])$

$V = \mathcal{WP}[\![S, Q]\!]$

$W = \mathcal{WP}[\![T, Q]\!]$

$X = Q$

$Y = Q$

Example

```
// { x == 50 }
// ... (see right)
// { (x < 3 ==> x == 89) && (x >= 3 ==> x == 50) }
if x < 3 {
  // { x == 89 }
  // { x + 1 + 10 == 100 }
  x, y := x + 1, 10;
  // { x + y == 100 }
} else {
  // { x == 50 }
  // { x + x == 100 }
  y := x;
  // { x + y == 100 }
}
// { x + y == 100 }
```

$$
\begin{aligned}
&((x < 3) \to (x = 89)) \wedge ((x \geq 3) \to (x = 50)) \equiv \\
\equiv\ &((x \geq 3) \vee (x = 89)) \wedge ((x < 3) \vee (x = 50)) \equiv \\
\equiv\ &((x \geq 3) \wedge (x < 3)) \vee ((x \geq 3) \wedge (x = 50)) \vee \\
&\vee ((x = 89) \wedge (x < 3)) \vee ((x = 89) \wedge (x = 50)) \equiv \\
\equiv\ &(\bot \vee (x = 50) \vee \bot \vee \bot) \equiv \\
\equiv\ &(x = 50)
\end{aligned}
$$

Method Correctness

Given

```
method M(x: Tx) returns (y: Ty)
  requires P
  ensures Q
{
  B
}
```

we need to prove $P \rightarrow \mathcal{WP}[\![B, Q]\!]$.

Method Calls

Methods are *opaque*, *i.e.*, we reason in terms of their *specifications*, not their implementations.

Example: Given the following definition (or rather, declaration):

```
method Triple(x: int) returns (y: int)
  ensures y == 3 * x
```

we expect to be able to prove, for example, the following method call:

$$\{\texttt{true}\} \quad v := \texttt{Triple}(u + 4) \quad \{v = 3 \cdot (u + 4)\}$$

Parameters

We need to *relate* the *actual* parameters (arguments of the method call) with the *formal* parameters (of the method).

To avoid any name slashes, we first *rename* the formal parameters to *fresh* variables:

```
method Triple(x1: int) returns (y1: int)
  ensures y1 == 3 * x1
```

Then, for a call `v := Triple(u + 1)` we have:

```
x1 := u + 1;
v := y1;
```

Assumptions

The called can assume that the method's post-condition holds.

We introduce a new statement, `assume E`, to capture this:

$$\mathcal{SP}[\![\,\text{assume } E, P\,]\!] = P \wedge E$$
$$\mathcal{WP}[\![\,\text{assume } E, Q\,]\!] = E \rightarrow Q$$

The semantics of `v := Triple(u + 1)` is then given by

```
var x1; var y1;
x1 := u + 1;
assume y1 == 3 * x1;
v := y1;
```

```
method Triple(x1: int)
returns (y1: int)
  ensures y1 == 3 * x1
```

Weakest Pre-condition for Method Calls

```
method M(x: X) returns (y: Y) ensures R[x, y]
```

$$\mathcal{WP}[\![\, r := M(E), Q \,]\!] =$$
$$= \mathcal{WP}[\![\, \text{var } x_E; \text{var } y_E; x_E := E; \text{assume } R[x, y := x_E, y_r]; r := y_r, Q \,]\!] =$$
$$= \mathcal{WP}[\![\, \text{var } x_E, \mathcal{WP}[\![\, \text{var } y_r, \mathcal{WP}[\![\, x_E := E, \mathcal{WP}[\![\, \text{assume } R[x, y := x_E, y_r], \mathcal{WP}[\![\, r := y_r, Q \,]\!] \,]\!] \,]\!] \,]\!] =$$
$$= \mathcal{WP}[\![\, \text{var } x_E, \mathcal{WP}[\![\, \text{var } y_r, \mathcal{WP}[\![\, x_E := E, \mathcal{WP}[\![\, \text{assume } R[x, y := x_E, y_r], Q[r := y_r] \,]\!] \,]\!] \,]\!] =$$
$$= \mathcal{WP}[\![\, \text{var } x_E, \mathcal{WP}[\![\, \text{var } y_r, \mathcal{WP}[\![\, x_E := E, R[x, y := x_E, y_r] \to Q[r := y_r] \,]\!] \,]\!] =$$
$$= \mathcal{WP}[\![\, \text{var } x_E, \forall x_E.\, R[x, y := x_E, y_r] \to Q[r := y_r] \,]\!] =$$
$$= \forall y_r. \forall x_E.\, R[x, y := x_E, y_r] \to Q[r := y_r]$$

Overall:

$$\mathcal{WP}[\![\, r := M(E), Q \,]\!] = \forall y_r.\, R[x, y := E, y_r] \to Q[r := y_r]$$

where x is M's input, y is M's output, and R is M's post-condition.

Example

Example:

```
method Triple(x: int) returns (y: int)
  ensures y == 3 * x
```

Consider calling this method with $Q = \{v = 48\}$. Backward reasoning:

```
// { u == 15 }
// { 3 * (u + 1) == 48 }
// { forall y1 :: y1 == 3 * (u + 1) ==> y1 == 48 }
v := Triple(u + 1);
// { v == 48 }
```

Assertions

`assert E` does nothing when E holds, otherwise it crashes the program.

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

$$\mathcal{SP}[\![\,\text{assert } E, P\,]\!] = P \wedge E$$
$$\mathcal{WP}[\![\,\text{assert } E, Q\,]\!] = E \wedge Q$$

Note: Both $\mathcal{SP}$ and $\mathcal{WP}$ are conjunctions, contrary to `assume`!

Method Calls with Pre-conditions

Given a method with a pre-condition:

```
method M(x: X) returns (y: Y)
  requires P
  ensures R
```

The semantics of `r := M(E)` is:

```
var x_E; var y_r;
x_E := E;
assert P[x := x_E];
assume R[x,y := x_E,y_r];
r := y_r;
```

$$\mathcal{WP}[\![r := M(E), Q]\!] = P[x := E] \wedge \forall y_r.\, R[x, y := E, y_r] \rightarrow Q[r := y_r]$$

Function Calls

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

Differences from method calls:

- No output parameters, just a single output.
- The body is an *expression*, not a statement.
- Functions are *transparent*: we reason about them in terms of their definition by *unfolding* it.

```
method Triple(x: int) return (r: int)
  ensures r == 3 * x
{ r := Average(2*x, 4*x); }
```

```
method Triple(x: int) return (r: int)
  ensures r == 3 * x
{ r := (2*x + 4*x) / 2; }
```

Ghost Functions

In *Dafny*, functions are part of the *code*.

If you want to use a function in a *specification*, you need to use a *ghost function*.

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

Partial Expressions

An expression may be not always well-defined, *e.g.*, c/d when d evaluates to 0.

Associated with such *partial expressions* are *implicit assertions*.

Example:

```
assert d != 0 && v != 0;
if c/d < u/v {
  assert 0 <= i < a.Length;
  x := a[i];
}
```

Function may have pre-conditions, making calls to them *partial*.

Example:

```
function MinusOne(x: int): int
  requires 0 < x
```

The call `z := MinusOne(y + 1)` has an implicit assertion `assert 0 < y + 1`.

Exercises

1. Suppose you want $x + y = 22$ to hold after the statement

$$\text{if } x < 20 \text{ then } \{y := 3\} \text{ else } \{y := 2\}$$

   In which states can you start the statement? (Compute the weakest pre-condition.)

2. Compute the weakest pre-condition for the following statement with respect to $y < 10$. Simplify

```
if x < 8 {
  if x == 5 { y := 10; } else { y := 2; }
} else {
  y := 0;
}
```

Exercises [2]

3. Compute the weakest pre-condition for the following statement with respect to `y % 2 == 0`.

```
if x < 10 {
  if x < 20 { y := 1; } else { y := 2; }
} else {
  y := 4;
}
```

4. Compute the weakest pre-condition for the following statement with respect to `y % 2 == 0`.

```
if x < 8 {
  if x < 4 { x := x + 1; } else { y := 2; }
} else {
  if x < 32 { y := 1; } else { }
}
```

Exercises [3]

5. Determine under which circumstances the following program establishes $0 \leq y < 100$. Try first to do that in your head. Write down the answer you come up with, and then write out the full computations to check that you got the right answer.

```
if x < 34 {
  if x == 2 { y := x + 1; } else { y := 233; }
} else {
  if x < 55 { y := 21; } else { y := 144; }
}
```

6. Which of the following Hoare-triple combinations are valid?
   1. $\{0 \leq x\} \quad x := x + 1 \quad \{-2 \leq x\} \quad y := 0 \quad \{-10 \leq x\}$
   2. $\{0 \leq x\} \quad x := x + 1 \quad \{\texttt{true}\} \quad x := x + 1 \quad \{2 \leq x\}$
   3. $\{0 \leq x\} \quad x := x + 1; \quad x := x + 1 \quad \{2 \leq x\}$
   4. $\{0 \leq x\} \quad x := 3x; \quad x := x + 1 \quad \{3 \leq x\}$
   5. $\{x < 2\} \quad y := x + 5; \quad x := 2x \quad \{x < y\}$

Exercises [4]

7. Compute the weakest pre-conditions with respect to the post-condition $x + y < 100$.
   1. `x := 32; y := 40;`
   2. `x := x + 2; y := y - 3 * x;`
8. Compute the weakest pre-conditions with respect to the post-condition $x < 10$.
   1. `if x % 2 == 0 { y := y + 3; } else { y := 4; }`
   2. `if y < 10 { y := x + y; } else { x := 8; }`
9. Compute the weakest pre-conditions with respect to the post-condition $x < 100$.
   1. `assert y == 25;`
   2. `assert 0 <= x;`
   3. `assert x < 200;`
   4. `assert x <= 100;`
   5. `assert 0 <= x < 100;`

Exercises [5]

10. If x_1 does not appear in the desired post-condition Q, then prove that $x_1 := E; \text{assert } P[x := x_1]$ is the same as $P[x := E]$ by showing that the weakest pre-conditions of these two statements with respect to Q are the same.

11. What implicit assertions are associated with the following expressions?
    1. `x / (y + z)`
    2. `arr[2 * i]`
    3. `MinusOne(MinusOne(y))`

12. What implicit assertions are associated with the following expressions? **Note:** the right-hand expression in a conjunction is only evaluated when the left-hand conjunction holds.
    1. `a / b < c / d`
    2. `a / b < 10 && c / d < 100`
    3. `MinusOne(y) = 8 ==> arr[y] = 2`

Recursion and Termination

Recursive Methods

```
method Double(x: int) returns (y: int)
  requires x >= 0
  ensures r == 2 * x
{
  // { x != 0 ==> x > 0 }
  // { (x == 0) ==> (0 == 2 * x)  &&  (x != 0) ==> (x - 1 >= 0) }
  if x == 0 {
    // { 0 == 2 * x }
    y := 0;
    // { y == 2 * x }
  } else {
    // { forall t :: x - 1 >= 0 }
    var t;
    // { x - 1 >= 0  &&  forall r :: (r == 2 * (x - 1)) ==> (r + 2 == 2 * x) }
    t := Double(x - 1);
    // { t + 2 == 2 * x }
    y := t + 2;
    // { y == 2 * x }
  }
  // { y == 2 * x }
}
```

Recursive methods can be analyzed like any methods that call other methods... **if they terminate**.

Problematic Recursion

```
method BadDouble(x: int) returns (y: int)
  requires x >= 0
  ensures y == 2 * x
{
  var t := BadDouble(x - 1);  // Infinite recursion, does not terminate!
  y := t + 2;
}


method PartialIdentity(x: int) returns (y: int)
  ensures y == x
{
  if x % 2 == 2 {
    y := x;
  } else {
    y := PartialIdentity(x);  // Infinite recursion, does not terminate!
  }
}
```

Avoiding Infinite Recursion

```
function Fib(n: nat): nat
  decreases n  // suggestion for Dafny
{
  if n < 2 then n else Fib(n - 2) + Fib(n - 1)
}
```

```
function Ack(m: nat, n: nat): nat
  decreases m, n  // tuples can also be used
{
  if m == 0 then n + 1
  else if n == 0 then Ack(m - 1, 1)
  else Ack(m - 1, Ack(m, n - 1))
}
```

```
function SeqSum(s: seq<int>, lo: nat, hi: nat): int
  requires 0 <= lo <= hi <= |s|
  decreases hi - lo  // complex expressions can be used!
{
  if lo == hi then 0 else s[lo] + SeqSum(s, lo + 1, hi)
}
```

Exercises

Exercise: Write a `decreases` clause that proves the termination of the following function:

```
function F(x: int): int {
  if x < 10 then x else F(x - 1)
}
```

Exercise: Write a `decreases` clause that proves the termination of the following function:

```
function G(x: int): int {
  if 0 <= x then G(x - 2) else x
}
```

Exercise: Write a `decreases` clause that proves the termination of the following function:

```
function H(x: int): int {
  if x < -60 then x else H(x - 1)
}
```

Exercises [2]

Exercise: Write a `decreases` clause that proves the termination of the following function:

```
function I(x: nat, y: nat): int {
  if x == 0 || y == 0 then 12
  else if x % 2 == y % 2 then I(x - 1, y)
}
```

Exercise: Write a `decreases` clause that proves the termination of the following function:

```
function I(x: nat, y: nat): int {
  if x == 0 || y == 0 then 12
  else if x % 2 == y % 2 then I(x - 1, y)
}
```

Exercises [3]

Exercise: Write a `decreases` clause that proves the termination of the following function:

```
function J(x: nat, y: nat): int {
  if x == 0 then y
  else if y == 0 then J(x - 1, 3)
  else J(x, y - 1)
}
```

Exercise: Write a `decreases` clause that proves the termination of the following function:

```
function K(x: nat, y: nat, z: nat): int {
  if x < 10 || y < 5 then x + y
  else if z == 0 then K(x - 1, y, 5)
  else K(x, y - 1, z - 1)
}
```

Exercises [4]

Exercise: Write a `decreases` clause that proves the termination of the following function:

```
function L(x: int): int {
  if x < 100 then L(x + 1) + 10 else x
}
```

Exercise: The following function computes the *Hofstadter G sequence*:

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

Find an appropriate `decreases` clause to prove that G terminates.

Termination Metric

Definition 9: *Termination metrics* in Dafny, which are declared by `decreases` clauses, are lexicographic tuples of expressions. At each *recursive* (or mutually recursive) call to a function or method, Dafny checks that the effective `decreases` clause of the callee is *strictly smaller* than the effective `decreases` clause of the caller.

- *Termination metrics* do not have to be natural numbers.
- Any set of values with a *well-founded order* can be used.
- An order $\succ$ is well-founded when:
  - $\succ$ is *irreflexive*: $a \succ a$ never holds
  - $\succ$ is *transitive*: if $a \succ b$ and $b \succ c$ then $a \succ c$
  - there is *no infinite descending chain*: $a_0 \succ a_1 \succ a_2 \succ \ldots$

Well-Founded Orders in Dafny

Type	$X \succ y$ ("X decreases to y") iff...
`bool`	$X \wedge \neg y$ (`true` decreases to `false`)
`int`	$(y < X) \wedge (0 \leq X)$ (note: *negative* integers are *not ordered*)
`real`	$(y \leq X - 1.0) \wedge (0.0 \leq X)$ (note: *negative* reals are *not ordered*)
`set<T>`	y is a proper subset of X (*e.g.* `{a,b,c} > {a,c}`)
`seq<T>`	y is a consecutive proper sub-sequence of X (*e.g.*, `[a,b,c] > [b,c]`)
inductive datatypes	y is structurally included in X (*e.g.* `Cons(42, Cons(5, Nil)) > 5`)

Exercises

Exercise: Write a `decreases` clause that proves the termination of the following function:

```
function M(x: int, b: bool): int {
  if b then x else M(x + 25, true)
}
```

Exercise: Write a `decreases` clause that proves the termination of the following function:

```
function N(x: int, y: int, b: bool): int {
  if x <= 0 || y <= 0 then
    x + y
  else if b then
    N(x, y + 3, !b)
  else
    N(x - 1, y, true)
}
```

Lexicographic Tuples

Definition 10: The *lexicographic order* on tuples is a component-wise comparison where earlier components are treated as more significant.

Examples:

- `4, 12 > 4, 2 > 3, 5256` (first component is more significant)
- `4, 12 > 4, 12, 365, 0` (*shorter* tuples exceed, *if prefixes are equal*)
- `2, 5 > 1` (prefix `2` exceeds `1`)
- `12, true, 1.9 > 12, false, 57.3`

Exercise: Determine if the first tuple exceeds the second.

1. `2,5` ☐ `1,7`
2. `1,7` ☐ `7,1`
3. `5,0,8` ☐ `4,93`
4. `4,9,3` ☐ `4,93`
5. `4,93` ☐ `4,9,3`
6. `3` ☐ `2,9`
7. `true,80` ☐ `false,66`
8. `true,2` ☐ `19,1`
9. `4,true,50` ☐ `4,false,800`
10. `7.0,{3,4,9},false,10` ☐ `7.0,{3,9},true,10`

Ackermanm Function

```
function Ack(m: nat, n: nat): nat
  decreases m, n
{
  if m == 0 then
    n + 1
  else if n == 0 then
    Ack(m - 1, 1)
  else
    Ack(m - 1, Ack(m, n - 1))
}
```

On each recursive call, the ordered pair (m, n) decreases.

Mutually Recursive Functions

```
method StudyPlan(n: nat)
  requires n <= 40
  decreases 40 - n
{
  if n == 40 { /* done */ }
  else {
    var hours := RequiredStudyTime(n);
    Learn(n, hours);
  }
}

method Learn(n: nat, h: nat)
  requires n < 40
  decreases 40 - n, h
{
  if h == 0 { StudyPlan(n + 1); }
  else { Learn(n, h - 1); }
}
```

Call	Proof obligation for termination
`StudyPlan` calls `Learn`	$40 - n \succ (40 - n, h)$ ($40 - n$ is a proper prefix of $(40 - n, h)$)
`Learn` calls `StudyPlan`	$(40 - n, h) \succ 40 - (n + 1)$ (first component is decreased)
`Learn` calls `Learn`	$(40 - n, h) \succ (40 - n, h - 1)$ (second component is decreased)

Exercises

Exercise: Add `decreases` clauses to prove termination of the following program.

```
method Outer(a: nat) {
  if a != 0 {
    var b := RequiredStudyTime(a - 1);
    Inner(a, b);
  }
}

method Inner(a: nat, b: nat)
  requires 1 <= a
{
  if b == 0 {
    Outer(a - 1);
  } else {
    Inner(a, b - 1);
  }
}
```

Exercises [2]

Exercise: Add `decreases` clauses to prove termination of the following program.

```
method Outer(a: nat) {
  if a != 0 {
    var b := RequiredStudyTime(a - 1);
    Inner(a - 1, b);
  }
}

method Inner(a: nat, b: nat) {
  if b == 0 {
    Outer(a);
  } else {
    Inner(a, b - 1);
  }
}
```

Loops

Loops in Dafny

```
while G
  decreases M
  invariant J
{
  Body
}
```

- G is the *loop guard*, a Boolean expression
- M is the *termination measure*, an expression that *decreases* in each iteration
- J is the *loop invariant*, a condition that *holds* in each iteration

Note: While loops are *opaque*, they are always abstracted by their invariant.

```
...
while G
  invariant J
  // Look ma, no body!
...
```

Examples

```
while x < 300
  invariant x % 2 == 0

while x % 2 == 1
  invariant 0 <= x <= 100

x := 2;
while x < 50
  invariant x % 2 == 0
// After the loop, the invariant and
// the negation of the guard hold:
assert x >= 50 && x % 2 == 0;

x := 0;
while x % 2 == 0
  invariant 0 <= x <= 20
assert x == 19; // not provable!
```

Attaining Equality

```
i := 0;
while i != 100
  invariant 0 <= i <= 100
assert i == 100;
```

Assertion is provable from just the negation of the guard.

```
i := 0;
while i < 100
  invariant 0 <= i <= 100
assert i == 100;
```

Assertion requires the invariant *and* the negation of the guard to hold.

Attaining Equality [2]

```
i := 0;
while i != 100
  invariant true  // note!
assert i == 100;
```

Assertion is provable from just the negation of the guard.

```
i := 0;
while i < 100
  invariant true  // note!
assert i == 100;  // not provable
```

Assertion requires the invariant *and* the negation of the guard to hold.

Relations Between Variables

```
x, y := 0, 0;
while x < 300
  invariant 2 * x == 3 * y;
assert 200 <= y;
```

```
x, y := 0, 191;
while !(y < 7)
  invariant 0 <= y && 7 * x + y == 191
assert x == 191 / 7 && y == 191 % 7;
```

```
n, s := 0, 0;
while n != 33
  invariant s == n * (n - 1) / 2
assert s == 33 * 32 / 2;
```

Hoare Triples for Loops

```
// { J }
while G
  invariant J
// { J && !G }
```

Example:

```
r := 0;
N := 104;
while (r+1)*(r+1) <= N
  invariant 0 <= r && r*r <= N
assert 0 <= r && r*r <= N < (r+1)*(r+1);
```

Floyd-Hoare Logic for Loop Body

To prove the *partial* correctness of a loop

```
while G
  invariant J
{
  Body
}
```

we need to prove the validity of

```
// { J && G }
Body
// { J }
```

Example: Quotient Modulus

Let's write a program (using a loop) that computes the quotient and modulus of 191 and 7.

Here is the specification:

```
x, y := 0, 191;
while !(y < 7)
  invariant 0 <= y && 7*x + y == 191
{
  // { 0 <= y && 7*x + y == 191 && 7 <= y }
  // ... body?
  // { 0 <= y && 7*x + y == 191 }
}
assert x == 191 / 7 && y == 191 % 7;
```

Example: Quotient Modulus [2]

```
x, y := 0, 191;
while !(y < 7)
  invariant 0 <= y && 7*x + y == 191
{
  // { 0 <= y && 7*x + y == 191 && 7 <= y }
  // { 0 <= y - 7 && 7*x + 7 + (y - 7) == 191 }
  y := y - 7;
  // { 0 <= y && 7*x + 7 + y == 191 }
  // { 0 <= y && 7*(x + 1) + y == 191 }
  x := x + 1;
  // { 0 <= y && 7*x + y == 191 }
}
assert x == 191 / 7 && y == 191 % 7;
```

Leap to the Answer

There's more than one way to implement the loop body for the quotient-modulus program.

```
x, y := 0, 191;
while !(y < 7)
  invariant 0 <= y && 7*x + y == 191
{
  // { 0 <= y && 7*x + y == 191 && 7 <= y }
  // { true }
  // { 0 <= 2 && 7*27 + 2 == 191 }
  x, y := 27, 2;
  // { 0 <= y && 7*x + y == 191 }
}
assert x == 191 / 7 && y == 191 % 7;
```

Going Twice as Fast

Let's dwell on this first program with a loop body some more, to consider something that *does not work*. How about we try to combine two loop bodies into one? Instead of incrementing x by 1 and decrementing y by 7, let's try incrementing x by 2 and decrementing y by 14.

```
// { 0 <= y && 7*x + y == 191 && 7 <= y }
// { 14 <= y && 7 * x + 2 == 191 } -- error: does not follow from above!
// { 0 <= 14 - y && 7*(x + 2) + (y - 14) == 191 }
x, y := x + 2, y - 14;
// { 0 <= y && 7*x + y == 191 }
```

Here, `14 <= y` does not follow from the top line, so this loop body is not correct.

Exercise: Introduce an `if` statement in the body of the loop, where one branch is `x, y := x + 2, y - 14` and the other is `x, y := x + 1, y - 7`. What guard condition do you need in the `if` statement to make the loop correct?

Loop Termination

To prove the *total* correctness of a loop

```
while G
  invariant J
  decreases D
{
  Body
}
```

we also need to prove the validity of

```
// { J && G }
ghost var d := D;
Body
// { d > D }
```

Note: `ghost` variables are for reasoning only, they are not part of the compiled code.

Termination of the Quotient-Modulus Program

```
var x, y := 0, 191;
while 7 <= y
  invariant 0 <= y && 7 * x + y == 191
  decreases y
{
  y := y - 7;
  x := x + 1;
}

// { 0 <= y && 7 * x + y == 191 && 7 <= y }
ghost var d := y;
y := y - 7;
x := x + 1;
// { d > y && d >= 0 }
//   -- {d > y} follows from y := y - 7
//   -- {d >= 0} follows from 0 <= y in invariant
```

Quick Body

```
var x, y := 0, 191;
while 7 <= y
  invariant 0 <= y && 7 * x + y == 191
  decreases y
{
  y := 2;
  x := 27;
}

// { 0 <= y && 7 * x + y == 191 && 7 <= y }
ghost var d := y;
y := 2;
x := 27;
// { d > y && d >= 0 }
//   -- {d > y} follows from 7 <= y in invariant
//   -- {d >= 0} follows from 0 <= y in invariant
```

Default `decreases` Clauses for Loops in Dafny

If the loop guard is an arithmetic comparison of the form `E < F` or `E <= F`, then the default is

```
decreases F - E
```

If the loop guard is an arithmetic comparison of the form `E != F`, then the default is the absolute difference between `E` and `F`:

```
decreases if E < F then F - E else E - F
```

Complete Loop Rule

```
// { J }
while G
  invariant J
  decreases D
{
  Body
}
// { J && !G }

// { J && G }
ghost var d := D;
Body
// { J && d > D }
```

Computing Sums

```
while n != 33
  invariant s == n * (n - 1) / 2
{
  // { s == n * (n - 1) / 2 && n != 33 }
  // { s == n * (n - 1) / 2 }
  s := s + n;
  // { s = n * (n - 1) / 2 + n }
  // { s = (n*n − n) / 2 + 2*n / 2 }
  // { s == (n*n − n + 2*n) / 2 }
  // { s == (n*n + n) / 2 }
  // { s == (n + 1) * n / 2 }
  // { s == (n + 1) * (n + 1 - 1) / 2 }
  n := n + 1;
  // { s == n * (n - 1) / 2 }
}
assert s == 33 * 32 / 2;
```

Full Program

Need to choose initial values of `n` and `s` to establish invariant.

```
n, s := 0, 0;
while n != 33
  invariant s == n * (n - 1) / 2
{
  s := s + n;
  n := n + 1;
}
```

Exercise: Write a different (but still correct) initializing assignment for the loop above.

Exercise: Write an initializing assignment and a loop implementation for the following loop specifications:

```
x := ???;
while x < 300
  invariant x % 2 == 0
{ ??? }
```

```
x := ???;
while x % 2 == 1
  invariant 0 <= x <= 100
{ ??? }
```

Full Program [2]

Exercise: Consider the following program fragment:

```
x := 0;
while x < 100
{
  x := x + 3;
}
assert x == 102;
```

Write a loop invariant that holds initially, is maintained by the loop body, and allows you to prove the assertion after the loop.

Integer Square Root

Definition 11 (Loop design technique 1 — *"Omit a conjuct"*): For a post-condition `A && B`, use `A` as the invariant and `!B` as the guard. That is, use a loop specification:

```
while !B
  invariant A
```

```
method SquareRoot(N: nat) returns (r: nat)
  ensures r*r <= N && N < (r+1)*(r+1)
{
  r := 0;
  while (r+1)*(r+1) <= N
    invariant r*r <= N
  { r := r + 1; }
}
```

More Efficient Algorithm

Rather than calculate `(r + 1)*(r + 1)` on each iteration, add a new variable `s` and maintain the invariant

```
s == (r + 1)*(r + 1)
```

Then we have `s` initially 1, loop guard `s <= N` and invariant `s == (r + 1)*(r + 1)`.

```
// { s == (r + 1)*(r + 1) }
// { s + 2*r + 3 == (r + 1)*(r + 1) + 2*r + 3 }
s := s + 2*r + 3;
// { s == (r + 1)*(r + 1) + 2*r + 3}
// { s == r*r + 2*r + 1 + 2*r + 3 }
// { s == r*r + 4*r + 4}
// { s == (r + 1 + 1)*(r + 1 + 1)}
r := r + 1
// { s == (r + 1)*(r + 1) }
```

Full Program

```
method SquareRoot(N: nat) returns (r: nat)
  ensures r*r <= N < (r+1)*(r+1)
{
  r := 0;
  var s := 1;
  while s <= N
    invariant r*r <= N
    invariant s == (r+1)*(r+1)
  {
    s := s + 2*r + 3;
    r := r + 1;
  }
}
```

Advanced Dafny

Dafny Verification Architecture

Dafny compiles your program *and* its proof obligations into a single pipeline:

Stage	What happens	Tool
1. Parse + type-check	Syntax, types, well-formedness	Dafny frontend
2. Generate VCs	Verification conditions from specs + code	Boogie
3. Discharge VCs	Check each VC via SMT	Z3
4. Compile	Generate target code (C#, Java, Go, *etc.*)	Dafny backend

Key insight: Dafny translates your program into *Boogie* (an intermediate verification language), which generates *verification conditions* (VCs) — first-order formulas that Z3 checks. If all VCs are valid, the program is correct.

Algebraic Datatypes

Dafny supports *inductive datatypes* — algebraic types like those in Haskell or OCaml.

Example:

```
datatype Tree<T> = Leaf | Node(left: Tree<T>, value: T, right: Tree<T>)

function Size<T>(t: Tree<T>): nat {
  match t
  case Leaf => 0
  case Node(l, _, r) => 1 + Size(l) + Size(r)
}

function Mirror<T>(t: Tree<T>): Tree<T> {
  match t
  case Leaf => Leaf
  case Node(l, v, r) => Node(Mirror(r), v, Mirror(l))
}
```

Algebraic Datatypes [2]

Pattern matching + structural recursion = Dafny can automatically verify termination of functions over datatypes via structural descent.

Lemmas

Lemmas are ghost methods — they exist only for the proof, not in the compiled code.

Example:

```
lemma MirrorMirror<T>(t: Tree<T>)
  ensures Mirror(Mirror(t)) == t
{
  match t
  case Leaf =>
  case Node(l, v, r) =>
    MirrorMirror(l);
    MirrorMirror(r);
}
```

The lemma proves that mirroring twice gives back the original tree, by structural induction.

Lemma = proof by induction. Each recursive call to the lemma corresponds to applying the induction hypothesis on a smaller substructure.

Calc Blocks: Structured Proofs

Dafny's `calc` statement lets you write *equational proofs* step by step:

Example:

```
lemma SumFormula(n: nat)
  ensures Sum(n) == n * (n + 1) / 2
{
  if n == 0 {
  } else {
    SumFormula(n - 1);
    calc {
      Sum(n);
    ==  // by definition of Sum
      Sum(n - 1) + n;
    ==  // by induction hypothesis
      (n - 1) * n / 2 + n;
    ==  // algebra
      n * (n + 1) / 2;
    }
```

Calc Blocks: Structured Proofs [2]

```
  }
}
```

Note: Each step in a `calc` block is verified by Z3. You only need to provide enough intermediate steps for Z3 to "connect the dots."

Arrays and Framing

Arrays in Dafny are *heap-allocated* mutable objects. Verification requires *framing* — specifying which memory locations a method may read or modify.

Example:

```
method Swap(a: array<int>, i: nat, j: nat)
  requires i < a.Length && j < a.Length
  modifies a
  ensures a[i] == old(a[j]) && a[j] == old(a[i])
  ensures forall k :: 0 <= k < a.Length && k != i && k != j
            ==> a[k] == old(a[k])
{
  var tmp := a[i];
  a[i] := a[j];
  a[j] := tmp;
}
```

Arrays and Framing [2]

The frame problem: `modifies a` declares that `Swap` may change array `a`. The last `ensures` clause explicitly states that *all other elements are unchanged* — Dafny does not infer this automatically.

Sequences

Dafny also has *immutable sequences* `seq<T>` — useful for specifications:

Sequence operations:

- `|s|` — length
- `s[i]` — element access
- `s[i..j]` — slicing
- `s + t` — concatenation
- `x in s` — membership
- `s[i := v]` — update (returns new seq)

Ghost vs. compiled:

- Sequences are *value types* (immutable).
- Often used in `ensures` and `invariant` clauses.
- Can be used in executable code, but arrays are more efficient at runtime.
- Common pattern: specify with `seq`, implement with `array`.

Example:

```
predicate Sorted(s: seq<int>) {
  forall i, j :: 0 <= i < j < |s| ==> s[i] <= s[j]
}
```

Verified Sorting: Insertion Sort

Example:

```
predicate Sorted(s: seq<int>) {
  forall i, j :: 0 <= i < j < |s| ==> s[i] <= s[j]
}

method InsertionSort(a: array<int>)
  modifies a
  ensures Sorted(a[..])
  ensures multiset(a[..]) == multiset(old(a[..]))
{
  for i := 1 to a.Length
    invariant Sorted(a[..i])
    invariant multiset(a[..]) == multiset(old(a[..]))
  {
    var j := i;
    while j > 0 && a[j - 1] > a[j]
      invariant 0 <= j <= i
      invariant Sorted(a[..j]) && Sorted(a[j..i+1])
```

Verified Sorting: Insertion Sort [2]

```
      invariant multiset(a[..]) == multiset(old(a[..]))
    {
      a[j - 1], a[j] := a[j], a[j - 1];
      j := j - 1;
    }
  }
}
```

Two postconditions:

1. `Sorted(a[..])` — the result is sorted.
2. `multiset(a[..]) == multiset(old(a[..]))` — it is a *permutation* of the input (no elements lost or duplicated).

Verified Sorting: Why Two Postconditions?

A sorting algorithm must satisfy *two* properties:

Sortedness alone is not enough:

```
// "Sorts" by replacing everything with 0
method BadSort(a: array<int>)
  modifies a
  ensures Sorted(a[..])
{
  for i := 0 to a.Length {
    a[i] := 0;  // Sorted ✓, but wrong!
  }
}
```

Permutation alone is not enough:

```
// "Sorts" by doing nothing
method LazySort(a: array<int>)
  modifies a
  ensures multiset(a[..]) ==
multiset(old(a[..]))
{
  // no-op: permutation ✓, but not sorted!
}
```

Note: Formal verification requires stating *all* aspects of correctness. A permutation-only specification allows "sorts" that don't actually sort.

Exercises: Advanced Dafny

1. Write a Dafny function `Flatten<T>(t: Tree<T>): seq<T>` that returns the in-order traversal of a tree as a sequence. Verify that `|Flatten(t)| == Size(t)`.

2. Prove the following lemma about sequences:

   ```
   lemma AppendAssoc<T>(a: seq<T>, b: seq<T>, c: seq<T>)
     ensures (a + b) + c == a + (b + c)
   ```

   Hint: Dafny may verify this automatically. If not, use structural induction on `a`.

3. Modify `InsertionSort` to sort in *descending* order. Adjust the specification and loop invariants accordingly.

4. ⋆ Write a verified binary search:

   ```
   method BinarySearch(a: array<int>, key: int) returns (index: int)
     requires Sorted(a[..])
     ensures 0 <= index < a.Length ==> a[index] == key
     ensures index == -1 ==> key !in a[..]
   ```

   Supply the loop invariant so that Dafny verifies the implementation.

Exercises: Advanced Dafny [2]

5. ⋆ Verify a `Reverse` method on arrays:

```
method Reverse(a: array<int>)
  modifies a
  ensures forall i :: 0 <= i < a.Length ==> a[i] == old(a[a.Length - 1 - i])
```

What loop invariant do you need for the swapping loop?

Bibliography

[1] M. Leino and K. Rustan, “Accessible Software Verification with Dafny,” *IEEE Software*, vol. 34, no. 6, pp. 94–97, Nov. 2017, doi: 10.1109/MS.2017.4121212.

[2] C. A. R. Hoare, “An Axiomatic Basis for Computer Programming,” *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969, doi: 10.1145/363235.363259.

[3] R. W. Floyd, “Assigning Meanings to Programs,” *Mathematical Aspects of Computer Science*, vol. 19. American Mathematical Society, pp. 19–32, 1967.