Chapter 2

Making It Formal

This chapter gives the formal underpinnings of what the previous chapter described informally. The meaning of programs is given precisely with mathematical formulas. These are said to give the *semantics* of the program statements.

Throughout this book, the reasoning is based on what is called *Floyd logic*. It uses boolean formulas to describe what is known before and after each statement, which allows us to break down the reasoning about programs into reasoning about individual statements of the program. Along the way, I will define *Hoare triples*, which give a good guide to understanding semantics, and *weakest preconditions*, which give convenient ways to automate program reasoning.

The study of program semantics may seem overly concerned with picky details. Think of it as long multiplication—it prescribes steps for computing something in detail. Once you've applied long multiplication on many problems, you develop a deeper understanding of multiplication. You then realize that some steps can be replaced by shortcuts. More importantly, you can then put multiplication to use in daily problems. It is not the tiny steps of long multiplication that let you solve problems, but it is your familiarity with multiplication that does. Because you can carry out the steps, you know of a way to check something in detail, but in practice you use a calculator or spreadsheet to perform multiplication for you.

Program proving is like that. Carrying out the small steps lets you see how program reasoning is done. If you're unsure of something, you can return to the foundational steps and convince yourself of your program's correctness (or find mistakes). In practice, however, you will typically use an automated program verifier that performs the semantic computations for you.

Let's get on with the details. Program semantics, here we come!

2.0.Program State

A program has several kinds of *variables*. These include the in- and out-parameters of methods and the local variables declared in a method body. The variables that can be used at a point in a program are said to be *in scope* at that program point. The *state* at a program point is a valuation of the variables in scope at that program point.

For illustration, consider the following method:

```
method MyMethod(x: int) returns (y: int)
requires 10 <= x
ensures 25 <= y

{
var a, b;

a := x + 3;

if x < 20 {

b := 32 - x;

} else {

b := 16;

}

y := a + b;

}
```

On entry to this method, that is, at the program point labeled , only the method's in-parameter (`x`) is in scope. Just inside the method body, at program point , both in- and out-parameters (`x` and `y`) are in scope. At program point , after the statement that declares local variables `a` and `b`, the scope consists of `x`, `y`, `a`, and `b`. The same is true at program points through . On exit from the method, at program point , the local variables have gone out of scope so the scope once again consists of just the parameters (`x` and `y`).

The state at program point , which is called the *initial state* of the method, is a valuation of `x`. For example, `x` may be `18` or `37`. If it is `37`, the state at program point is `37` for `x`, `56` for `y`, `40` for `a`, and `16` for `b`. Moreover, the state at , which is called the *final state* of the method, is then `37` for `x` and `56` for `y` . You can determine these states if you trace the program execution through the method body.

> **Exercise 2.0.**
>
> If the initial program state is `18` for `x`, what is the program state at and what is the final program state?

As we saw in the previous chapter, callers of a method are responsible for establishing the method's precondition. Consequently, we know the initial state satisfies the precondition. For example, the initial state cannot be `2` for `x`, because `10 <= x` does not hold if `x` were `2`.

As we also saw in the previous chapter, a method implementation is responsible for establishing the method's postcondition. Consequently, to declare `MyMethod` to be correctly implemented, we must check that `25 <= y` holds in state for every possible trace through the method body, for every possible initial state.

We cannot reasonably determine this if we had to try out traces (like those starting from `x` being `18` or `37`) one at a time. Instead, we need a way to reason about many states at a time. This is done using boolean expressions. More precisely, we use a boolean expression to denote those states that make the boolean expression evaluate to **`true`**. For example, the boolean expression `15 <= x <= 40` represents all initial states where the value of `x` lies in the range `15` through `40`.

Another name for a boolean expression is *logical formula* (or just *formula*) or *predicate*, which simply means an expression that for a given state evaluates to either **`false`** or **`true`**. You can also think of a predicate as *characterizing a set of states*.

Instead of tracing a state through the program, we are going to trace a *predicate* through the program. The main purpose of this chapter is to teach you how that is done. Here are some examples of what we'd like to be able to figure out:

- If `12 <= x && a % 2 == 0` holds at , what can we ssay about the possible states at ?
- If we want to be sure that the state satisfies `a > b` whenever control reaches , what needs to hold at program point ?
- Suppose we write down a predicate at each program point through , write the method precondition at , write the method postcondition at , and then check that each such predicate includes the states reachable from the prior predicates. Does this mean the method implemented correctly?

The first bullet is answered by what's called strongest postconditions, the second bullet by what's called weakest preconditions, and the third bullet is the idea behind Floyd logic. Weakest preconditions are also the basis for what in Part 2 of this book I will often refer to as “working backward”.

2.1.Floyd Logic

Consider this simple program:

```
method MyMethod(x: int) returns (y: int)
requires 10 <= x
ensures 25 <= y

{
var a := x + 3;

var b := 12;

y := a + b;

}
```

where I have labeled the the various program points. Let's write a predicate at each program point. In the initial state, the only thing we know is the method precondition, so let's write it at . In the final state, the only thing we care about is the method postcondition, so let's write it at .

```
10 <= x
25 <= y
```

So, shows what we're given, and shows our proof goal.

Let's trace through the program from the initial state. Whatever is known to hold at is also known to hold at , namely `10 <= x`. Then, what we *know* in each subsequent state is this:

```
10 <= x
10 <= x && a == x + 3
10 <= x && a == x + 3 && b == 12
10 <= x && a == x + 3 && b == 12 && y == a + b
```

I didn't yet tell you how I came up with these predicates, but I hope they look plausible to you. Now, to show that the method body correctly meets the method specification, we must show that the formula at logically implies the formula at . In other words, we must show that

```
10 <= x && a == x + 3 && b == 12 && y == a + b ==> 25 <= y
```

is a *valid* formula, that is, a formula that evaluates to **`true`** for any values of its variables.

> **Exercise 2.1.**
>
> Argue—as rigorously as you can—that the formula is indeed valid.

We filled in the formulas through in the forward direction. In this way, each formula describes the states that can reach that point. Each formula thus speaks of “this is what we've got” or “this is what is known to hold here”.

It is also possible to fill in the formulas in the opposite order. To do that, we start with the *desired* final state and work our way back to the initial state. In this way, each formula speaks of “this is what we want” or “this is what we need to hold here”.

Let's try it. Whatever we want to hold at is also what we want to hold at , namely `25 <= y`. Then, what we *need* in each preceding state is this:

```
25 <= y
25 <= a + b
25 <= a + 12
25 <= x + 3 + 12
```

Now, to show that the method body correctly meets the specification, we must show that the formula at logically implies the formula at . That is, we must show that

```
10 <= x ==> 25 <= x + 3 + 12
```

is a valid formula.

> **Exercise 2.2.**
>
> Rigorously argue that the formula is indeed valid.

The direction in which we fill in the formulas does not matter. What matters is that we decorate each program point with a formula. With the formulas in place, we check, for each statement `S`, that starting in any state satisfying the formula before `S` leads to a state satisfying the formula written after `S`. We also check that the method precondition implies the formula used to decorate the initial state (`==>`) and check that the formula used to decorate the final state implies the method postcondition (`==>`).

This program-reasoning technique was captured by Robert W. Floyd in his ground-breaking paper “Assigning Meanings to Programs” [50] and is therefore known as *Floyd logic*.

2.2.Hoare Triples

We need a notation for the central building block of Floyd logic, which we can summarize as “a statement takes some pre-states to some post-states”. We'll use the triples introduced for this purpose by C. A. R. Hoare [63]. For `P` a predicate on the pre-state of a program `S` and `Q` a predicate on the post-state of `S`, the *Hoare triple*

```
{{ P }} S {{ Q }}
```

says that if `S` is started in any state that satisfies `P`, then `S` will not crash (or do other bad things) and will terminate in some state satisfying `Q`.[1]

For example, the Hoare triple

```
{{ x == 12 }} x := x + 8 {{ x == 20 }}
```

expresses the idea that if the program `x := x + 8` is started in a state where `x` is `12`, then the program is guaranteed to terminate in a state where `x` has the value `20`. This is true about the program `x := x + 8`. That is, we say that this Hoare triple *holds*, or that the Hoare triple is *valid*.

> **Sidebar 2.0**
>
> When discussing program semantics and in various other places of the book, I may omit punctuation required in the Dafny programming language. For example, in the previous paragraph, I spoke of the assignment statement `x := x + 8`. Writing it in Dafny requires a terminating semi-colon, as in `x := x + 8;`.

As another example, the triple

```
{{ x < 18 }} S {{ 0 <= y }}
```

is notation for saying that whenever you start program `S` in a state where `x` is less than `18`, then `S` will terminate in a state where `y` is non-negative. This triple holds if `S` is the program `y := 5` or the program `y := 18 - x`, but it does not hold if `S` is the program `y := x` or the program `y := 2 * (x + 3)`, because these programs are not guaranteed to terminate with a non-negative `y` when started with `x` being less than `18`. Case in point: a state where `x` is `-5` satisfies `x < 18`, and yet the execution of `y := x` or `y := 2 * (x + 3)` from that state leads to a state where `y` is negative.

For brevity, and without any risk of confusion, we usually talk about a program "starting in `P`" or "ending in `Q`" when we mean "the program starts in a state satisfying predicate `P`" or "the program is guaranteed to be crash-free and to terminate in a state satisfying the predicate `Q`".

Exercise 2.3.

Explain rigorously why each of these triples holds:

```
a) {{ x == y }} z := x - y {{ z == 0 }}
b) {{ true }} x := 100 {{ x == 100 }}
c) {{ true }} x := 2 * y {{ x is even }}
d) {{ x == 89 }} y := x - 34 {{ x == 89 }}
e) {{ x == 3 }} x := x + 1 {{ x == 4 }}
f) {{ 0 <= x < 100 }} x := x + 1 {{ 0 < x <= 100 }}
```

Exercise 2.4.

For each of the following triples, find initial values of `x` and `y` that demonstrate that the triple does not hold.

a) {{ **true** }} x := 2 * y {{ y <= x }}

b) {{ x == 3 }} x := x + 1 {{ y == 4 }}

c) {{ **true** }} x := 100 {{ **false** }}

d) {{ 0 <= x }} x := x - 1 {{ 0 <= x }}

Exercise 2.5.

For each of the following triples, come up with some predicate to replace the question mark to make it a Hoare triple that holds. Make your conditions as precise as possible.

a) {{ 0 <= x < 100 }} x := 2 * x {{ ? }}

b) {{ 0 <= x <= y < 100 }} z := y - x {{ ? }}

c) {{ 0 <= x < N }} x := x + 1 {{ ? }}

Exercise 2.6.

For each of the following triples, come up with some predicate to replace the question mark to make it a Hoare triple that holds. Make your conditions as precise as possible.

a) {{ -128 <= x < 0 }} x := 1 - x {{ ? }}

b) {{ 0 <= x <= y < 100 }} y := y - x {{ ? }}

c) {{ x is even && y < 100 }} x, y := y, x {{ ? }}

Exercise 2.7.

For each of the following triples, come up with some predicate to replace the question mark to make it a Hoare triple that holds. Make your conditions as general as possible.

a) {{ ? }} x := 400 {{ x == 400 }}

b) {{ ? }} x := x + 3 {{ x is even }}

c) {{ ? }} x := 65 {{ y <= x }}

Exercise 2.8.

For each of the following triples, come up with some predicate to replace the question mark to make it a Hoare triple that holds. Make your conditions as general as possible.

a) {{ ? }} b := y < 10 {{ b ==> x < y }}

b) {{ ? }} x, y := 2*x, x+y {{ 0 <= x <= 100 && y <= x }}

c) {{ ? }} x := 2*y {{ 10 <= x <= y }}

2.3.Strongest Postconditions and Weakest Preconditions

In Section 2.1, I sketched two approaches for how to fill in, or *derive*, the decorations needed in Floyd logic to prove a program correct. One of these derivations proceeds in the forward direction, which you may think of as the what-we-have approach. The other derivation proceeds in the backward direction, which you may think of as the goal-oriented approach. It's time to make these two approaches more precise and systematic.

In the forward derivation, we construct the postcondition of a statement from a given precondition. There are many predicates that hold after a statement. For example, all of the following Hoare triples are valid:

```
{{ x == 0 }} y := x + 3 {{ y < 100 }}
{{ x == 0 }} y := x + 3 {{ x == 0 }}
{{ x == 0 }} y := x + 3 {{ 0 <= x && y == 3 }}
{{ x == 0 }} y := x + 3 {{ 3 <= y }}
{{ x == 0 }} y := x + 3 {{ true }}
```

As it turns out, if the two triples {{ `P` }} `S` {{ `Q0` }} and {{ `P` }} `S` {{ `Q1` }} are both valid, then so is the triple {{ `P` }} `S` {{ `Q0 && Q1` }}. In a forward derivation, it is then natural to want to compute the strongest—that is, most precise—post-state predicate. The forward derivation that does that computes what is referred to as the *strongest postcondition* of a statement (with respect to a given predicate on the statement's pre-state).

Similarly, there are many predicates on the pre-state of a statement that guarantee that the statement establishes a given post-state predicate. For example, all of the following Hoare triples are valid:

```
{{ x <= 70 }} y := x + 3 {{ y <= 80 }}
{{ x == 65 && y < 21 }} y := x + 3 {{ y <= 80 }}
{{ x <= 77 }} y := x + 3 {{ y <= 80 }}
{{ x*x + y*y <= 2500 }} y := x + 3 {{ y <= 80 }}
{{ false }} y := x + 3 {{ y <= 80 }}
```

If the two triples {{ `P0` }} `S` {{ `Q` }} and {{ `P1` }} `S` {{ `Q` }} are both valid, then so is the triple {{ `P0 || P1` }} `S` {{ `Q` }}, which suggests that we want to compute the weakest—that is, most general—pre-state predicate. The backward derivation that does that computes what is referred to as the *weakest precondition* of a statement (with respect to a given predicate on the statement's post-state).

The first time you see this goal-oriented, backward derivation, you may find it more dicult to understand than the what-we-have, forward derivation. As it turns out, the backward derivation of formulas is actually easier to construct. Here's how you do it. If you have an assignment `x := E` that you want to establish a formula `Q`, then the formula that needs to hold before the assignment is: `Q` in which you replace all occurrences of `x` with `E`. In other words, the most general condition you can write for the question mark in {{ `?` }} `x := E` {{ `Q` }} is `Q` in which you replace all occurrences of `x` with `E`, a formula that I will write using the notation `Qx := E`.

For example, if the assignment `y := a + b` in the program needs to establish the condition `25 <= y` (see the program in Section 2.1), then the formula that must hold before the assignment is `25 <= y` in which you replace `y` by `a + b`, that is, `25 <= a + b`. In the notation of Hoare triples, the most general condition you can write for the question mark in

```
{{ ? }} y := a + b {{ 25 <= y }}
```

is `25 <= a + b`. Easy, right? Similarly, to work out the most general condition you can write for the question mark in

```
{{ ? }} a := x + 3 {{ 25 <= a + 12 }}
```

you take the desired condition, `25 <= a + 12`, and replace `a` with `x + 3`, which gives you `25 <= x + 3 + 12`.

In a functional program (that is, a program written in a functional programming language), a variable is assigned once when the variable is introduced. (In functional programming, this is known as *let binding*.) In an imperative program, the assignment statement allows you to change the value of a variable. Not only that, but the right-hand side of the assignment can refer to the variable itself, which gives a way to update the variable. For example, the statement

```
x := x + 1
```

increments the value of `x` by `1`.

The recipe for computing the weakest precondition also works if the right-hand side of the assignment mentions the variable being updated. For example, we follow the same recipe as before for assignments like `x := x + 1` and `x := 2*x + y` (it is best to read the following triples from right to left):

```
{{ x+1 <= y }} x := x + 1 {{ x <= y }}
{{ 3 * (2*x + y) + 5*y < 100 }} x := 2*x + y {{ 3*x + 5*y < 100 }}
```

2.3.0.Swap

Let's compute weakest preconditions to verify the correctness of a program that swaps the values stored in `x` and `y`:

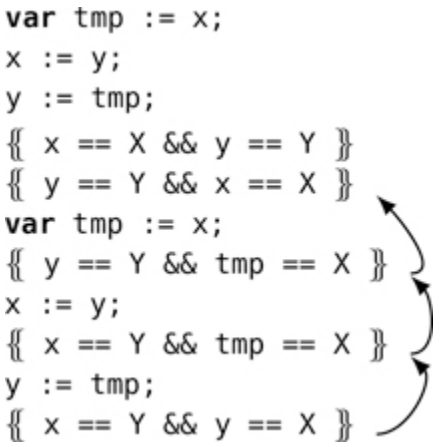


Figure 2.0. To prove that this program fragment swaps the values of `x` and `y`, the reasoning is shown backwards from the desired postcondition. One then has to show that the first annotation implies the second.

To specify what we expect this program to do, we need a way in the postcondition to refer to the initial values of `x` and `y`. When we use Hoare triples, this feat is accomplished by introducing *logical variables*, that is, variables that stand for some values in our proof, but that cannot be used in the

program. Using `X` and `Y` as logical variables, we write

```
{{ x == X && y == Y }}
var tmp := x;
x := y;
y := tmp;
{{ x == Y && y == X }}
```

This program consists of three statements, sequentially composed. To compute weakest preconditions for a sequence of statements, we simply do them one at the time, as is suggested by the following chaining sequence of Hoare triples:

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

In this sequence, we intend to use weakest preconditions to compute predicates for the question marks. When we are done, the first question mark will have been replaced by the weakest precondition of the sequence of statements. We then need to prove that the given precondition implies the weakest precondition. Here we go—one at a time, from back to front, see Figure 2.0.

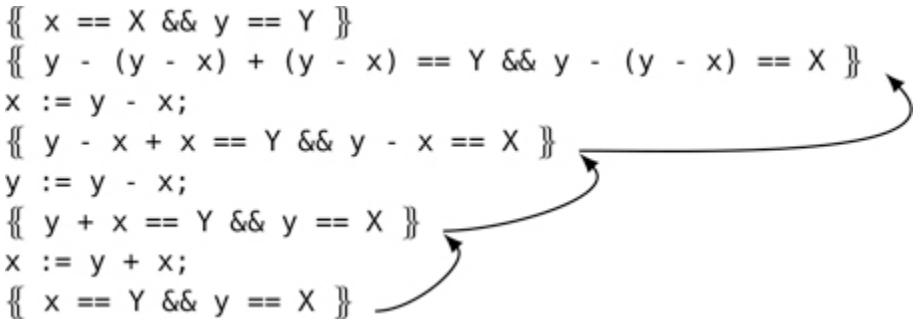


Figure 2.1. Backward reasoning for proving the correctness of this program for swapping integers.

To come up with these intermediate formulas, we just apply the recipe for computing weakest preconditions. It's like when we do long multiplication—we systematically apply the steps we've learned. Next, we need to do a proof of the logical implication

```
x == X && y == Y ==> y == Y && x == X
```

In this case, the implication holds trivially. So, we have proved that the program above does swap `x` and `y`.

2.3.1.A notation for program-proof bookkeeping

Let's repeat the exercise of verifying that a program correctly swaps, but for a different program. This time, we assume `x` and `y` to hold integer values:

```
{{ x == X && y == Y }}

x := y - x;

y := y - x;

x := y + x;

{{ x == Y && y == X }}
```

Constructing weakest preconditions, we get what is shown in Figure 2.1.

Filling in the missing parts in the order suggested by the arrows, all we did was perform substitutions according to the recipe for constructing weakest preconditions. Now, we have a proof obligation:

```
x == X && y == Y

==>

y - (y - x) + (y - x) == Y && y - (y - x) == X
```

After some arithmetic simplifications, we see that this implication holds, so the program correctly swaps.

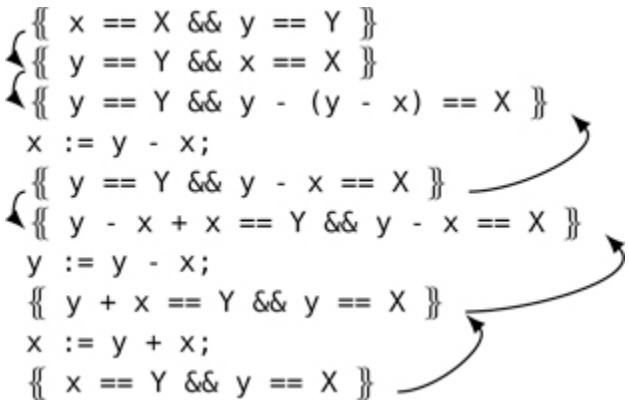


Figure 2.2. Consecutive annotations in this program fragment show simplifications of the formulas. The formula in one such annotation implies the formula in the next annotation. I find such simplifications easiest to read in the forward direction (connected by implication). In contrast, annotations around a program statement are constructed backwards (using weakest preconditions) as shown by the arrows.

When we construct weakest preconditions like we did, it helps to simplify our formulas as we go along, rather than doing so only at the end. A convenient notation for this is to, working backward, write the simplified formula on the line above the one we just wrote. If we perform several

simplifications, we can write several formulas above each other. If we want to, we are also allowed to strengthen the formula as we continue working backward. (Beware! A common mistake is to *weaken* the formula as you go backward. This is not allowed, because it will not give you a proof.) In the end, our derivation shows a sequence of formulas of program statements. For every pair of consecutive formulas, we need to prove that the former implies the latter, and every program statement between two formulas must be a valid Hoare triple.

Figure 2.2 shows the previous program with intermediate simplifications, as just described. The arrows on the right show constructions of weakest preconditions, thus forming valid Hoare triples. The arrows on the left connect consecutive annotations. These may be constructed in the backward direction as strengthenings, but many people are more conditioned to check them as implications in the forward direction. Therefore, I'm showing the arrows on the left in the forward direction.

> **Exercise 2.9.**
>
> Verify that the following program correctly swaps, where ^ denotes xor:
>
> ```
> x := x ^ y;
> y := x ^ y;
> x := x ^ y;
> ```
>
> You can use the commutativity and associativity of xor, as well as the properties `x ^ x == 0` and `x ^ 0 == x`.

> **Exercise 2.10.**
>
> Find the error in the following proof attempt:
>
> ```
> {{ x == 0 }}
> {{ x == 0 && y == 6 }}
> x := x + 2
> {{ x == 2 && y == 6 }}
> {{ x + y == 8 }}
> y := x + y
> {{ y == 8 }}
> ```

2.3.2.Simultaneous assignments

Some languages allow several variables to be assigned in one statement. This is called *simultaneous assignment*. For example,

```
x, y := 10, 3;
```

sets `x` to `10` and, at the same time, sets `y` to `3`. As another example,

```
x, y := x + y, x - y;
```

computes the sum of `x` and `y` and the difference between `x` and `y`, and then updates `x` and `y` to these,

respectively. Note that all of the right-hand sides are computed before any variable is assigned. This is what makes the assignment “simultaneous”, and it is what distinguishes it from a *sequence* of two assignments:

```
x := x + y; y := x - y;
```

The semantics of simultaneous assignment is no more dicult than an assignment to a single variable. All we need is for the weakest-precondition substitution to happen simultaneously as well. More precisely, for an assignment `x, y := E, F` to establish a condition `Q`, the condition we need before the assignment is `Q[x, y := E, F]`, that is, `Q` in which we simultaneously substitute `E` for `x` and `F` for `y`.

Simultaneous assignment gives us a ridiculously easy way to write a program for swapping the values of two variables: `x, y := y, x`. Here is its proof:

```
{{ x == X && y == Y }}

{{ y == Y && x == X }}

x, y := y, x

{{ x == Y && y == X }}
```

> **Exercise 2.11.**
>
> Fill in the question marks in the following Hoare triples. Simplify the formulas you obtain by computing weakest preconditions.
>
> a) `{{ ? }} x, y := 6, 7 {{ x < 10 && y <= z }}`
>
> b) `{{ ? }} x, y := x + 1, 2 * x {{ y - x == 3 }}`
>
> c) `{{ ? }} x := x + 1; y := 2 * x {{ y - x == 3 }}`

2.3.3. Variable introduction

Local variables are introduced with the **`var`** statement. Most often, we use it together with an assignment, like **`var`** `tmp := x` in Section 2.3.0. So far in treating the semantics, I have ignored the actual variable introduction (that is, the “**`var`**” part) and focused on the assignment. But variable introduction also has semantics. To talk about it, we should think of **`var`** as being separate from any initial assignment that shares the syntax. That is, **`var`** `tmp := x` is really two statements:

```
var tmp;

tmp := x;
```

Every variable has a type, but the type of a local variable is usually inferred, so you may not see it explicitly in the program.

The statement **`var`** `x` introduces `x`, but does not promise anything about the initial value of `x` (other than that `x` will have some value of its type). Thus, in order for **`var`** `x` to establish a predicate `Q`, what we need before the variable introduction is that `Q` hold for all values of `x`. Written as a Hoare triple, we have

```
{{ forall x :: Q }} var x {{ Q }}
```

For example, for an integer variable x, suppose `Q` is `0 <= x < 100`. Then, the following is a valid Hoare triple:

```
{{ forall x :: 0 <= x < 100 }} var x {{ 0 <= x < 100 }}
```

The condition `forall x :: 0 <= x < 100` is just **`false`**, because the condition `0 <= x < 100` does not hold universally of all integers (case in point: `102`). So, this says that `var x` in no way guarantees that `x` will end up in the range from `0` to `100`.

Generally, if `Q` mentions `x` in any nontrivial way, then there's no way to prove that `var x` will establish `Q`. But don't go thinking that variable introduction is never useful—the semantics just properly captures the fact that uninitialized variables can have any value. If we want to depend on what value the variable has, then we had better first assign to it.

As an example, let's prove that we can introduce a local variable to temporarily store a value we intend to assign to another variable. We know from the weakest precondition of assignment that `x := E` will establish `Q` provided the statement is started in `Q[x:= E]`. When will `var tmp := E; x := tmp` establish `Q`, where `tmp` is a variable that is not used in either `E` or `Q`? The following chain of Hoare triples show the answer to be `Q[x := E]`, just as for the direct assignment `x := E`.

```
{{ Q[x := E] }}

{{ forall tmp :: Q[x := E] }}

var tmp

{{ Q[x := E] }}

{{ Q[x := tmp][tmp := E] }}

tmp := E

{{ Q[x := tmp] }}

x := tmp

{{ Q }}
```

As usual, the lines above are best read from bottom to top. The simplification from `Q[x := tmp][tmp := E]` to `Q[x := E]` is justified by the fact that `tmp` does not occur in `Q`. The simplification from

```
forall tmp :: Q[x := E]
```

to `Q[x := E]` is justified by the fact that `tmp` does not occur in the expression `Q[x := E]`.

> **Exercise 2.12.**
>
> Here is an example that does not depend on the initial value of `x`. Prove:
>
> ```
> {{ true }} var x; x := x * x {{ 0 <= x }}
> ```

2.3.4.The complicated one

Okay, it wouldn't be fair not to show you how to compute the strongest postcondition for an assignment, even if we won't use it much. It is not the simple substitution we had for weakest preconditions. It's complicated. Brace yourself.

To motivate the recipe for strongest postconditions, consider the following, innocent-looking Hoare triple:

```
{{ w < x < y }} x := 100 {{ ? }}
```

It's clear that `x == 100` is a postcondition, but is it the strongest? The condition `w < x < y` no longer holds, because the assignment overwrites the previous value of `x`. However, in the post-state, there exists some value x_0 (namely, the value of `x` in the pre-state) for which `w <` x_0 `< y` holds. So, the most precise condition we can write for the question mark is:

exists x_0 :: w < x_0 < y && x == 100

Since the variables involved are integers, this formula is logically equivalent to

```
w + 1 < y && x == 100
```

In general, the right-hand side of the assignment may also mention `x`, so we'll need to use the x_0 there, too. So here it is:

{{ P }} x := E {{ **exists** x_0 :: P[x := x_0] && x == E[x := x_0] }}

Aren't you glad you asked? 

> **Exercise 2.13.**
>
> Replace the ? in the following Hoare triples by computing strongest postconditions. In each case, simplify the formula, if possible.
>
> a) {{ y == 10 }} x := 12 {{ ? }}
>
> b) {{ 98 <= y }} x := x + 1 {{ ? }}
>
> c) {{ 98 <= x }} x := x + 1 {{ ? }}
>
> d) {{ 98 <= y < x }} x := 3 * y + x {{ ? }}

> **Exercise 2.14.**
>
> Verify the correctness of the various swap programs in Section 2.3.0 using strongest postconditions.

2.4. and

We can save many words by introducing some notation. Suppose `S` is a statement. Then, for any predicate `P` on the pre-state of `S`, let's define `S, P` to denote the strongest postcondition of `S` with respect to `P`. Similarly, for any predicate `Q` on the post-state of `S`, let's define `S, Q` to denote the weakest precondition of `S` with respect to `Q`.

2.4.0.Working backward

Using this notation, the semantics of assignment is defined as follows:

```
x := E, Q=Q[x := E]
x := E, P=exists x_0 :: P[x := x_0] && x == E[x := x_0]
```

These and equations apply for assignments to one variable as well as simultaneous assignment to several variables, as long as `x`, x_0, and `E` are lists of the same length.

It is very useful to *work backward* over an assignment statement, because it lets you convert a condition `Q` that you want to hold after the assignment into a condition that needs to hold prior to the assignment. I will make frequent use of this step in Part 2.

> **Exercise 2.15.**
>
> Practice working backward: Compute the weakest precondition of the following programs with respect to the postcondition `x + y <= 100`.
>
> ```
> a)x := 20       b)x := x + 1
> c)x := 2 * x    d)x := -x
> e)x := y        f)x := x + y
> g)x := y - x    h)x := x - y
> i)z := x + y
> ```
>
> In each case, simplify your answer.

> **Exercise 2.16.**
>
> Compute the strongest postcondition of the following programs with respect to the precondition `x + y <= 100`.
>
> ```
> a)x := 5        b)x := x + 1
> c)x := 2 * y    d)z := x + y
> ```
>
> In each case, simplify your answer.

2.4.1.Local variables

In Section 2.3.3, I presented the semantics of **`var`** in terms of weakest preconditions. Here is its semantics in terms of both and :

```
var x, Q=forall x :: Q
var x, P=exists x :: P
```

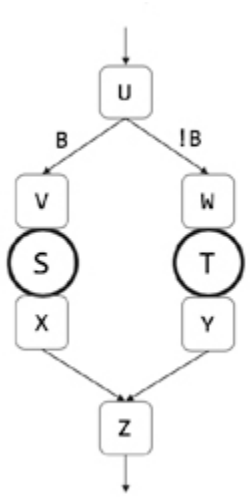
> **Exercise 2.17.**
>
> Compute
>
> a) `var x, x <= 100`
>
> b) `var x, x <= 100`
>
> In each case, simplify your answer.

2.5.Conditional Control Flow

To learn what Floyd logic says about conditional statements, like

```
if B { S } else { T }
```

it is instructive to draw the control-flow graph. Here, I'm using `U`, `V`, `W`, `X`, `Y`, and `Z` to denote predicates at the beginning and end of the conditional statement and of its two substatements, `S` and `T`:



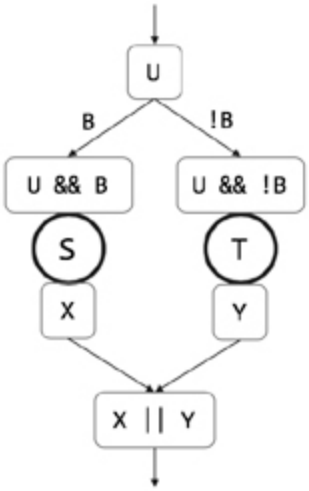
Floyd logic tells us that this decorated program is correct if the following are valid formulas and valid Hoare triples:

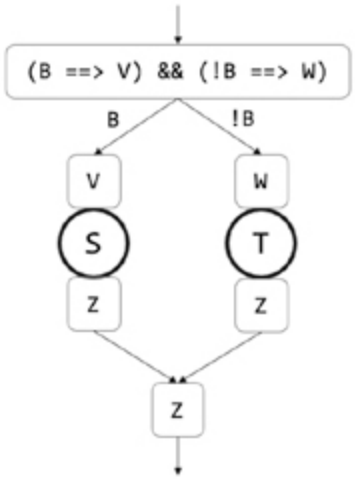
- `U && B ==> V`
- `U && !B ==> W`
- `{{ V }} S {{ X }}`
- `{{ W }} T {{ Y }}`
- `X ==> Z`
- `Y ==> Z`

If we're computing strongest postconditions, then we

- set `V` to `U && B` and set `W` to `U && !B`,
- compute `X` as the strongest postcondition for `V,S` and compute `Y` as the strongest postcondition for `W,T`, and
- set `Z` to `X || Y`.

Pictorially, we have the diagram shown here to the left:





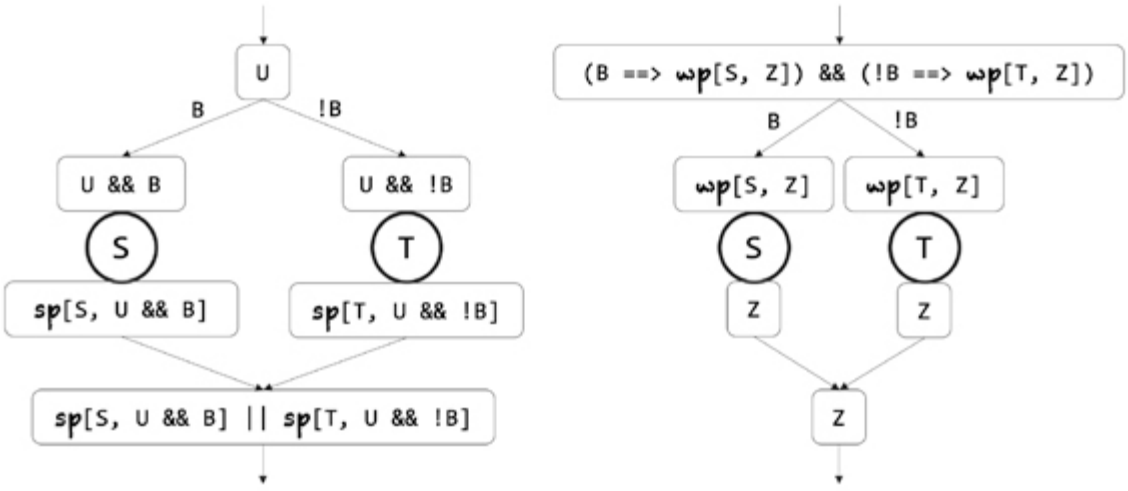
To compute weakest preconditions, we instead

- set both `X` and `Y` to `Z`,
- compute `V` as the weakest precondition of `S,X` and compute `W` as the weakest pre-condition of `T,Y`, and
- set `U` to `(B ==> V) && (!B ==> W)`.

Pictorially, we have the diagram shown above to the right.

Regardless of how you come up with the intermediate predicates in these diagrams, what you should observe is how the predicates are transformed as you follow—either forwards or backwards—the edges of the control-flow graph.

Here are the same flow diagrams, but showing and in the diagrams:



> **Exercise 2.18.**
>
> (a) Draw a decorated flow diagram as the ones above for the conditional statement
>
> `if x < 3 { x, y := x + 1, 10; } else { y := x; }`
>
> (b) In the forward direction and using `x + y == 100` as the precondition (that is, as `U`), fill in the formulas in the remaining boxes. (c) In the backward direction and using `x + y == 100` as the postcondition (that is, as `Z`), fill in the formulas in the remaining boxes.

2.5.0.Just the formulas, ma’am

Written with just formulas—no diagrams—the meaning of `if` statements in terms of and is:

`if B { S } else { T }, P =`

`S, P && B || T, P && !B`

`if B { S } else { T }, Q =`

`(B ==> S, Q) && (!B ==> T, Q)`

> **Exercise 2.19.**
>
> Suppose `x < 100` is known to hold before the statement
>
> `if x < 20 { y := 3; } else { y := 2; }`
>
> then what is the strongest condition you know after the statement? In other words, compute the strongest postcondition of the statement with respect to `x < 100`. Simplify the condition after you have computed it.

> **Exercise 2.20.**
>
> Suppose you want `x + y == 22` to hold after the statement
>
> `if x < 20 { y := 3; } else { y := 2; }`

then in which states can you start the statement? In other words, compute the weakest precondition of the statement with respect to `x + y == 22`. Simplify the condition after you have computed it.

Exercise 2.21.

Compute the weakest precondition for the following statement with respect to `y < 10`. Simplify the condition.

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

Exercise 2.22.

Compute the weakest precondition for the following statement with respect to `y % 2 == 0` (that is, "y is even"). Simplify the condition.

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

Exercise 2.23.

Compute the weakest precondition for the following statement with respect to `y % 2 == 0` (that is, "y is even"). Simplify the condition.

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

Exercise 2.24.

Determine under which circumstances the following program establishes `0 <= y < 100`. If you're starting to get a hang of how to compute weakest preconditions, try to do the computation 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; }
}
```

2.6.Sequential Composition

Another way to compose two statements is to follow one by the other. This simple idea goes by the multi-syllable name *sequential composition*. For statements `S` and `T`, we can convey the Floyd logic semantics of `S;T` by writing two overlapping Hoare triples

```
{{ P }} S {{ Q }} T {{ R }}
```

This is valid if the following two individual Hoare triples are valid:

- `{{ P }} S {{ Q }}`
- `{{ Q }} T {{ R }}`

There's no reason we have to involve the predicate `Q`, because there is no loss in generality by setting `Q` to either `S, P` or `T, R`. In other words, we can directly define the validity of the Hoare triple

```
{{ P }} S;T {{ R }}
```

in terms of strongest postconditions or weakest preconditions. In formulas, we have

```
S;T, P=T, [S, P
S;T, R=S, T, R
```

Notice that the compositions in these two right-hand sides go in opposite directions— is computed forwards from `P` and is computed backward from `R`.

Exercise 2.25.

Which of the following Hoare-triple combinations are valid?

```
a) {{ 0 <= x }} x := x + 1 {{ -2 <= x }} y := 0 {{ -10 <= x }}
b) {{ 0 <= x }} x := x + 1 {{ true }} x := x + 1 {{ 2 <= x }}
c) {{ 0 <= x }} x := x + 1; x := x + 1 {{ 2 <= x }}
d) {{ 0 <= x }} x := 3 * x; x := x + 1 {{ 3 <= x }}
e) {{ x < 2 }} y := x + 5; x := 2 * x {{ x < y }}
```

-

Exercise 2.26.

Practice working backward over several assignment statements. Compute with respect to `x + y <= 100` for the following programs:

```
a) x := x + 1; y := x + y
b) y := x + y; x := x + 1
c) x, y := x + 1, x + y
```

Exercise 2.27.

Compute with respect to `x + y <= 100` for the following programs:

```
a) x := x + 1; y := x + y
b) y := x + y; x := x + 1
c) x, y := x + 1, x + y
```

Exercise 2.28.

Compute the strongest postcondition and weakest precondition of the following statements, each with respect to the predicate `x + y < 100`. Simplify the predicates.

```
a) x := 32; y := 40
b) x := x + 2; y := y - 3 * x
```

Exercise 2.29.

Compute

```
a) [var x; x := 10, x <=
   100]
b) [var x; x := 10, x <=
   100]
```

Exercise 2.30.

Compute the strongest postcondition and weakest precondition of the following statements, each with respect to the predicate `x < 10`:

```
if x % 2 == 0 { y := y + 3; } else { y := 4; }
if y < 10 { y := x + y; } else { x := 8; }
```

2.7.Method Calls and Postconditions

Methods are opaque. This means we reason about them in terms of their specifications, not their implementations (their bodies). Consider our `Triple` method from Section 1.4 again:

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

The basic idea is that we expect to be able to prove Hoare triples like

```
{{ true }} t := Triple(u + 3) {{ t == 3 * (u + 3) }}
```

(This *is* what you'd expect, right?) To make this precise, we need to consider several details. Let's start with the parameters.

2.7.0.Parameters

For illustration, let's use the call

```
t := Triple(u + 3)
```

The specification uses the formal parameters `x` and `y`, whereas the call site uses the actual parameters `u + 3` and `t`. Somehow, we must connect these. Because of various possible name clashes, including any overlap in the names of the formal and actual parameters, we start by giving the formal parameters *fresh* names. That is, we rename the formal parameters to names that are not already used in our verification. Stated in yet one more way, the names are fresh if they are unique to this call site.

For this step, I will rename `x` to `x′` and `y` to `y′`. With these fresh names, the method and its specification look like this:

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

The meaning of the call can now be explained as the sequential composition of first assigning the actual in-parameters to the freshly named formal in-parameters and then assigning the freshly named formal out-parameters to the actual out-parameters, where—importantly—we get to assume the relation between the freshly named variables entailed by the postconditions. For our example, these two assignments are `x′ := u + 3` and `t := y′`, and the relation we get to assume is `y′ == 3 * x′`.

2.7.1.Assumptions

To make things more precise, it would be convenient to have a program statement that introduced an assumption. For this purpose, let me define a statement **`assume`** `E` for a boolean expression `E`. For any pre-state predicate `P` and post-state predicate `Q`, the semantics of **`assume`** is given by the following two equations:

```
assume E, P=P && E
assume E, Q=E ==> Q
```

The first of these equations says that if you start **`assume`** `E` in a state satisfying `P`, then you get to assume that both `P` and `E` hold afterwards. The second equation says that if you want to prove `Q` to hold after **`assume`** `E`, then it suces to prove, before the statement, that `Q` holds under the assumption `E`.

The **`assume`** statement is fictitious. That is, it is not a statement that a compiler could handle. After all, it introduces an assumption, much like making a wish come true. Nevertheless, if you just think of the fictitious statement as “oh, at this point, I apparently get to assume that condition `E` holds”, then this **`assume`** statement is quite handy as a primitive in defining the semantics of non-fictitious statements. Like the call statement.

2.7.2.Semantics of method call with a postcondition

Using renaming and an assumption, we can define the semantics of our illustrative call. Above, we already freshly renamed the formal parameters of `Triple`. Next, the semantics of the call is given by the following program snippet:

```
var x′, y′;
x′ := u + 3;
assume y′ == 3 * x′;
t := y′
```

In other words, I'm saying that you reason about the call

```
t := Triple(u + 3)
```

in the same way as you reason about these four statements.

In terms of weakest preconditions, we thus have, for any predicate Q,

```
t := Triple(u + 3), Q
={ definition of a call in terms of the four statements }
var x′,y′; x′ := u + 3; assume y′ == 3 * x′; t := y′, Q
={ definition of  for ; }
var x′,y′, x′ := u + 3,
assume y′ == 3 * x′, t := y′, Q
={ definition of  for := and assume and var }
forall x′, y′ :: (y′ == 3 * x′ ==> Q[t := y′])[x′ := u + 3]
={ apply the substitution for x′ }
forall x′, y′ :: y′ == 3 * (u + 3) ==> Q[t := y′]
={ x′ is not used }
forall y′ :: y′ == 3 * (u + 3) ==> Q[t := y′]
={ logic }
Q[t := y′][y′ := 3 * (u + 3)]
={ apply substitution for y′ }
Q[t := 3 * (u + 3)]
```

Let's try this out. As an example, take `Q` to be `t == 54`. That is, if we want the call `t := Triple(u + 3)` to establish `t == 54`, then what needs to holds before we make the call?

```
t := Triple(u + 3), t == 54
```

```
={  of call, see above }

(t == 54)[t := 3 * (u + 3)]

={ substitutions }

3 * (u + 3) == 54

={ arithmetic }

u == 15
```

So, if we want `t := Triple(u + 3)` to establish `t == 54`, we must make sure to make the call when `u == 15`. This matches our understanding of the call and its specification, because if `u` is `15`, then `u + 3` is `18`, and then `Triple` returns `54`.

More generally, for a method

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

the semantics of a call to `M` is

```
t := M(E), Q =
forall x', y' :: (R[x,y := x',y'] ==> Q[t := y'])[x' := E]
```

where `x'` and `y'` are fresh names and `Q` is any predicate on the post-state of the call. Notice that I included the renaming in this equation, rather than first doing a renaming step. Since `x'` is fresh, it does not occur in `Q`, so we can simplify the equation by distributing the substitutions:

```
t := M(E), Q =
forall y' :: R[x,y := E,y'] ==> Q[t := y']
```

What I wrote applies to any number of in- and out-parameters. That is, it applies equally well if `x`, `y`, and `t` are lists of variables and `E` is a list of expressions (provided `x` and `E` have the same lengths, `y` and `t` have the same lengths, and there are no duplicates among the variables in the list `t`).

Exercise 2.31.

Compute the weakest precondition for a call `t := Abs(7 * u)` with respect to the postcondition `u < t`, where `Abs` is defined as follows:

```
method Abs(x: int) returns (y: int)

ensures 0 <= y && (x == y || x == -y)
```

Simplify your answer.

Exercise 2.32.

Compute the weakest precondition for a call `t := Max(2 * u, u + 7)` with respect to the postcondition `t % 2 == 0`, where `Max` is defined as follows:

```
method Max(x: int, y: int) returns (m: int)

ensures m == x || m == y
```

> **ensures** x <= m && y <= m
>
> Simplify your answer.

Since we have a way to give the semantics of a call in terms of the four more primitive statements, the strongest-postcondition semantics of a call follows. For any predicate `P` on the pre-state of the illustrative call to `Triple`, we have:

```
t := Triple(u + 3), P

={ definition of a call in terms of the four statements }

var x′,y′; x′ := u + 3; assume y′ == 3 * x′; t := y′, P

={ definition of  for ; }

var x′,y′, t := y′,

[assume y′ == 3 * x′, x′ := u + 3, P

={ definition of  for :=, simplifying since x′ is fresh }

var x′,y′, t := y′,

assume y′ == 3 * x′, P && x′ == u + 3

={ definition of  for assume }

var x′,y′, t := y′,

P && x′ == u + 3 && y′ == 3 * x′

={ definition of  for :=, where t0 is fresh }

var x′,y′,

exists t0 :: (P && x′ == u + 3 && y′ == 3 * x′)[t := t0] && t == y′

={ simplification }

var x′,y′,

exists t0 :: P[t := t0] && x′ == u + 3 && y′ == 3 * x′ && t == y′

={ definition of  for var }

exists x′, y′ :: exists t0 ::

P[t := t0] && x′ == u + 3 && y′ == 3 * x′ && t == y′

={ logic simplification for x′ }

exists y′ :: exists t0 ::

(P[t := t0] && y′ == 3 * x′ && t == y′)[x′ := u + 3]

={ apply substitution for x′ }

exists y′ :: exists t0 ::

P[t := t0] && y′ == 3 * (u + 3) && t == y′
```

```
={ logic simplification for y' }

(exists t0 :: P[t := t0] && y' == 3 * (u + 3))[y' := t]

={ apply substitution for y' }

exists t0 :: P[t := t0] && t == 3 * (u + 3)
```

In the general case where M is the method above with the postcondition `R`, we have

```
t := M(E), P =

exists t0 :: P[t := t0] && R[x := E[t := t0]][y := t]
```

where t_0 is fresh. (I don't know about you, but I find the weakest-precondition formulation easier to understand than this strongest-postcondition formulation.)

2.8.Assert Statements

In what I described above for method calls, I only showed postconditions. Before I cover preconditions, I will define the semantics of **`assert`** statements.

The statement **`assert`** `E` is a no-op if `E` holds; otherwise, it crashes your program. So if you want to prove that a condition `Q` holds after the statement, then you must prove that both `E` and `Q` hold before it—`E` to prevent the assertion from crashing, and `Q` because the assertion doesn't change the state. This is captured in the following equation:

```
assert E, Q=E && Q
```

We're always interested in proving our programs to be crash-free, so the effect of the **`assert`** statement is to introduce a proof obligation.

> **Exercise 2.33.**
>
> Compute the weakest precondition of the following statements with respect to the postcondition `x < 100`. Simplify each answer.
>
> a) **`assert`** `y == 25`
>
> b) **`assert`** `0 <= x`
>
> c) **`assert`** `x < 200`
>
> d) **`assert`** `x <= 100`
>
> e) **`assert`** `0 <= x < 100`

2.8.0.The real difference between and

Here is the equation for **`assert`**:

```
assert E, P=P && E
```

It says that the most you can be sure of after a crash-free execution of the statement is that both `P` and `E` hold.

Until now, the choice of using versus has seemed rather arbitrary, or perhaps simply a matter of taste. Okay, for assignment statements, is more complicated than , but is there any deeper difference between and ? Yes, there is, and the **`assert`** statement brings out this difference.

If you would consider defining the formal meaning of statements using just , then you would find yourself with two questions about the equation for **`assert`** above:

0. Under what circumstances does **`assert`** `E` crash? For example, will the statement always crash if `E` does not hold? Of greater interest, can we be assured that the statement does *not* crash if `E` holds?
1. An assertion is supposed to give us a way to introduce a proof obligation, so how does capture the fact that `E` is a proof obligation?

The answer to question 0 is that does *not* say when a statement can crash. It only says that *if* the statement does *not* crash, then we know what holds afterwards.

The answer to question 1 is that does not. That is, is incognizant of proof obligations. If you look back to Section 2.7.1, you may be alarmed to find that is the same for **`assert`** `E` as it is for **`assume`** `E`. In that section, I had defined **`assume`** `E` as a fictitious statement that introduces an assumption. This assumption comes for free, like a wish or a pipe dream, and there is no proof obligation involved. In contrast, the intended meaning of **`assert`** `E` is to introduce a proof obligation—we're not just assuming the condition `E` to hold, but we also want to prove that it does hold. Yet, **`assert`** `E, P` is the same as **`assume`** `E, P`. Both of these say that *in the event that* the statement terminates without crashing, then `P && E` holds afterwards.

Exercise 2.34.

Suppose we didn't use , but we defined the semantics of a statement in terms of alone. Then, according to the equation, would you say that an **`assert`** statement could lower your Candy Crush score?

In conclusion, tells us the strongest thing we can say upon crash-free execution of a statement, but it does not tell us under which circumstances a statement *is* crash-free. In contrast, tells us the pre-states from which a statement is guaranteed to be crash-free and terminate in a state we'd like. The **`assert`** and **`assume`** statements highlight this difference:

```
assert E, Q=E && Q
assume E, Q=E ==> Q
assert E, P=P && E
assume E, P=P && E
```

These equations say that if we want to establish `Q` after the statement, then for **`assert`** `E`, we must prove that `E` holds *and* prove that `Q` holds, whereas for **`assume`** `E`, we never need to prove that `E` holds and we still get to assume `E` when proving `Q`. The equations just say that if the statements terminate without crashing, then `P && E` holds.

2.8.1.An informal reading

The names of the statements **`assert`** and **`assume`** are not accidental. They have been chosen to let us internalize what the statements mean. You can forget the mathematical definitions for these statements if (or: there's no reason to get caught up in the formal details as long as) you understand the statements like this:

> The statement **`assert`** `E` says “at this program point, we assert that `E` holds; that is, we expect that `E` holds and we're going to prove that it does”. Alternatively, “we expect that whenever control reaches this program point, `E` holds, and we're going to prove that this is indeed the case for all possible executions”.

> The statement **`assume`** `E` says “at this program point, we have no idea if `E` holds or not; nevertheless, for the purpose of proving the correctness of our program, we are going to assume that `E` does hold here”. Alternatively, “when you run the program and control reaches this point, `E` may or may not hold, but as far as our proof is concerned, we are going to ignore any execution that gets here and `E` does not hold”.

2.8.2.Using `assume` to reason about calls

How come I dare use **`assume`** when defining the meaning of calls, as I did in Section 2.7.2? Are we really proving the correctness of our program if the **`assume`** causes the proof to ignore certain executions? In this case, yes, we are still proving the correctness of our program, because we are going to make sure that no executions are ignored. For a call to a method `M`, I used the **`assume`** statement at the call site to assume the post-condition of `M`. Sooner or later in the reasoning about our program, we're going to prove the correctness of the implementation of `M`. In that proof, we are going to ensure that the method implementation establishes the postcondition. In this way, it will always be the case that, when a call returns, the postcondition of the method called holds. So, no executions are ignored after all.

> This justifies the use of **`assume`** in the definition of a call.

2.9.Weakest Liberal Preconditions

My aim in this book is to equip you with an understanding of how to reason about your programs, so that you can rigorously explain (to yourself or to a fellow software engineer, or even to a machine) why a program is correct. There is a beautiful aspect of the underlying semantics that is not strictly needed to prove your programs correct. I'm going to give you a taste of that beautiful aspect in this section. It also explains some of the mystery surrounding the difference between and that I talked about above in Section 2.8.0. If you don't care for such discussions about the underlying semantics, you can skip this section and instead continue with Section 2.10.

2.9.0.Turning capturing proof obligations into a separate concern

In Section 2.8.0, I pointed out that does not capture the proof obligations that ensure crash freedom, whereas does. It has also been clear from the beginning that proceeds in the forward direction of a program, whereas proceeds in the backward direction. Does going forward mean that we cannot speak about (crash-freedom) proof obligations?

No, not entirely. We could define a second semantic function in the forward direction, one that would keep track of all proof obligations. (See the upcoming Exercise 2.35.) In the backward direction, already does this. That is, as proceeds in the backward direction, it carries with it any proof obligations it comes across. For example,

```
x := x + 1; assert x <= 10, true

={ definition of  for ; }

x := x + 1, assert x <= 10, true

={ definition of  for assert }

x := x + 1, x <= 10

={ definition of  for := }

x + 1 <= 10
```

Here, we started by computing the weakest precondition necessary to establish **`true`**, and we end up with something that says `x + 1 <= 10` needs to hold in the pre-state. The predicate `x + 1 <= 10` is thus a proof obligation.

So, then, if somehow plays double duty, is there a way to split into more primitive semantic functions? Yes. The function we've seen is sometimes known as the *weakest conservative precondition*. We can define it in terms of a *weakest liberal precondition* that ignores crash executions and a semantic function that accumulates proof obligations for avoiding crashes. Let's define these two.

For any statement `S` and any predicate `Q` on the post-state of `S`, the weakest liberal precondition of `S` with respect to `Q`, written `S, Q`, denotes those pre-states from which every crash-free execution of `S` terminates in a state satisfying `Q`. That is, if an execution starts in a state that satisfies `S, Q`, then either the execution crashes or the execution terminates in a state satisfying `Q`. Note, if `S` is a statement that always terminates without crashing, then `S, Q` is the same as `S, Q`.

For any statement `S`, we need a semantic function that tells us the proof obligations of `S`, that is, that tells us from which states `S` is guaranteed not to crash. We already have such a function, namely `S, true`. Recalling the definition of , we have that `S, true` describes those pre-states from which execution of `S` terminates in a state satisfying **`true`** (easy-peasy!) and *does not crash*. Since every state satisfies **`true`**, `S, true` is what we're looking for.

Putting these two pieces together, we have the following important equation, for every `S` and `Q`:

```
S, Q=S, Q && S, true
```

2.9.1.The connection between and

I argued in Section 2.8.0 that and are not the backward and forward formulations of the same function. But there is a way that and are. It turns out that, for every `S`, `P`, and `Q`, the following holds:

```
S, P ==> Qif and only ifP ==> S, Q
```

I have written and as taking two arguments, a program statement and a predicate. For a fixed `S`, we can think of `S, _` and `S, _` as functions of a predicate. Whenever such functions satisfy the if-and-only-if relation above, they are said to form a *Galois connection* [19]. Function `S, _` is called the *lower adjoint* and function `S, _` is called the *upper adjoint*.

Only certain functions can form a Galois connection. If a function has a corresponding upper adjoint, then that upper adjoint is unique, but the function may not have an upper adjoint at all. Similarly, if a function has a corresponding lower adjoint, then that lower adjoint is unique, but the function may not have an lower adjoint at all.

The functions of a Galois connection have many nice properties. One of these pertains to how the functions distribute over disjunction and conjunction. Every lower adjoint is *universally disjunctive* and every upper adjoint is *universally conjunctive*. For our functions and , this means:

```
S, P0 || P1 || P2 || … =

S, P0 || S, P1 || S, P2 || …
```

and

```
S, Q0 && Q1 && Q2 && … =

S, Q0 && S, Q1 && S, Q2 && …
```

where I have used the notations `P0, P1, P2, …` and `Q0, Q1, Q2, …` to refer to any collections of predicates. I really mean *any*. The collection can be infinite or finite. Indeed, it can be an empty collection, whose disjunction is **`false`** and whose conjunction is **`true`**. So, the following holds, for any statement `S`:

```
S, false=false
S, true=true
```

2.9.2.Strongest conservative postconditions? No such thing!

Is there a semantic function that we can pair with to form a Galois connection? If so, `S, _` would be an upper adjoint and would satisfy:

```
S, true=true
```

But this does not hold for a statement like **`assert false`**. Thus, we conclude that there is no version of that can be paired up with .

Exercise 2.35.

For any assignment statement, **assert** statement, sequential composition, or conditional statement S, define a semantic function

```
S, P
```

that gives the condition that needs to be checked to ensure that no execution of S from a state satisfying P crashes (that is, all executions of S from P are crash free). In other words, S, P should give a predicate that is equivalent to P ==> S, **true**. But instead of defining the semantic function backwards, as is done for , define S, P in the forward direction.

2.10.Method Calls with Preconditions

Back to method calls. In Section 2.7, I discussed the semantics of a call to a method with parameters and a postcondition. Using the **assert** statement from Section 2.8, we can now extend the method-call treatment to include a precondition, which induces a proof obligation for the caller.

As a general template for what a method specification looks like, consider

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

To define the semantics of a call

```
t := M(E)
```

we first find fresh names x′ and y′ for the parameters. Then, the semantics of the call is the same as the semantics for the following program snippet:

```
var x′, y′;
x′ := E;
assert P[x := x′];
assume R[x,y := x′,y′];
t := y′
```

In words, we assign the actual in-parameters E to the freshly named formal in-parameters x′, then we assert the precondition of the call (that is, we check that the precondition holds at the call site), and then, assuming that the freshly named formal out-parameters satisfy the method's postcondition, we assign those to the actual out-parameters of the call.

From this, we can compute the weakest precondition of a call:

```
t := M(E), Q
={ semantics of call }
var x′,y′; x′ := E; assert P[x := x′];
```

```
assume R[x,y := x',y']; t := y', Q

={ definition of  for ; and := }

var x',y'; x' := E; assert P[x := x'];

assume R[x,y := x',y']; Q[t := y']

={ definition of  for ; and assume }

var x',y'; x' := E; assert P[x := x'],

R[x,y := x',y'] ==> Q[t := y']

={ definition of  for ; and assert }

var x',y'; x' := E,

P[x := x'] && (R[x,y := x',y'] ==> Q[t := y'])

={ definition of  for ; and := }

var x',y', (P[x := x'] && (R[x,y := x',y'] ==> Q[t := y']))[x' := E]

={ definition of  for var }

forall x', y' :: (P[x := x'] && (R[x,y := x',y'] ==> Q[t := y']))[x' := E]

={ apply the substitution for x' }

forall x', y' :: P[x := E] && (R[x,y := E,y'] ==> Q[t := y'])

={ x' is not used }

forall y' :: P[x := E] && (R[x,y := E,y'] ==> Q[t := y'])
```

As we had it in Section 2.7.2, the variable `y'` is universally quantified in the last line, because `y'` stands for any values of the formal out-parameters that satisfy the postcondition. So, our final equation for a call, considering both the method's precondition and its postcondition, is:

```
t := M(E), Q =
P[x := E] && forall y' :: R[x,y := E,y'] ==> Q[t := y']
```

> **Exercise 2.36.**
>
> If `x'` is fresh, then prove that
>
> `x' := E; assert P[x := x']`
>
> is the same as
>
> `assert P[x := E]`
>
> by showing that the weakest preconditions of these two statements are the same.

2.11.Function Calls

We reason about a method call in terms of the specification of the method, because methods are opaque. Functions, on the other hand, are transparent, so we reason about them by simply unfolding the definition of the function.

For example, consider a function for computing the absolute value of a number:

```
function Abs(x: int): int {

if x < 0 then -x else x

}
```

Since `Abs` is a function (not a method), its body is an expression (see Section 1.5). Here, I'm using an **`if-then-else`** *expression*, which is an expression form of the **`if`** *statement*. The **`then`** and **`else`** branches of this expression are themselves expressions (whereas the branches of an **`if`** statement are statements). Using function `Abs`, suppose we want to prove, at some program point and for some expression `E`, that `0 <= Abs(E)` always holds. We can do that by writing

```
assert 0 <= Abs(E)
```

at that program point and proving that this statement does not crash. To prove that the statement is crash free and establishes a postcondition `Q`, we proceed as follows:

```
assert 0 <= Abs(E), Q

={  of assert }

0 <= Abs(E) && Q

={ def. Abs }

0 <= (if E < 0 then -E else E) && Q

={ distribute <= over if-then-else }

(if E < 0 then 0 <= -E else 0 <= E) && Q

={ arithmetic }

true && Q

=

Q
```

> **Exercise 2.37.**
>
> Compute `m := Max(x, y), m <= 100` where function `Max` is as defined in Exercise 2.32. Simplify your answer.

> **Exercise 2.38.**
>
> Compute `y := Plus3(x); x := Plus3(y), x < 10`, given

```
function Plus3(x: int): int {
x + 3
}
```

Simplify your answer.

2.12.Partial Expressions

An expression may not always be defined. For example, the division `c / d` makes sense only if `d` is non-`0`. We say that such an expression is *partial*. When an expression is defined in the context where it is used, we describe it using words like *well-formed* or *well-defined*, and if it is always defined, we say that it is *total*.

When we reason about statements, there is a proof obligation that all expressions in the statements be well-defined. Formally, such proof obligations are expressed by . This looks something like this:

```
x := E, Q=E && Q[x := E]
```

where `E` denotes an expression that (itself is total and) evaluates to **`true`** precisely when `E` is well-defined. For example,

```
[x := c / d, x == 100]
```

is

```
d != 0 && c / d == 100
```

I'm not going to give the definition of for all the expressions used in this book, nor am I going to redo the equations we've seen so far to incorporate for every subexpression. Instead, I will typically just speak of there being some proof obligation. For example, I will say that the statement `x := c / d` gives rise to the proof obligation `d != 0`.

Rather than incorporating into the definition of , we can think of every statement as starting with an implicit **`assert`** that contains the necessary proof obligations of the statement. If you always apply to those implicit assertions, then you will get the desired predicates using as I've defined it in this chapter.

Let me show you how this will go. When I say "when you write `c / d`, there is a proof obligation that `d` be non-`0`", what I'm saying is that the implicit assertion that precedes `x := c / d` is

```
assert d != 0
```

And when I point out that a program snippet like

```
if c / d < u / v {
x := a[i];
}
```

comes with the proof obligations that `d` and `v` be non-`0` and that the index `i` be a proper index into the array `a`, what I'm really saying is that the program has the following implicit assertions:

```
assert d != 0 && v != 0;

if c / d < u / v {

assert 0 <= i < a.Length; // this says that i is in range

x := a[i];

}
```

In effect, this transformation uniformly turns all well-definedness proof obligations into assertions. So, the transformed program will crash when the original program would have divided by zero, indexed an array outside its bounds, or engaged in any other undefined behavior.

Not all partial expressions are built-in operations. User-defined functions can have preconditions. For example, consider this function, `MinusOne`:

```
function MinusOne(x: int): int

requires 0 < x
```

Calls to `MinusOne` come with the proof obligation that the argument be positive. Thus, a statement like `z := MinusOne(y)` has the following implicit assertion:

```
assert 0 < y
```

Exercise 2.39.

What is of the following expressions?

a) `x / (y + z)`
b) `a / b < c / d`
c) `a[2 * i]`
d) `MinusOne(MinusOne(y))`

The programming-language operators `&&` and `||`, and `==>` are called *short-circuit operators*, because their well-definedness is sensitive to the value of the left argument. We have

```
E && F=E && (E ==> F)
E || F=E && (E || F)
E ==> F=E && (E ==> F)
```

As these definitions show, the well-definedness of each of the three expressions depends on the well-definedness of `F` only if `E` (is well-defined and) evaluates to **`true`**, **`false`**, and **`true`**, respectively. In a similar way, the well-definedness of an **`if-then-else`** expression depends on the value of the guard:

```
if B then E else F =
B && if B then E else F
```

Exercise 2.40.

What is of the following expressions?

```
a) p && c / d == 100
b) a / b < 10 || c / d < 100
c) if a == b then c / d else 10
d) MinusOne(y) == 8 ==> a[y] == 20
```

Exercise 2.41.

Use an **`if-then-else`** expression that is equivalent (both in value and well-definedness) to (a) `E && F`, (b) `E || F`, and (c) `E ==> F`.

Exercise 2.42.

Revisit Exercise 2.35, but this time, define to also consider the partial expressions for the expressions occurring in the program text.

2.13.Method Correctness

To prove that a method implementation is correct with respect to its specification, we need to show that the given precondition implies the weakest precondition of the method body with respect to the postcondition.

In symbols, to prove that the following method is correct:

```
method M(x: X) returns (y: Y)
requires P
ensures Q
{
Body
}
```

we need to prove

```
P ==> Body, Q
```

2.14.Summary

In this chapter, I defined the semantics of the most common statements we will encounter. In summary, Floyd logic comes down to writing formulas that characterize the program state and verifying that a program “takes” predicates to other predicates. We often use Hoare triples to notate the claim that a program statement takes a given pre-state predicate to a given post-state predicate. The weakest precondition of a statement gives a systematic way to compute a necessary pre-state predicate from a desired post-state predicate. Weakest preconditions give a way to write canonical Hoare triples, by computing the first component of the Hoare triple as the of the other two components. Using computations, the resulting pre-state predicate includes the proof obligations of the program.

Many of the hairier discussions in this chapter surround the use of strongest post-conditions. I have

included these discussions because strongest postconditions are computed in the direction of program flow, which makes them seem more obvious at first. Also, it would be irresponsible of me to omit strongest postconditions in a textbook chapter on program semantics. However, as we have seen, weakest preconditions are not just slightly easier to compute, but they also collect proof obligations as you go along. You wouldn't be able to prove that your program really establishes what you want if these proof obligations are not met. Still, it is good to explicitly check proof obligations, because it lets us pin responsibility on the use of various program statements, like making sure our programs don't divide by zero or call a method without establishing its declared precondition.

With the understanding of program semantics from this chapter, I encourage you to page through the informal arguments of Chapter 1 again. You will feel a sense of mastery of the material when you're able to back up the informal arguments in Chapter 1 with formal arguments in your head and on paper. You will then also appreciate the work automated program verifiers do to compute or behind the scenes for you.

Notes

The general technique in Floyd's logic targeted a flowchart language, which makes it applicable to programs with any kind of control flow [50]. Hoare's presentation of Floyd logic not only used Hoare's elegant triples but also restricted its attention to structured program statements [63]. Such a restriction more effectively suggests compositional design rules for using the constructs in a programming language. For example, Hoare's formulation emphasized the role of an *invariant* when reasoning about a loop. (We'll see loop invariants throughout Part 2.)

For a comprehensive comparison of early approaches to program proofs, see a survey paper on program logics [58].

In his seminal 1976 book, *A Discipline of Programming*, Edsger W. Dijkstra gave five *healthiness conditions* that statements of every reasonable programming language should satisfy [40]. One of these, dubbed the Law of the Excluded Miracle, says that every statement `S` should satisfy

```
S, false=false
```

That is, under no circumstance should a statement `S` guarantee establishing the post-condition **`false`**.

In the 1980s, Back [7], Morgan [92], Morris [96], and Nelson [99] all argued that it is useful to allow constructs like **`assume`** statements in specifications and in the study of program semantics. Note that **`assume false, false`** is **`true`**, which according to Dijkstra's terminology makes the statement miraculous. Nelson called such statements *partial commands* and used the partiality in the definition of **`if`** statements and loops. Morgan defined a general *specification statement* that can be partial. Morgan called the partial form *coercions* and showed that coercions can be combined to make compilable programs [95].

Back and von Wright built a beautiful lattice-theoretic framework of programs and specifications, where **`assert`** and **`assume`** statements are each other's duals [8, 9]. One way they view these statements is as “moves” in a game between two “agents”, often referred to as a “demon” and and an “angel”.

Defying Dijkstra, Burrows and Nelson incorporated **`assume`** statements into a useful programming language for string processing [101]. Playfully, they called it LIM—Language of the *Included* Miracle .

[1]In Hoare's seminal paper where he introduced these triples, crash-freedom and termination were not considered [63]. What I'm using here is a common variation of the triples that does consider these “total correctness” concerns (cf. [3]).

Chapter 3

Recursion and Termination

Your friend Amelia wants to borrow your phone. You agree, so long as Amelia promises to get it back to you. Then, Benny comes along and asks Amelia to borrow your phone. Amelia gives Benny the phone, but makes Benny promise to give it back to you. While Benny still has the phone, Chantal asks to borrow it, and then Dominykas, then Edsger, then Felistas, etc. If this goes on forever, you'll never get your phone back. Yet, it seems that each of your friends lives up to the promise to give the phone back by making an agreement with someone else to take it back. If we allow endless repetition, even of some behavior that seems justifiable by itself, then we don't achieve the results we're hoping for.

This chapter explains how to prevent endless repetitions.

3.0.The Endless Problem

Here are five little programs that demonstrate the need for considering termination.

First up is a method that claims to return a value equal to twice its argument:

```
method BadDouble(x: int) returns (d: int)
ensures d == 2 * x
{
var y := BadDouble(x - 1);
d := y + 2;
```