Formal Methods in Software Engineering

Specification and Verification — Spring 2025

Konstantin Chukharev

§1 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.

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

§2 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 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.

§3 Floyd-Hoare Logic

From Contracts to Floyd-Hoare Logic

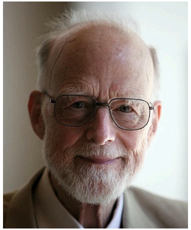
In the design-by-contract methodology, contracts are usually assigned to procedures or modules. In general, it is possible to assign contracts to each statement of a program.

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.

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

Definition 3: A is *stronger* than B if $A \to B$ 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 $B \to A$ 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. $\{\text{true}\} \quad x := 2 * y \quad \{x \text{ is even}\}$
   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:

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

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.

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

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

$$S;T$$

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

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

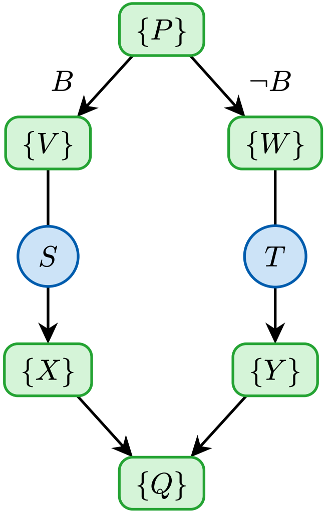
Strongest post-condition:

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

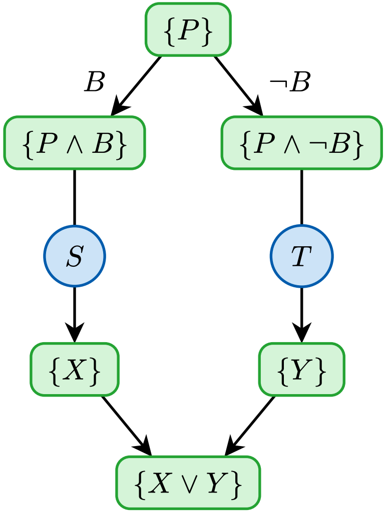
Conditional Control Flow



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

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

Strongest Post-condition for Condition



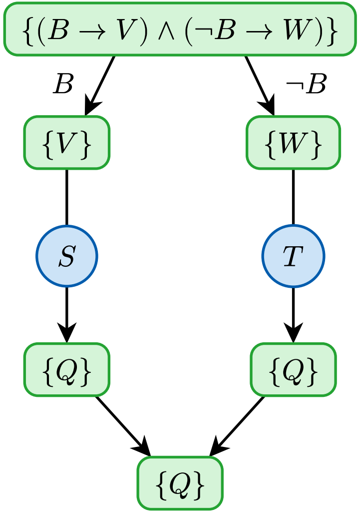
$$\{P\} \quad \text{if } B \text{ then } \{S\} \text{ else } \{T\} \quad \{Q\}$$

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

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

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

Weakest Pre-condition for Condition



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

$$\mathcal{WP}[\![\, \text{if } B \text{ then } \{S\} \text{ else } \{T\}, Q \,]\!] =$$
$$= (B \to V) \land (\neg B \to W) =$$
$$= (B \to \mathcal{WP}[\![\, S, Q \,]\!]) \land (\neg B \to \mathcal{WP}[\![\, T, Q \,]\!])$$

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

$$X = Q$$
$$Y = Q$$

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 }
```

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

Method Correctness

Given

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

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

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:

$$\mathcal{SP}[\![\, \text{assume } E, P \,]\!] = P \land E$$

$$\mathcal{WP}[\![\, \text{assume } E, Q \,]\!] = E \to Q$$

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 $Q = \{x = 48\}$. Backward reasoning:

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

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;
```

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

TODO

- [ ] …

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.