SOFTWARE FOUNDATIONS

VOLUME 2: PROGRAMMING LANGUAGE FOUNDATIONS

HOARE2

HOARE LOGIC, PART II

```
Set Warnings "-notation-overridden,-parsing".
Require Import Coq.Bool.Bool.
Require Import Coq.Arith.Arith.
Require Import Coq.Arith.EqNat.
Require Import Coq.omega.Omega.
Require Import Maps.
Require Import Imp.
Require Import Hoare.
```

Decorated Programs

The beauty of Hoare Logic is that it is *compositional*: the structure of proofs exactly follows the structure of programs.

This suggests that we can record the essential ideas of a proof (informally, and leaving out some low-level calculational details) by "decorating" a program with appropriate assertions on each of its commands.

Such a *decorated program* carries within it an argument for its own correctness.

For example, consider the program:

```
X ::= m;;
Z ::= p;
WHILE !(X = 0) DO
  Z ::= Z - 1;;
  X ::= X - 1
END
```

(Note the *parameters* `m` and `p`, which stand for fixed-but-arbitrary numbers. Formally, they are simply Coq variables of type `nat`.) Here is one possible specification for this program:

```
    {{ True }}
  X ::= m;;
  Z ::= p;
  WHILE !(X = 0) DO
    Z ::= Z - 1;;
    X ::= X - 1
  END
    {{ Z = p - m }}
```

Here is a decorated version of the program, embodying a proof of this specification:

```
    {{ True }} ->>
    {{ m = m }}
  X ::= m;;
    {{ X = m }} ->>
    {{ X = m ∧ p = p }}
  Z ::= p;
    {{ X = m ∧ Z = p }} ->>
    {{ Z - X = p - m }}
  WHILE !(X = 0) DO
      {{ Z - X = p - m ∧ X ≠ 0 }} ->>
      {{ (Z - 1) - (X - 1) = p - m }}
    Z ::= Z - 1;;
      {{ Z - (X - 1) = p - m }}
    X ::= X - 1
      {{ Z - X = p - m }}
  END
    {{ Z - X = p - m ∧ ¬ (X ≠ 0) }} ->> {{ Z = p - m }}
```

Concretely, a decorated program consists of the program text interleaved with assertions (either a single assertion or possibly two assertions separated by an implication).

To check that a decorated program represents a valid proof, we check that each individual command is *locally consistent* with its nearby assertions in the following sense:

- `SKIP` is locally consistent if its precondition and postcondition are the same:

```
        {{ P }} SKIP {{ P }}
```

- The sequential composition of c_1 and c_2 is locally consistent (with respect to assertions `P` and `R`) if c_1 is locally consistent (with respect to `P` and `Q`) and c_2 is locally consistent (with respect to `Q` and `R`):

```
        {{ P }} c1;; {{ Q }} c2 {{ R }}
```

- An assignment is locally consistent if its precondition is the appropriate substitution of its postcondition:

```
{{ P [X |-> a] }}
X ::= a
{{ P }}
```

- A conditional is locally consistent (with respect to assertions `P` and `Q`) if the assertions at the top of its "then" and "else" branches are exactly `P ∧ b` and `P ∧ ¬b` and if its "then" branch is locally consistent (with respect to `P ∧ b` and `Q`) and its "else" branch is locally consistent (with respect to `P ∧ ¬b` and `Q`):

```
{{ P }}
IFB b THEN
  {{ P ∧ b }}
  c1
  {{ Q }}
ELSE
  {{ P ∧ ¬b }}
  c2
  {{ Q }}
FI
{{ Q }}
```

- A while loop with precondition `P` is locally consistent if its postcondition is `P ∧ ¬b`, if the pre- and postconditions of its body are exactly `P ∧ b` and `P`, and if its body is locally consistent:

```
{{ P }}
WHILE b DO
  {{ P ∧ b }}
  c1
  {{ P }}
END
{{ P ∧ ¬b }}
```

- A pair of assertions separated by `->>` is locally consistent if the first implies the second:

```
{{ P }} ->>
{{ P' }}
```

  This corresponds to the application of `hoare_consequence`, and it is the *only* place in a decorated program where checking whether decorations are correct is not fully mechanical and syntactic, but rather may involve logical and/or arithmetic reasoning.

These local consistency conditions essentially describe a procedure for *verifying* the correctness of a given proof. This verification involves checking that every single command is locally consistent with the accompanying assertions.

If we are instead interested in *finding* a proof for a given specification, we need to discover the right assertions. This can be done in an almost mechanical way, with the exception of finding loop invariants, which is the subject of the next section. In the remainder of this section we explain in detail how to construct decorations for several simple programs that don't involve non-trivial loop invariants.

Example: Swapping Using Addition and Subtraction

Here is a program that swaps the values of two variables using addition and subtraction (instead of by assigning to a temporary variable).

```
X ::= X + Y;;
Y ::= X - Y;;
X ::= X - Y
```

We can prove using decorations that this program is correct — i.e., it always swaps the values of variables X and Y.

```
(1)     {{ X = m ∧ Y = n }} ->>
(2)     {{ (X + Y) - ((X + Y) - Y) = n ∧ (X + Y) - Y = m }}
       X ::= X + Y;;
(3)     {{ X - (X - Y) = n ∧ X - Y = m }}
       Y ::= X - Y;;
(4)     {{ X - Y = n ∧ Y = m }}
       X ::= X - Y
(5)     {{ X = n ∧ Y = m }}
```

These decorations can be constructed as follows:

- We begin with the undecorated program (the unnumbered lines).
- We add the specification — i.e., the outer precondition (1) and postcondition (5). In the precondition, we use parameters m and n to remember the initial values of variables X and Y so that we can refer to them in the postcondition (5).
- We work backwards, mechanically, starting from (5) and proceeding until we get to (2). At each step, we obtain the precondition of the assignment from its postcondition by substituting the assigned variable with the right-hand-side of the assignment. For instance, we obtain (4) by substituting X with X - Y in (5), and we obtain (3) by substituting Y with X - Y in (4).
- Finally, we verify that (1) logically implies (2) — i.e., that the step from (1) to (2) is a valid use of the law of consequence. For this we substitute X by m and Y by n and calculate as follows:

```
(m + n) - ((m + n) - n) = n ∧ (m + n) - n = m
(m + n) - m = n ∧ m = m
```

```
          n = n ∧ m = m
```

Note that, since we are working with natural numbers rather than fixed-width machine integers, we don't need to worry about the possibility of arithmetic overflow anywhere in this argument. This makes life quite a bit simpler!

Example: Simple Conditionals

Here is a simple decorated program using conditionals:

```
    (1)     {{True}}
          IFB X ≤ Y THEN
    (2)       {{True ∧ X ≤ Y}} ->>
    (3)       {{(Y - X) + X = Y ∨ (Y - X) + Y = X}}
            Z ::= Y - X
    (4)       {{Z + X = Y ∨ Z + Y = X}}
          ELSE
    (5)       {{True ∧ ~(X ≤ Y) }} ->>
    (6)       {{(X - Y) + X = Y ∨ (X - Y) + Y = X}}
            Z ::= X - Y
    (7)       {{Z + X = Y ∨ Z + Y = X}}
          FI
    (8)     {{Z + X = Y ∨ Z + Y = X}}
```

These decorations were constructed as follows:

- We start with the outer precondition (1) and postcondition (8).
- We follow the format dictated by the `hoare_if` rule and copy the postcondition (8) to (4) and (7). We conjoin the precondition (1) with the guard of the conditional to obtain (2). We conjoin (1) with the negated guard of the conditional to obtain (5).
- In order to use the assignment rule and obtain (3), we substitute `Z` by `Y - X` in (4). To obtain (6) we substitute `Z` by `X - Y` in (7).
- Finally, we verify that (2) implies (3) and (5) implies (6). Both of these implications crucially depend on the ordering of `X` and `Y` obtained from the guard. For instance, knowing that `X ≤ Y` ensures that subtracting `X` from `Y` and then adding back `X` produces `Y`, as required by the first disjunct of (3). Similarly, knowing that `¬ (X ≤ Y)` ensures that subtracting `Y` from `X` and then adding back `Y` produces `X`, as needed by the second disjunct of (6). Note that `n - m + m = n` does *not* hold for arbitrary natural numbers `n` and `m` (for example, `3 - 5 + 5 = 5`).

Exercise: 2 stars (if_minus_plus_reloaded)

Fill in valid decorations for the following program:

```
     {{ True }}
    IFB X ≤ Y THEN
        {{                                   }} ->>
```

```
           {{                                    }}
         Z  ::= Y - X
           {{                                    }}
       ELSE
           {{                                    }} ->>
           {{                                    }}
         Y  ::= X + Z
           {{                                    }}
       FI
         {{ Y = X + Z }}
```

☐

Example: Reduce to Zero

Here is a `WHILE` loop that is so simple it needs no invariant (i.e., the invariant `True` will do the job).

```
        (1)      {{ True }}
               WHILE !(X = 0) DO
        (2)        {{ True ∧ X ≠ 0 }} ->>
        (3)        {{ True }}
                 X ::= X - 1
        (4)        {{ True }}
               END
        (5)      {{ True ∧ X = 0 }} ->>
        (6)      {{ X = 0 }}
```

The decorations can be constructed as follows:

- Start with the outer precondition (1) and postcondition (6).
- Following the format dictated by the `hoare_while` rule, we copy (1) to (4). We conjoin (1) with the guard to obtain (2) and with the negation of the guard to obtain (5). Note that, because the outer postcondition (6) does not syntactically match (5), we need a trivial use of the consequence rule from (5) to (6).
- Assertion (3) is the same as (4), because `X` does not appear in `4`, so the substitution in the assignment rule is trivial.
- Finally, the implication between (2) and (3) is also trivial.

From an informal proof in the form of a decorated program, it is easy to read off a formal proof using the Coq versions of the Hoare rules. Note that we do *not* unfold the definition of `hoare_triple` anywhere in this proof — the idea is to use the Hoare rules as a self-contained logic for reasoning about programs.

```
Definition reduce_to_zero' : com :=
  WHILE !(X = 0) DO
    X ::= X - 1
  END.
```

```
Theorem reduce_to_zero_correct' :
  {{fun st ⇒ True}}
  reduce_to_zero'
  {{fun st ⇒ st X = 0}}.
```
+

Example: Division

The following Imp program calculates the integer quotient and remainder of two numbers `m` and `n` that are arbitrary constants in the program.

```
X ::= m;;
Y ::= 0;;
WHILE n ≤ X DO
  X ::= X - n;;
  Y ::= Y + 1
END;
```

In we replace `m` and `n` by concrete numbers and execute the program, it will terminate with the variable `X` set to the remainder when `m` is divided by `n` and `Y` set to the quotient.

In order to give a specification to this program we need to remember that dividing `m` by `n` produces a reminder `X` and a quotient `Y` such that `n * Y + X = m ∧ X < n`.

It turns out that we get lucky with this program and don't have to think very hard about the loop invariant: the invariant is just the first conjunct `n * Y + X = m`, and we can use this to decorate the program.

```
(1)    {{ True }} ->>
(2)    {{ n * 0 + m = m }}
     X ::= m;;
(3)    {{ n * 0 + X = m }}
     Y ::= 0;;
(4)    {{ n * Y + X = m }}
     WHILE n ≤ X DO
(5)      {{ n * Y + X = m ∧ n ≤ X }} ->>
(6)      {{ n * (Y + 1) + (X - n) = m }}
       X ::= X - n;;
(7)      {{ n * (Y + 1) + X = m }}
       Y ::= Y + 1
(8)      {{ n * Y + X = m }}
     END
(9)    {{ n * Y + X = m ∧ X < n }}
```

Assertions (4), (5), (8), and (9) are derived mechanically from the invariant and the loop's guard. Assertions (8), (7), and (6) are derived using the assignment rule going backwards from (8) to (6). Assertions (4), (3), and (2) are again backwards applications of the assignment rule.

Now that we've decorated the program it only remains to check that the two uses of the consequence rule are correct — i.e., that (1) implies (2) and that (5) implies (6). This is indeed the case, so we have a valid decorated program.

Finding Loop Invariants

Once the outermost precondition and postcondition are chosen, the only creative part in verifying programs using Hoare Logic is finding the right loop invariants. The reason this is difficult is the same as the reason that inductive mathematical proofs are: strengthening the loop invariant (or the induction hypothesis) means that you have a stronger assumption to work with when trying to establish the postcondition of the loop body (or complete the induction step of the proof), but it also means that the loop body's postcondition (or the statement being proved inductively) is stronger and thus harder to prove!

This section explains how to approach the challenge of finding loop invariants through a series of examples and exercises.

Example: Slow Subtraction

The following program subtracts the value of X from the value of Y by repeatedly decrementing both X and Y. We want to verify its correctness with respect to the pre- and postconditions shown:

```
  {{ X = m ∧ Y = n }}
WHILE !(X = 0) DO
  Y ::= Y - 1;;
  X ::= X - 1
END
  {{ Y = n - m }}
```

To verify this program, we need to find an invariant I for the loop. As a first step we can leave I as an unknown and build a *skeleton* for the proof by applying the rules for local consistency (working from the end of the program to the beginning, as usual, and without any thinking at all yet).

This leads to the following skeleton:

```
(1)      {{ X = m ∧ Y = n }}  ->>             (a)
(2)      {{ I }}
       WHILE !(X = 0) DO
(3)        {{ I ∧ X ≠ 0 }}  ->>               (c)
(4)        {{ I [X |-> X-1] [Y |-> Y-1] }}
         Y ::= Y - 1;;
(5)        {{ I [X |-> X-1] }}
         X ::= X - 1
```

```
(6)           {{ I }}
          END
(7)         {{ I ∧ ¬ (X ≠ 0) }}  ->>              (b)
(8)         {{ Y = n - m }}
```

By examining this skeleton, we can see that any valid `I` will have to respect three conditions:

- (a) it must be *weak* enough to be implied by the loop's precondition, i.e., (1) must imply (2);
- (b) it must be *strong* enough to imply the program's postcondition, i.e., (7) must imply (8);
- (c) it must be *preserved* by one iteration of the loop, i.e., (3) must imply (4).

These conditions are actually independent of the particular program and specification we are considering. Indeed, every loop invariant has to satisfy them. One way to find an invariant that simultaneously satisfies these three conditions is by using an iterative process: start with a "candidate" invariant (e.g., a guess or a heuristic choice) and check the three conditions above; if any of the checks fails, try to use the information that we get from the failure to produce another — hopefully better — candidate invariant, and repeat.

For instance, in the reduce-to-zero example above, we saw that, for a very simple loop, choosing `True` as an invariant did the job. So let's try instantiating `I` with `True` in the skeleton above and see what we get...

```
(1)         {{ X = m ∧ Y = n }} ->>        (a - OK)
(2)         {{ True }}
          WHILE !(X = 0) DO
(3)           {{ True ∧ X ≠ 0 }}  ->>     (c - OK)
(4)           {{ True }}
            Y ::= Y - 1;;
(5)           {{ True }}
            X ::= X - 1
(6)           {{ True }}
          END
(7)         {{ True ∧ X = 0 }}  ->>        (b - WRONG!)
(8)         {{ Y = n - m }}
```

While conditions (a) and (c) are trivially satisfied, condition (b) is wrong, i.e., it is not the case that `True ∧ X = 0` (7) implies `Y = n - m` (8). In fact, the two assertions are completely unrelated, so it is very easy to find a counterexample to the implication (say, `Y = X = m = 0` and `n = 1`).

If we want (b) to hold, we need to strengthen the invariant so that it implies the postcondition (8). One simple way to do this is to let the invariant *be* the postcondition. So let's return to our skeleton, instantiate `I` with `Y = n - m`, and check conditions (a) to (c) again.

```
    (1)       {{ X = m ∧ Y = n }}  ->>           (a - WRONG!)
    (2)       {{ Y = n - m }}
            WHILE !(X = 0) DO
    (3)         {{ Y = n - m ∧ X ≠ 0 }}  ->>   (c - WRONG!)
    (4)         {{ Y - 1 = n - m }}
              Y ::= Y - 1;;
    (5)         {{ Y = n - m }}
              X ::= X - 1
    (6)         {{ Y = n - m }}
            END
    (7)       {{ Y = n - m ∧ X = 0 }}  ->>       (b - OK)
    (8)       {{ Y = n - m }}
```

This time, condition (b) holds trivially, but (a) and (c) are broken. Condition (a) requires that (1) X = m ∧ Y = n implies (2) Y = n - m. If we substitute Y by n we have to show that n = n - m for arbitrary m and n, which is not the case (for instance, when m = n = 1). Condition (c) requires that n - m - 1 = n - m, which fails, for instance, for n = 1 and m = 0. So, although Y = n - m holds at the end of the loop, it does not hold from the start, and it doesn't hold on each iteration; it is not a correct invariant.

This failure is not very surprising: the variable Y changes during the loop, while m and n are constant, so the assertion we chose didn't have much chance of being an invariant!

To do better, we need to generalize (8) to some statement that is equivalent to (8) when X is 0, since this will be the case when the loop terminates, and that "fills the gap" in some appropriate way when X is nonzero. Looking at how the loop works, we can observe that X and Y are decremented together until X reaches 0. So, if X = 2 and Y = 5 initially, after one iteration of the loop we obtain X = 1 and Y = 4; after two iterations X = 0 and Y = 3; and then the loop stops. Notice that the difference between Y and X stays constant between iterations: initially, Y = n and X = m, and the difference is always n - m. So let's try instantiating I in the skeleton above with Y - X = n - m.

```
    (1)       {{ X = m ∧ Y = n }}  ->>                (a - OK)
    (2)       {{ Y - X = n - m }}
            WHILE !(X = 0) DO
    (3)         {{ Y - X = n - m ∧ X ≠ 0 }}  ->>     (c - OK)
    (4)         {{ (Y - 1) - (X - 1) = n - m }}
              Y ::= Y - 1;;
    (5)         {{ Y - (X - 1) = n - m }}
              X ::= X - 1
    (6)         {{ Y - X = n - m }}
            END
    (7)       {{ Y - X = n - m ∧ X = 0 }}  ->>        (b - OK)
    (8)       {{ Y = n - m }}
```

Success! Conditions (a), (b) and (c) all hold now. (To verify (c), we need to check that, under the assumption that X ≠ 0, we have Y - X = (Y - 1) - (X - 1); this holds for all natural numbers X and Y.)

Exercise: Slow Assignment

Exercise: 2 stars (slow_assignment)

A roundabout way of assigning a number currently stored in X to the variable Y is to start Y at 0, then decrement X until it hits 0, incrementing Y at each step. Here is a program that implements this idea:

```
      {{ X = m }}
    Y ::= 0;;
    WHILE !(X = 0) DO
      X ::= X - 1;;
      Y ::= Y + 1
    END
      {{ Y = m }}
```

Write an informal decorated program showing that this procedure is correct.

```
(* FILL IN HERE *)
```

□

Exercise: Slow Addition

Exercise: 3 stars, optional (add_slowly_decoration)

The following program adds the variable X into the variable Z by repeatedly decrementing X and incrementing Z.

```
    WHILE !(X = 0) DO
       Z ::= Z + 1;;
       X ::= X - 1
    END
```

Following the pattern of the subtract_slowly example above, pick a precondition and postcondition that give an appropriate specification of add_slowly; then (informally) decorate the program accordingly.

```
(* FILL IN HERE *)
```

□

Example: Parity

Here is a cute little program for computing the parity of the value initially stored in X (due to Daniel Cristofani).

```
      {{ X = m }}
    WHILE 2 ≤ X DO
      X ::= X - 2
    END
      {{ X = parity m }}
```

The mathematical `parity` function used in the specification is defined in Coq as follows:

```
Fixpoint parity x :=
  match x with
  | 0 ⇒ 0
  | 1 ⇒ 1
  | S (S x') ⇒ parity x'
  end.
```

The postcondition does not hold at the beginning of the loop, since `m = parity m` does not hold for an arbitrary `m`, so we cannot use that as an invariant. To find an invariant that works, let's think a bit about what this loop does. On each iteration it decrements `X` by `2`, which preserves the parity of `X`. So the parity of `X` does not change, i.e., it is invariant. The initial value of `X` is `m`, so the parity of `X` is always equal to the parity of `m`. Using `parity X = parity m` as an invariant we obtain the following decorated program:

```
        {{ X = m }} ->>                               (a - OK)
        {{ parity X = parity m }}
      WHILE 2 ≤ X DO
          {{ parity X = parity m ∧ 2 ≤ X }}  ->>     (c - OK)
          {{ parity (X-2) = parity m }}
        X ::= X - 2
          {{ parity X = parity m }}
      END
        {{ parity X = parity m ∧ X < 2 }}  ->>        (b - OK)
        {{ X = parity m }}
```

With this invariant, conditions (a), (b), and (c) are all satisfied. For verifying (b), we observe that, when `X < 2`, we have `parity X = X` (we can easily see this in the definition of `parity`). For verifying (c), we observe that, when `2 ≤ X`, we have `parity X = parity (X-2)`.

Exercise: 3 stars, optional (parity_formal)

Translate this proof to Coq. Refer to the `reduce_to_zero` example for ideas. You may find the following two lemmas useful:

```
Lemma parity_ge_2 : ∀ x,
  2 ≤ x →
  parity (x - 2) = parity x.
+

Lemma parity_lt_2 : ∀ x,
  ¬ 2 ≤ x →
  parity (x) = x.
+

Theorem parity_correct : ∀ m,
    {{ fun st ⇒ st X = m }}
  WHILE 2 ≤ X DO
    X ::= X - 2
  END
    {{ fun st ⇒ st X = parity m }}.
Proof.
  (* FILL IN HERE *) Admitted.
```

☐

Example: Finding Square Roots

The following program computes the (integer) square root of X by naive iteration:

```
      {{ X=m }}
    Z ::= 0;;
    WHILE (Z+1)*(Z+1) ≤ X DO
      Z ::= Z+1
    END
      {{ Z*Z≤m ∧ m<(Z+1)*(Z+1) }}
```

As above, we can try to use the postcondition as a candidate invariant, obtaining the following decorated program:

```
    (1)   {{ X=m }}  -
>>            (a - second conjunct of (2) WRONG!)
    (2)   {{ 0*0 ≤ m ∧ m<1*1 }}
            Z ::= 0;;
    (3)   {{ Z*Z ≤ m ∧ m<(Z+1)*(Z+1) }}
            WHILE (Z+1)*(Z+1) ≤ X DO
    (4)     {{ Z*Z≤m ∧ (Z+1)*(Z+1)≤X }}  -
>>              (c - WRONG!)
    (5)     {{ (Z+1)*(Z+1)≤m ∧ m<(Z+2)*(Z+2) }}
              Z ::= Z+1
    (6)     {{ Z*Z≤m ∧ m<(Z+1)*(Z+1) }}
            END
    (7)   {{ Z*Z≤m ∧ m<(Z+1)*(Z+1) ∧ ~((Z+1)*(Z+1)≤X) }}  -
>> (b - OK)
    (8)   {{ Z*Z≤m ∧ m<(Z+1)*(Z+1) }}
```

This didn't work very well: conditions (a) and (c) both failed. Looking at condition (c), we see that the second conjunct of (4) is almost the same as the first conjunct of (5), except that (4) mentions X while (5) mentions m. But note that X is never assigned in this program, so we should always have X=m, but we didn't propagate this information from (1) into the loop invariant.

Also, looking at the second conjunct of (8), it seems quite hopeless as an invariant (why?); fortunately, we don't need it, since we can obtain it from the negation of the guard — the third conjunct in (7) — again under the assumption that X=m.

So we now try X=m ∧ Z*Z ≤ m as the loop invariant:

```
        {{ X=m }}  -
>>                                             (a - OK)
        {{ X=m ∧ 0*0 ≤ m }}
      Z ::= 0;
        {{ X=m ∧ Z*Z ≤ m }}
      WHILE (Z+1)*(Z+1) ≤ X DO
          {{ X=m ∧ Z*Z≤m ∧ (Z+1)*(Z+1)≤X }}  ->>       (c - OK)
          {{ X=m ∧ (Z+1)*(Z+1)≤m }}
        Z ::= Z+1
          {{ X=m ∧ Z*Z≤m }}
      END
        {{ X=m ∧ Z*Z≤m ∧ X<(Z+1)*(Z+1) }}  ->>          (b - OK)
        {{ Z*Z≤m ∧ m<(Z+1)*(Z+1) }}
```

This works, since conditions (a), (b), and (c) are now all trivially satisfied.

Very often, even if a variable is used in a loop in a read-only fashion (i.e., it is referred to by the program or by the specification and it is not changed by the loop), it is necessary to add the fact that it doesn't change to the loop invariant.

Example: Squaring

Here is a program that squares `X` by repeated addition:

```
      {{ X = m }}
    Y ::= 0;;
    Z ::= 0;;
    WHILE !(Y = X)  DO
      Z ::= Z + X;;
      Y ::= Y + 1
    END
      {{ Z = m*m }}
```

The first thing to note is that the loop reads `X` but doesn't change its value. As we saw in the previous example, it is a good idea in such cases to add `X = m` to the invariant. The other thing that we know is often useful in the invariant is the postcondition, so let's add that too, leading to the invariant candidate `Z = m * m ∧ X = m`.

```
        {{ X = m }} ->>                              (a - WRONG)
        {{ 0 = m*m ∧ X = m }}
      Y ::= 0;;
        {{ 0 = m*m ∧ X = m }}
      Z ::= 0;;
        {{ Z = m*m ∧ X = m }}
      WHILE !(Y = X) DO
          {{ Z = Y*m ∧ X = m ∧ Y ≠ X }} ->>      (c - WRONG)
          {{ Z+X = m*m ∧ X = m }}
        Z ::= Z + X;;
          {{ Z = m*m ∧ X = m }}
        Y ::= Y + 1
          {{ Z = m*m ∧ X = m }}
      END
        {{ Z = m*m ∧ X = m ∧ ~(Y ≠ X) }} ->>            (b - OK)
        {{ Z = m*m }}
```

Conditions (a) and (c) fail because of the `Z = m*m` part. While `Z` starts at `0` and works itself up to `m*m`, we can't expect `Z` to be `m*m` from the start. If we look at how `Z` progresses in the loop, after the 1st iteration `Z = m`, after the 2nd iteration `Z = 2*m`, and at the end `Z = m*m`. Since the variable `Y` tracks how many times we go through the loop, this leads us to derive a new invariant candidate: `Z = Y*m ∧ X = m`.

```
      {{ X = m }} ->>                                  (a - OK)
      {{ 0 = 0*m ∧ X = m }}
    Y ::= 0;;
      {{ 0 = Y*m ∧ X = m }}
    Z ::= 0;;
      {{ Z = Y*m ∧ X = m }}
    WHILE !(Y = X) DO
        {{ Z = Y*m ∧ X = m ∧ Y ≠ X }} ->>          (c - OK)
        {{ Z+X = (Y+1)*m ∧ X = m }}
      Z ::= Z + X;
        {{ Z = (Y+1)*m ∧ X = m }}
      Y ::= Y + 1
        {{ Z = Y*m ∧ X = m }}
    END
      {{ Z = Y*m ∧ X = m ∧ ~(Y ≠ X) }} ->>             (b - OK)
      {{ Z = m*m }}
```

This new invariant makes the proof go through: all three conditions are easy to check.

It is worth comparing the postcondition `Z = m*m` and the `Z = Y*m` conjunct of the invariant. It is often the case that one has to replace parameters with variables — or with expressions involving both variables and parameters, like `m - Y` — when going from postconditions to invariants.

Exercise: Factorial

Exercise: 3 stars (factorial)

Recall that `n!` denotes the factorial of `n` (i.e., `n! = 1*2*...*n`). Here is an Imp program that calculates the factorial of the number initially stored in the variable `X` and puts it in the variable `Y`:

```
    {{ X = m }}
  Y ::= 1 ;;
  WHILE !(X = 0)
  DO
     Y ::= Y * X ;;
     X ::= X - 1
  END
    {{ Y = m! }}
```

Fill in the blanks in following decorated program:

```
    {{ X = m }} ->>
    {{                                      }}
  Y ::= 1;;
    {{                                      }}
  WHILE !(X = 0)
  DO   {{                                      }} ->>
       {{                                      }}
     Y ::= Y * X;;
       {{                                      }}
     X ::= X - 1
       {{                                      }}
  END
    {{                                      }} ->>
    {{ Y = m! }}
```

□

Exercise: Min

Exercise: 3 stars (Min_Hoare)

Fill in valid decorations for the following program. For the ⇒ steps in your annotations, you may rely (silently) on the following facts about min

```
  Lemma lemma1 : ∀ x y,
    (x=0 ∨ y=0) → min x y = 0.
  Lemma lemma2 : ∀ x y,
    min (x-1) (y-1) = (min x y) - 1.
```

plus standard high-school algebra, as always.

```
  {{ True }} ->>
  {{                    }}
  X ::= a;;
  {{                       }}
  Y ::= b;;
  {{                       }}
```

```
Z ::= 0;;
{{                        }}
WHILE !(X = 0) && !(Y = 0) DO
{{                                           }} ->>
{{                                     }}
X := X - 1;;
{{                              }}
Y := Y - 1;;
{{                         }}
Z := Z + 1
{{                        }}
END
{{                              }} ->>
{{ Z = min a b }}
```

□

Exercise: 3 stars (two_loops)

Here is a very inefficient way of adding 3 numbers:

```
X ::= 0;;
Y ::= 0;;
Z ::= c;;
WHILE !(X = a) DO
  X ::= X + 1;;
  Z ::= Z + 1
END;;
WHILE !(Y = b) DO
  Y ::= Y + 1;;
  Z ::= Z + 1
END
```

Show that it does what it should by filling in the blanks in the following decorated program.

```
  {{ True }} ->>
  {{                                        }}
X ::= 0;;
  {{                                        }}
Y ::= 0;;
  {{                                        }}
Z ::= c;;
  {{                                        }}
WHILE !(X = a) DO
    {{                                        }} ->>
    {{                                        }}
  X ::= X + 1;;
    {{                                         }}
  Z ::= Z + 1
    {{                                         }}
END;;
  {{                                         }} ->>
  {{                                         }}
WHILE !(Y = b) DO
    {{                                         }} ->>
    {{                                         }}
  Y ::= Y + 1;;
    {{                                         }}
  Z ::= Z + 1
    {{                                         }}
END
  {{                                         }} ->>
  {{ Z = a + b + c }}
```

□

Exercise: Power Series

Exercise: 4 stars, optional (dpow2_down)

Here is a program that computes the series: `1 + 2 + 2^2 + ... + 2^m = 2^(m+1) - 1`

```
    X ::= 0;;
    Y ::= 1;;
    Z ::= 1;;
    WHILE !(X = m) DO
      Z ::= 2 * Z;;
      Y ::= Y + Z;;
      X ::= X + 1
    END
```

Write a decorated program for this.

```
(* FILL IN HERE *)
```

□

Weakest Preconditions (Optional)

Some Hoare triples are more interesting than others. For example,

```
      {{ False }}  X ::= Y + 1  {{ X ≤ 5 }}
```

is *not* very interesting: although it is perfectly valid, it tells us nothing useful. Since the precondition isn't satisfied by any state, it doesn't describe any situations where we can use the command `X ::= Y + 1` to achieve the postcondition `X ≤ 5`.

By contrast,

```
      {{ Y ≤ 4 ∧ Z = 0 }}  X ::= Y + 1 {{ X ≤ 5 }}
```

is useful: it tells us that, if we can somehow create a situation in which we know that `Y ≤ 4 ∧ Z = 0`, then running this command will produce a state satisfying the postcondition. However, this triple is still not as useful as it could be, because the `Z = 0` clause in the precondition actually has nothing to do with the postcondition `X ≤ 5`. The *most* useful triple (for this command and postcondition) is this one:

```
      {{ Y ≤ 4 }}  X ::= Y + 1  {{ X ≤ 5 }}
```

In other words, `Y ≤ 4` is the *weakest* valid precondition of the command `X ::= Y + 1` for the postcondition `X ≤ 5`.

In general, we say that "`P` is the weakest precondition of command `c` for postcondition `Q`" if `{{P}} c {{Q}}` and if, whenever `P'` is an assertion such that `{{P'}} c {{Q}}`, it is the case that `P' st` implies `P st` for all states `st`.

```
Definition is_wp P c Q :=
  {{P}} c {{Q}} ∧
  ∀ P', {{P'}} c {{Q}} → (P' ->> P).
```

That is, `P` is the weakest precondition of `c` for `Q` if (a) `P` *is* a precondition for `Q` and `c`, and (b) `P` is the *weakest* (easiest to satisfy) assertion that guarantees that `Q` will hold after executing `c`.

Exercise: 1 star, optional (wp)

What are the weakest preconditions of the following commands for the following postconditions?

```
  1) {{ ? }}  SKIP  {{ X = 5 }}

  2) {{ ? }}  X ::= Y + Z {{ X = 5 }}

  3) {{ ? }}  X ::= Y  {{ X = Y }}

  4) {{ ? }}
     IFB X == 0 THEN Y ::= Z + 1 ELSE Y ::= W + 2 FI
     {{ Y = 5 }}

  5) {{ ? }}
     X ::= 5
     {{ X = 0 }}

  6) {{ ? }}
```

```
      WHILE true DO X ::= 0 END
      {{ X = 0 }}

  (* FILL IN HERE *)
```
☐

Exercise: 3 stars, advanced, optional (is_wp_formal)

Prove formally, using the definition of hoare_triple, that Y ≤ 4 is indeed the weakest precondition of X ::= Y + 1 with respect to postcondition X ≤ 5.

```
  Theorem is_wp_example :
    is_wp (fun st ⇒ st Y ≤ 4)
      (X ::= Y + 1) (fun st ⇒ st X ≤ 5).
  Proof.
    (* FILL IN HERE *) Admitted.
```
☐

Exercise: 2 stars, advanced, optional (hoare_asgn_weakest)

Show that the precondition in the rule hoare_asgn is in fact the weakest precondition.

```
  Theorem hoare_asgn_weakest : ∀ Q X a,
    is_wp (Q [X |-> a]) (X ::= a) Q.
  Proof.
  (* FILL IN HERE *) Admitted.
```
☐

Exercise: 2 stars, advanced, optional (hoare_havoc_weakest)

Show that your havoc_pre rule from the himp_hoare exercise in the Hoare chapter returns the weakest precondition.

```
  Module Himp2.
  Import Himp.

  Lemma hoare_havoc_weakest : ∀ (P Q : Assertion) (X : string),
    {{ P }} HAVOC X {{ Q }} →
    P ->> havoc_pre X Q.
  Proof.
  (* FILL IN HERE *) Admitted.
```
☐

Formal Decorated Programs (Optional)

Our informal conventions for decorated programs amount to a way of displaying Hoare triples, in which commands are annotated with enough embedded assertions that checking the validity of a triple is reduced to simple logical and algebraic calculations showing that some assertions imply others. In this section, we show that this informal presentation style can actually be made completely formal and indeed that checking the validity of decorated programs can mostly be automated.

Syntax

The first thing we need to do is to formalize a variant of the syntax of commands with embedded assertions. We call the new commands *decorated commands*, or `dcoms`.

We don't want both preconditions and postconditions on each command, because a sequence of two commands would contain redundant decorations, the postcondition of the first likely being the same as the precondition of the second. Instead, decorations are added corresponding to each postcondition. A separate type, `decorated`, is used to add the precondition for the entire program.

```
Inductive dcom : Type :=
  | DCSkip : Assertion → dcom
  | DCSeq : dcom → dcom → dcom
  | DCAsgn : string → aexp → Assertion → dcom
  | DCIf : bexp → Assertion → dcom → Assertion → dcom
          → Assertion→ dcom
  | DCWhile : bexp → Assertion → dcom → Assertion → dcom
  | DCPre : Assertion → dcom → dcom
  | DCPost : dcom → Assertion → dcom.

Inductive decorated : Type :=
  | Decorated : Assertion → dcom → decorated.

Notation "'SKIP' {{ P }}"
      := (DCSkip P)
      (at level 10) : dcom_scope.
Notation "l '::=' a {{ P }}"
      := (DCAsgn l a P)
      (at level 60, a at next level) : dcom_scope.
Notation "'WHILE' b 'DO' {{ Pbody }} d 'END' {{ Ppost }}"
      := (DCWhile b Pbody d Ppost)
      (at level 80, right associativity) : dcom_scope.
Notation "'IFB' b 'THEN' {{ P }} d 'ELSE' {{ P' }} d' 'FI' {{ Q }}"
      := (DCIf b P d P' d' Q)
      (at level 80, right associativity) : dcom_scope.
Notation "'->>' {{ P }} d"
      := (DCPre P d)
      (at level 90, right associativity) : dcom_scope.
Notation "d '->>' {{ P }}"
      := (DCPost d P)
      (at level 80, right associativity) : dcom_scope.
Notation " d ;; d' "
      := (DCSeq d d')
      (at level 80, right associativity) : dcom_scope.
Notation "{{ P }} d"
      := (Decorated P d)
      (at level 90) : dcom_scope.

Delimit Scope dcom_scope with dcom.
Open Scope dcom_scope.
```

To avoid clashing with the existing `Notation` definitions for ordinary `commands`, we introduce these notations in a special scope called `dcom_scope`, and we `Open` this scope for the remainder of the file.

Careful readers will note that we've defined two notations for specifying a precondition explicitly, one with and one without a ->>. The "without" version is intended to be used to supply the initial precondition at the very top of the program.

```
Example dec_while : decorated :=
  {{ fun st ⇒ True }}
  WHILE !(X = 0)
  DO
    {{ fun st ⇒ True ∧ st X ≠ 0}}
    X ::= X - 1
    {{ fun _ ⇒ True }}
  END
  {{ fun st ⇒ True ∧ st X = 0}} ->>
  {{ fun st ⇒ st X = 0 }}.
```

It is easy to go from a `dcom` to a `com` by erasing all annotations.

```
Fixpoint extract (d:dcom) : com :=
  match d with
  | DCSkip _ ⇒ SKIP
  | DCSeq d1 d2 ⇒ (extract d1 ;; extract d2)
  | DCAsgn X a _ ⇒ X ::= a
  | DCIf b _ d1 _ d2 _ ⇒ IFB b THEN extract d1 ELSE extract d2 FI
  | DCWhile b _ d _ ⇒ WHILE b DO extract d END
  | DCPre _ d ⇒ extract d
  | DCPost d _ ⇒ extract d
  end.

Definition extract_dec (dec : decorated) : com :=
  match dec with
  | Decorated P d ⇒ extract d
  end.
```

The choice of exactly where to put assertions in the definition of `dcom` is a bit subtle. The simplest thing to do would be to annotate every `dcom` with a precondition and postcondition. But this would result in very verbose programs with a lot of repeated annotations: for example, a program like `SKIP;SKIP` would have to be annotated as

```
{{P}} ({{P}} SKIP {{P}}) ;; ({{P}} SKIP {{P}}) {{P}},
```

with pre- and post-conditions on each `SKIP`, plus identical pre- and post-conditions on the semicolon!

Instead, the rule we've followed is this:

- The *post*-condition expected by each `dcom d` is embedded in `d`.
- The *pre*-condition is supplied by the context.

In other words, the invariant of the representation is that a `dcom d` together with a precondition `P` determines a Hoare triple `{{P}} (extract d) {{post d}}`, where `post` is defined as follows:

```
Fixpoint post (d:dcom) : Assertion :=
  match d with
  | DCSkip P ⇒ P
```

```
    | DCSeq d₁ d₂ ⇒ post d₂
    | DCAsgn X a Q ⇒ Q
    | DCIf _ _ d₁ _ d₂ Q ⇒ Q
    | DCWhile b Pbody c Ppost ⇒ Ppost
    | DCPre _ d ⇒ post d
    | DCPost c Q ⇒ Q
    end.
```

It is straightforward to extract the precondition and postcondition from a decorated program.

```
Definition pre_dec (dec : decorated) : Assertion :=
  match dec with
  | Decorated P d ⇒ P
  end.

Definition post_dec (dec : decorated) : Assertion :=
  match dec with
  | Decorated P d ⇒ post d
  end.
```

We can express what it means for a decorated program to be correct as follows:

```
Definition dec_correct (dec : decorated) :=
  {{pre_dec dec}} (extract_dec dec) {{post_dec dec}}.
```

To check whether this Hoare triple is *valid*, we need a way to extract the "proof obligations" from a decorated program. These obligations are often called *verification conditions*, because they are the facts that must be verified to see that the decorations are logically consistent and thus add up to a complete proof of correctness.

Extracting Verification Conditions

The function `verification_conditions` takes a `dcom` `d` together with a precondition `P` and returns a *proposition* that, if it can be proved, implies that the triple `{{P}} (extract d) {{post d}}` is valid.

It does this by walking over `d` and generating a big conjunction including all the "local checks" that we listed when we described the informal rules for decorated programs. (Strictly speaking, we need to massage the informal rules a little bit to add some uses of the rule of consequence, but the correspondence should be clear.)

```
Fixpoint verification_conditions (P : Assertion) (d:dcom)
                              : Prop :=
  match d with
  | DCSkip Q ⇒
      (P ->> Q)
  | DCSeq d₁ d₂ ⇒
      verification_conditions P d₁
      ∧ verification_conditions (post d₁) d₂
  | DCAsgn X a Q ⇒
      (P ->> Q [X |-> a])
  | DCIf b P₁ d₁ P₂ d₂ Q ⇒
      ((fun st ⇒ P st ∧ bassn b st) ->> P₁)
```

```
        ∧ ((fun st ⇒ P st ∧ ¬ (bassn b st)) ->> P₂)
        ∧ (post d₁ ->> Q) ∧ (post d₂ ->> Q)
        ∧ verification_conditions P₁ d₁
        ∧ verification_conditions P₂ d₂
    | DCWhile b Pbody d Ppost ⇒
        (* post d is the loop invariant and the initial
           precondition *)
        (P ->> post d)
        ∧ ((fun st ⇒ post d st ∧ bassn b st) ->> Pbody)
        ∧ ((fun st ⇒ post d st ∧ ~(bassn b st)) ->> Ppost)
        ∧ verification_conditions Pbody d
    | DCPre P' d ⇒
        (P ->> P') ∧ verification_conditions P' d
    | DCPost d Q ⇒
        verification_conditions P d ∧ (post d ->> Q)
    end.
```

And now the key theorem, stating that `verification_conditions` does its job correctly. Not surprisingly, we need to use each of the Hoare Logic rules at some point in the proof.

```
Theorem verification_correct : ∀ d P,
  verification_conditions P d → {{P}} (extract d) {{post d}}.
+
```

Automation

Now that all the pieces are in place, we can verify an entire program.

```
Definition verification_conditions_dec (dec : decorated) : Prop
:=
  match dec with
  | Decorated P d ⇒ verification_conditions P d
  end.

Lemma verification_correct_dec : ∀ dec,
  verification_conditions_dec dec → dec_correct dec.
Proof.
  intros [P d]. apply verification_correct.
Qed.
```

The propositions generated by `verification_conditions` are fairly big, and they contain many conjuncts that are essentially trivial.

```
Eval simpl in (verification_conditions_dec dec_while).

  ==>
  (((fun _ : state ⇒ True) ->> (fun _ : state ⇒ True)) ∧
   ((fun st : state ⇒ True ∧ bassn (! (X = 0)) st) ->>
    (fun st : state ⇒ True ∧ st X ≠ 0)) ∧
   ((fun st : state ⇒ True ∧ ¬ bassn (! (X = 0)) st) ->>
    (fun st : state ⇒ True ∧ st X = 0)) ∧
    (fun st : state ⇒ True ∧ st X ≠ 0) ->>
```

```
        (fun _ : state ⇒ True) [X |-> X - 1]) ∧
        (fun st : state ⇒ True ∧ st X = 0) -
»  (fun st : state ⇒ st X = 0)
```

In principle, we could work with such propositions using just the tactics we have so far, but we can make things much smoother with a bit of automation. We first define a custom `verify` tactic that uses `split` repeatedly to turn all the conjunctions into separate subgoals and then uses `omega` and `eauto` (described in chapter Auto in *Logical Foundations*) to deal with as many of them as possible.

```
Tactic Notation "verify" :=
  apply verification_correct;
  repeat split;
  simpl; unfold assert_implies;
  unfold bassn in *; unfold beval in *; unfold aeval in *;
  unfold assn_sub; intros;
  repeat rewrite t_update_eq;
  repeat (rewrite t_update_neq; [| (intro X; inversion X)]);
  simpl in *;
  repeat match goal with [H : _ ∧ _ |- _] ⇒ destruct H end;
  repeat rewrite not_true_iff_false in *;
  repeat rewrite not_false_iff_true in *;
  repeat rewrite negb_true_iff in *;
  repeat rewrite negb_false_iff in *;
  repeat rewrite beq_nat_true_iff in *;
  repeat rewrite beq_nat_false_iff in *;
  repeat rewrite leb_iff in *;
  repeat rewrite leb_iff_conv in *;
  try subst;
  repeat
    match goal with
      [st : state |- _] ⇒
        match goal with
          [H : st _ = _ |- _] ⇒ rewrite → H in *; clear H
        | [H : _ = st _ |- _] ⇒ rewrite <- H in *; clear H
        end
    end;
  try eauto; try omega.
```

What's left after `verify` does its thing is "just the interesting parts" of checking that the decorations are correct. For very simple examples `verify` immediately solves the goal (provided that the annotations are correct!).

```
Theorem dec_while_correct :
  dec_correct dec_while.
Proof. verify. Qed.
```

Another example (formalizing a decorated program we've seen before):

```
Example subtract_slowly_dec (m:nat) (p:nat) : decorated :=
    {{ fun st ⇒ st X = m ∧ st Z = p }} ->>
    {{ fun st ⇒ st Z - st X = p - m }}
  WHILE ! (X = 0)
  DO {{ fun st ⇒ st Z - st X = p - m ∧ st X ≠ 0 }} ->>
       {{ fun st ⇒ (st Z - 1) - (st X - 1) = p - m }}
```

```
          Z ::= Z - 1
            {{ fun st ⇒ st Z - (st X - 1) = p - m }} ;;
          X ::= X - 1
            {{ fun st ⇒ st Z - st X = p - m }}
      END
        {{ fun st ⇒ st Z - st X = p - m ∧ st X = 0 }} ->>
        {{ fun st ⇒ st Z = p - m }}.

Theorem subtract_slowly_dec_correct : ∀ m p,
  dec_correct (subtract_slowly_dec m p).
Proof. intros m p. verify. (* this grinds for a bit! *) Qed.
```

Examples

In this section, we use the automation developed above to verify formal decorated programs corresponding to most of the informal ones we have seen.

Swapping Using Addition and Subtraction

```
Definition swap : com :=
  X ::= X + Y;;
  Y ::= X - Y;;
  X ::= X - Y.

Definition swap_dec m n : decorated :=
   {{ fun st ⇒ st X = m ∧ st Y = n}} ->>
   {{ fun st ⇒ (st X + st Y) - ((st X + st Y) - st Y) = n
                ∧ (st X + st Y) - st Y = m }}
  X ::= X + Y
   {{ fun st ⇒ st X - (st X - st Y) = n ∧ st X - st Y = m }};;
  Y ::= X - Y
   {{ fun st ⇒ st X - st Y = n ∧ st Y = m }};;
  X ::= X - Y
   {{ fun st ⇒ st X = n ∧ st Y = m}}.

Theorem swap_correct : ∀ m n,
  dec_correct (swap_dec m n).
Proof. intros; verify. Qed.
```

Simple Conditionals

```
Definition if_minus_plus_com :=
  IFB X ≤ Y
    THEN Z ::= Y - X
    ELSE Y ::= X + Z
  FI.

Definition if_minus_plus_dec :=
  {{fun st ⇒ True}}
  IFB X ≤ Y THEN
      {{ fun st ⇒ True ∧ st X ≤ st Y }} ->>
      {{ fun st ⇒ st Y = st X + (st Y - st X) }}
    Z ::= Y - X
      {{ fun st ⇒ st Y = st X + st Z }}
  ELSE
```

```
        {{ fun st ⇒ True ∧ ~(st X ≤ st Y) }} ->>
        {{ fun st ⇒ st X + st Z = st X + st Z }}
      Y ::= X + Z
        {{ fun st ⇒ st Y = st X + st Z }}
    FI
    {{fun st ⇒ st Y = st X + st Z}}.

  Theorem if_minus_plus_correct :
    dec_correct if_minus_plus_dec.
  Proof. verify. Qed.

  Definition if_minus_dec :=
    {{fun st ⇒ True}}
    IFB X ≤ Y THEN
        {{fun st ⇒ True ∧ st X ≤ st Y }} ->>
        {{fun st ⇒ (st Y - st X) + st X = st Y
                 ∨ (st Y - st X) + st Y = st X}}
      Z ::= Y - X
        {{fun st ⇒ st Z + st X = st Y ∨ st Z + st Y = st X}}
    ELSE
        {{fun st ⇒ True ∧ ~(st X ≤ st Y) }} ->>
        {{fun st ⇒ (st X - st Y) + st X = st Y
                 ∨ (st X - st Y) + st Y = st X}}
      Z ::= X - Y
        {{fun st ⇒ st Z + st X = st Y ∨ st Z + st Y = st X}}
    FI
      {{fun st ⇒ st Z + st X = st Y ∨ st Z + st Y = st X}}.

  Theorem if_minus_correct :
    dec_correct if_minus_dec.
  Proof. verify. Qed.
```

Division

```
  Definition div_mod_dec (a b : nat) : decorated :=
    {{ fun st ⇒ True }} ->>
    {{ fun st ⇒ b * 0 + a = a }}
    X ::= a
    {{ fun st ⇒ b * 0 + st X = a }};;
    Y ::= 0
    {{ fun st ⇒ b * st Y + st X = a }};;
    WHILE b ≤ X DO
      {{ fun st ⇒ b * st Y + st X = a ∧ b ≤ st X }} ->>
      {{ fun st ⇒ b * (st Y + 1) + (st X - b) = a }}
      X ::= X - b
      {{ fun st ⇒ b * (st Y + 1) + st X = a }};;
      Y ::= Y + 1
      {{ fun st ⇒ b * st Y + st X = a }}
    END
    {{ fun st ⇒ b * st Y + st X = a ∧ ~(b ≤ st X) }} ->>
    {{ fun st ⇒ b * st Y + st X = a ∧ (st X < b) }}.

  Theorem div_mod_dec_correct : ∀ a b,
    dec_correct (div_mod_dec a b).
  Proof. intros a b. verify.
```

```
    rewrite mult_plus_distr_l. omega.
  Qed.
```

Parity

```
  Definition find_parity : com :=
    WHILE 2 ≤ X DO
       X ::= X - 2
    END.
```

There are actually several ways to phrase the loop invariant for this program. Here is one natural one, which leads to a rather long proof:

```
  Inductive ev : nat → Prop :=
    | ev_0 : ev O
    | ev_SS : ∀ n:nat, ev n → ev (S (S n)).

  Definition find_parity_dec m : decorated :=
     {{ fun st ⇒ st X = m}} ->>
     {{ fun st ⇒ st X ≤ m ∧ ev (m - st X) }}
    WHILE 2 ≤ X DO
       {{ fun st ⇒ (st X ≤ m ∧ ev (m - st X)) ∧ 2 ≤ st X }} ->>
       {{ fun st ⇒ st X - 2 ≤ m ∧ (ev (m - (st X - 2))) }}
       X ::= X - 2
       {{ fun st ⇒ st X ≤ m ∧ ev (m - st X) }}
    END
     {{ fun st ⇒ (st X ≤ m ∧ ev (m - st X)) ∧ st X < 2 }} ->>
     {{ fun st ⇒ st X=0 ↔ ev m }}.

  Lemma l1 : ∀ m n p,
    p ≤ n →
    n ≤ m →
    m - (n - p) = m - n + p.
  Proof. intros. omega. Qed.

  Lemma l2 : ∀ m,
    ev m →
    ev (m + 2).
  Proof. intros. rewrite plus_comm. simpl. constructor.
  assumption. Qed.

  Lemma l3' : ∀ m,
    ev m →
    ¬ev (S m).
  Proof. induction m; intros H1 H2. inversion H2. apply IHm.
         inversion H2; subst; assumption. assumption. Qed.

  Lemma l3 : ∀ m,
    1 ≤ m →
    ev m →
    ev (m - 1) →
    False.
  Proof. intros. apply l2 in H1.
         assert (G : m - 1 + 2 = S m). clear H0 H1. omega.
```

```
        rewrite G in H₁. apply l₃' in H₀. apply H₀. assumption.
Qed.

Theorem find_parity_correct : ∀ m,
  dec_correct (find_parity_dec m).
Proof.
  intro m. verify;
    (* simplification too aggressive ... reverting a bit *)
    fold (leb 2 (st X)) in *;
    try rewrite leb_iff in *;
    try rewrite leb_iff_conv in *; eauto; try omega.
    - (* invariant holds initially *)
      rewrite minus_diag. constructor.
    - (* invariant preserved *)
      rewrite l₁; try assumption.
      apply l₂; assumption.
    - (* invariant strong enough to imply conclusion
         (-> direction) *)
      rewrite <- minus_n_O in H₂. assumption.
    - (* invariant strong enough to imply conclusion
         (<- direction) *)
      destruct (st X) as [| [| n]].
      (* by H₁ X can only be 0 or 1 *)
      + (* st X = 0 *)
        reflexivity.
      + (* st X = 1 *)
        apply l₃ in H; try assumption. inversion H.
      + (* st X = 2 *)
        clear H₀ H₂. (* omega confused otherwise *)
        omega.
Qed.
```

Here is a more intuitive way of writing the invariant:

```
Definition find_parity_dec' m : decorated :=
  {{ fun st ⇒ st X = m}} ->>
  {{ fun st ⇒ ev (st X) ↔ ev m }}
 WHILE 2 ≤ X DO
    {{ fun st ⇒ (ev (st X) ↔ ev m) ∧ 2 ≤ st X }} ->>
    {{ fun st ⇒ (ev (st X - 2) ↔ ev m) }}
    X ::= X - 2
    {{ fun st ⇒ (ev (st X) ↔ ev m) }}
 END
 {{ fun st ⇒ (ev (st X) ↔ ev m) ∧ ~(2 ≤ st X) }} ->>
 {{ fun st ⇒ st X=0 ↔ ev m }}.

Lemma l₄ : ∀ m,
  2 ≤ m →
  (ev (m - 2) ↔ ev m).
Proof.
  induction m; intros. split; intro; constructor.
  destruct m. inversion H. inversion H₁. simpl in *.
  rewrite <- minus_n_O in *. split; intro.
    constructor. assumption.
```

```
      inversion H0. assumption.
  Qed.

  Theorem find_parity_correct' : ∀ m,
    dec_correct (find_parity_dec' m).
  Proof.
    intros m. verify;
      (* simplification too aggressive ... reverting a bit *)
      fold (leb 2 (st X)) in *;
      try rewrite leb_iff in *;
      try rewrite leb_iff_conv in *; intuition; eauto; try omega.
    - (* invariant preserved (part 1) *)
      rewrite l4 in H0; eauto.
    - (* invariant preserved (part 2) *)
      rewrite l4; eauto.
    - (* invariant strong enough to imply conclusion
           (-> direction) *)
      apply H0. constructor.
    - (* invariant strong enough to imply conclusion
           (<- direction) *)
        destruct (st X) as [| [| n]].
  (* by H1 X can only be 0 or 1 *)
        + (* st X = 0 *)
          reflexivity.
        + (* st X = 1 *)
          inversion H.
        + (* st X = 2 *)
          clear H0 H H3. (* omega confused otherwise *)
          omega.
  Qed.
```

Here is the simplest invariant we've found for this program:

```
  Definition parity_dec m : decorated :=
    {{ fun st ⇒ st X = m}} ->>
    {{ fun st ⇒ parity (st X) = parity m }}
   WHILE 2 ≤ X DO
      {{ fun st ⇒ parity (st X) = parity m ∧ 2 ≤ st X }} ->>
      {{ fun st ⇒ parity (st X - 2) = parity m }}
      X ::= X - 2
      {{ fun st ⇒ parity (st X) = parity m }}
   END
   {{ fun st ⇒ parity (st X) = parity m ∧ ~(2 ≤ st X) }} ->>
   {{ fun st ⇒ st X = parity m }}.

  Theorem parity_dec_correct : ∀ m,
    dec_correct (parity_dec m).
  Proof.
    intros. verify;
      (* simplification too aggressive ... reverting a bit *)
      fold (leb 2 (st X)) in *;
      try rewrite leb_iff in *;
      try rewrite leb_iff_conv in *; eauto; try omega.
    - (* invariant preserved *)
      rewrite <- H. apply parity_ge_2. assumption.
    - (* invariant strong enough *)
      rewrite <- H. symmetry. apply parity_lt_2. assumption.
  Qed.
```

Square Roots

```
Definition sqrt_dec m : decorated :=
    {{ fun st ⇒ st X = m }} ->>
    {{ fun st ⇒ st X = m ∧ 0*0 ≤ m }}
  Z ::= 0
    {{ fun st ⇒ st X = m ∧ st Z*st Z ≤ m }};;
  WHILE (Z+1)*(Z+1) ≤ X DO
      {{ fun st ⇒ (st X = m ∧ st Z*st Z≤m)
                   ∧ (st Z + 1)*(st Z + 1) ≤ st X }} ->>
      {{ fun st ⇒ st X = m ∧ (st Z+1)*(st Z+1)≤m }}
    Z ::= Z + 1
      {{ fun st ⇒ st X = m ∧ st Z*st Z≤m }}
  END
    {{ fun st ⇒ (st X = m ∧ st Z*st Z≤m)
                   ∧ ~((st Z + 1)*(st Z + 1) ≤ st X) }} ->>
    {{ fun st ⇒ st Z*st Z≤m ∧ m<(st Z+1)*(st Z+1) }}.

Theorem sqrt_correct : ∀ m,
  dec_correct (sqrt_dec m).
Proof. intro m. verify. Qed.
```

Squaring

Again, there are several ways of annotating the squaring program. The simplest variant we've found, `square_simpler_dec`, is given last.

```
Definition square_dec (m : nat) : decorated :=
  {{ fun st ⇒ st X = m }}
  Y ::= X
  {{ fun st ⇒ st X = m ∧ st Y = m }};;
  Z ::= 0
  {{ fun st ⇒ st X = m ∧ st Y = m ∧ st Z = 0}} ->>
  {{ fun st ⇒ st Z + st X * st Y = m * m }};;
  WHILE !(Y = 0) DO
    {{ fun st ⇒ st Z + st X * st Y = m * m ∧ st Y ≠ 0 }} ->>
    {{ fun st ⇒ (st Z + st X) + st X * (st Y - 1) = m * m }}
    Z ::= Z + X
    {{ fun st ⇒ st Z + st X * (st Y - 1) = m * m }};;
    Y ::= Y - 1
    {{ fun st ⇒ st Z + st X * st Y = m * m }}
  END
  {{ fun st ⇒ st Z + st X * st Y = m * m ∧ st Y = 0 }} ->>
  {{ fun st ⇒ st Z = m * m }}.

Theorem square_dec_correct : ∀ m,
  dec_correct (square_dec m).
Proof.
  intro n. verify.
  - (* invariant preserved *)
    destruct (st Y) as [| y']. apply False_ind. apply H0.
    reflexivity.
    simpl. rewrite <- minus_n_O.
    assert (G : ∀ n m, n * S m = n + n * m). {
      clear. intros. induction n. reflexivity. simpl.
      rewrite IHn. omega. }
    rewrite <- H. rewrite G. rewrite plus_assoc. reflexivity.
Qed.

Definition square_dec' (n : nat) : decorated :=
  {{ fun st ⇒ True }}
  X ::= n
  {{ fun st ⇒ st X = n }};;
  Y ::= X
  {{ fun st ⇒ st X = n ∧ st Y = n }};;
  Z ::= 0
  {{ fun st ⇒ st X = n ∧ st Y = n ∧ st Z = 0 }} ->>
  {{ fun st ⇒ st Z = st X * (st X - st Y)
               ∧ st X = n ∧ st Y ≤ st X }};;
  WHILE !(Y = 0) DO
    {{ fun st ⇒ (st Z = st X * (st X - st Y)
                ∧ st X = n ∧ st Y ≤ st X)
                 ∧ st Y ≠ 0 }}
    Z ::= Z + X
    {{ fun st ⇒ st Z = st X * (st X - (st Y - 1))
                 ∧ st X = n ∧ st Y ≤ st X }};;
    Y ::= Y - 1
    {{ fun st ⇒ st Z = st X * (st X - st Y)
                 ∧ st X = n ∧ st Y ≤ st X }}
  END
  {{ fun st ⇒ (st Z = st X * (st X - st Y)
              ∧ st X = n ∧ st Y ≤ st X)
               ∧ st Y = 0 }} ->>
  {{ fun st ⇒ st Z = n * n }}.

Theorem square_dec'_correct : ∀ n,
  dec_correct (square_dec' n).
Proof.
  intro n. verify.
  - (* invariant holds initially *)
    rewrite minus_diag. omega.
  - (* invariant preserved *) subst.
    rewrite mult_minus_distr_l.
    repeat rewrite mult_minus_distr_l. rewrite mult_1_r.
    assert (G : ∀ n m p,
                  m ≤ n → p ≤ m → n - (m - p) = n - m + p).
      intros. omega.
    rewrite G. reflexivity. apply mult_le_compat_l. assumption.
    destruct (st Y). apply False_ind. apply H0. reflexivity.
      clear. rewrite mult_succ_r. rewrite plus_comm.
      apply le_plus_l.
  - (* invariant + negation of guard imply
       desired postcondition *)
    rewrite <- minus_n_O. reflexivity.
Qed.

Definition square_simpler_dec (m : nat) : decorated :=
  {{ fun st ⇒ st X = m }} ->>
```

```
    {{ fun st ⇒ 0 = 0*m ∧ st X = m }}
    Y ::= 0
    {{ fun st ⇒ 0 = (st Y)*m ∧ st X = m }};;
    Z ::= 0
    {{ fun st ⇒ st Z = (st Y)*m ∧ st X = m }}->>
    {{ fun st ⇒ st Z = (st Y)*m ∧ st X = m }};;
    WHILE !(Y = X) DO
      {{ fun st ⇒ (st Z = (st Y)*m ∧ st X = m)
          ∧ st Y ≠ st X }} ->>
      {{ fun st ⇒ st Z + st X = ((st Y) + 1)*m ∧ st X = m }}
      Z ::= Z + X
      {{ fun st ⇒ st Z = ((st Y) + 1)*m ∧ st X = m }};;
      Y ::= Y + 1
      {{ fun st ⇒ st Z = (st Y)*m ∧ st X = m }}
    END
    {{ fun st ⇒ (st Z = (st Y)*m ∧ st X = m) ∧ st Y = st X }} ->>
    {{ fun st ⇒ st Z = m*m }}.

  Theorem square_simpler_dec_correct : ∀ m,
    dec_correct (square_simpler_dec m).
  Proof.
    intro m. verify.
    rewrite mult_plus_distr_r. simpl. rewrite <- plus_n_O.
    reflexivity.
  Qed.
```

Two loops

```
  Definition two_loops_dec (a b c : nat) : decorated :=
    {{ fun st ⇒ True }} ->>
    {{ fun st ⇒ c = 0 + c ∧ 0 = 0 }}
    X ::= 0
    {{ fun st ⇒ c = st X + c ∧ 0 = 0 }};;
    Y ::= 0
    {{ fun st ⇒ c = st X + c ∧ st Y = 0 }};;
    Z ::= c
    {{ fun st ⇒ st Z = st X + c ∧ st Y = 0 }};;
    WHILE !(X = a) DO
      {{ fun st ⇒ (st Z = st X + c ∧ st Y = 0) ∧ st X ≠ a }} ->>
      {{ fun st ⇒ st Z + 1 = st X + 1 + c ∧ st Y = 0 }}
      X ::= X + 1
      {{ fun st ⇒ st Z + 1 = st X + c ∧ st Y = 0 }};;
      Z ::= Z + 1
      {{ fun st ⇒ st Z = st X + c ∧ st Y = 0 }}
    END
    {{ fun st ⇒ (st Z = st X + c ∧ st Y = 0) ∧ st X = a }} ->>
    {{ fun st ⇒ st Z = a + st Y + c }};;
    WHILE !(Y = b) DO
      {{ fun st ⇒ st Z = a + st Y + c ∧ st Y ≠ b }} ->>
      {{ fun st ⇒ st Z + 1 = a + st Y + 1 + c }}
      Y ::= Y + 1
      {{ fun st ⇒ st Z + 1 = a + st Y + c }};;
      Z ::= Z + 1
      {{ fun st ⇒ st Z = a + st Y + c }}
    END
    {{ fun st ⇒ (st Z = a + st Y + c) ∧ st Y = b }} ->>
    {{ fun st ⇒ st Z = a + b + c }}.

  Theorem two_loops_correct : ∀ a b c,
    dec_correct (two_loops_dec a b c).
  Proof. intros a b c. verify. Qed.
```

Power Series

```
Fixpoint pow2 n :=
  match n with
  | 0 ⇒ 1
  | S n' ⇒ 2 * (pow2 n')
  end.

Definition dpow2_down (n: nat) :=
  {{ fun st ⇒ True }} ->>
  {{ fun st ⇒ 1 = (pow2 (0 + 1))-1 ∧ 1 = pow2 0 }}
  X ::= 0
  {{ fun st ⇒ 1 = (pow2 (0 + 1))-1 ∧ 1 = pow2 (st X) }};;
  Y ::= 1
  {{ fun st ⇒ st Y = (pow2 (st X + 1))-1 ∧ 1 = pow2 (st X) }};;
  Z ::= 1
  {{ fun st ⇒ st Y = (pow2 (st X + 1))-1 ∧ st Z = pow2 (st X)
}};;
  WHILE !(X = n) DO
    {{ fun st ⇒ (st Y = (pow2 (st X + 1))-1 ∧ st Z = pow2 (st X))
                ∧ st X ≠ n }} ->>
    {{ fun st ⇒ st Y + 2 * st Z = (pow2 (st X + 2))-1
                ∧ 2 * st Z = pow2 (st X + 1) }}
    Z ::= 2 * Z
    {{ fun st ⇒ st Y + st Z = (pow2 (st X + 2))-1
                ∧ st Z = pow2 (st X + 1) }};;
    Y ::= Y + Z
    {{ fun st ⇒ st Y = (pow2 (st X + 2))-1
                ∧ st Z = pow2 (st X + 1) }};;
    X ::= X + 1
    {{ fun st ⇒ st Y = (pow2 (st X + 1))-1
                ∧ st Z = pow2 (st X) }}
  END
  {{ fun st ⇒ (st Y = (pow2 (st X + 1))-1 ∧ st Z = pow2 (st X))
              ∧ st X = n }} ->>
  {{ fun st ⇒ st Y = pow2 (n+1) - 1 }}.

Lemma pow2_plus_1 : ∀ n,
  pow2 (n+1) = pow2 n + pow2 n.
Proof. induction n; simpl. reflexivity. omega. Qed.

Lemma pow2_le_1 : ∀ n, pow2 n ≥ 1.
Proof. induction n. simpl. constructor. simpl. omega. Qed.

Theorem dpow2_down_correct : ∀ n,
  dec_correct (dpow2_down n).
Proof.
  intro m. verify.
  - (* 1 *)
```

```
      rewrite pow2_plus_1. rewrite <- H0. reflexivity.
    - (* 2 *)
      rewrite <- plus_n_O.
      rewrite <- pow2_plus_1. remember (st X) as n.
      replace (pow2 (n + 1) - 1 + pow2 (n + 1))
         with (pow2 (n + 1) + pow2 (n + 1) - 1) by omega.
      rewrite <- pow2_plus_1.
      replace (n + 1 + 1) with (n + 2) by omega.
      reflexivity.
    - (* 3 *)
      rewrite <- plus_n_O. rewrite <- pow2_plus_1.
      reflexivity.
    - (* 4 *)
      replace (st X + 1 + 1) with (st X + 2) by omega.
      reflexivity.
  Qed.
```

Further Exercises

Exercise: 3 stars, advanced (slow_assignment_dec)

In the `slow_assignment` exercise above, we saw a roundabout way of assigning a number currently stored in `X` to the variable `Y`: start `Y` at `0`, then decrement `X` until it hits `0`, incrementing `Y` at each step. Write a formal version of this decorated program and prove it correct.

```
Example slow_assignment_dec (m:nat) : decorated
  (* REPLACE THIS LINE WITH ":= _your_definition_ ." *).
Admitted.

Theorem slow_assignment_dec_correct : ∀ m,
  dec_correct (slow_assignment_dec m).
Proof. (* FILL IN HERE *) Admitted.
```

☐

Exercise: 4 stars, advanced (factorial_dec)

Remember the factorial function we worked with before:

```
Fixpoint real_fact (n:nat) : nat :=
  match n with
  | O ⇒ 1
  | S n' ⇒ n * (real_fact n')
  end.
```

Following the pattern of `subtract_slowly_dec`, write a decorated program `factorial_dec` that implements the factorial function and prove it correct as `factorial_dec_correct`.

```
(* FILL IN HERE *)
```

☐

Exercise: 4 stars, advanced, optional (fib_eqn)

The Fibonacci function is usually written like this:

```
    Fixpoint fib n :=
      match n with
      | 0 ⇒ 1
      | 1 ⇒ 1
      | _ ⇒ fib (pred n) + fib (pred (pred n))
      end.
```

This doesn't pass Coq's termination checker, but here is a slightly clunkier definition that does:

```
Fixpoint fib n :=
  match n with
  | 0 ⇒ 1
  | S n' ⇒ match n' with
            | 0 ⇒ 1
            | S n'' ⇒ fib n' + fib n''
            end
  end.
```

Prove that `fib` satisfies the following equation:

```
Lemma fib_eqn : ∀ n,
  n > 0 →
  fib n + fib (Init.Nat.pred n) = fib (n + 1).
Proof.
  (* FILL IN HERE *) Admitted.
```

☐

Exercise: 4 stars, advanced, optional (fib)

The following Imp program leaves the value of `fib n` in the variable `Y` when it terminates:

```
    X ::= 1;;
    Y ::= 1;;
    Z ::= 1;;
    WHILE !(X = n+1) DO
      T ::= Z;
      Z ::= Z + Y;;
      Y ::= T;;
      X ::= X + 1
    END
```

Fill in the following definition of `dfib` and prove that it satisfies this specification:

```
      ⦃True⦄ dfib ⦃ Y = fib n ⦄
```

```
Definition T : string := "T".

Definition dfib (n:nat) : decorated
(* REPLACE THIS LINE WITH ":= _your_definition_ ." *). Admitted.
```

```
Theorem dfib_correct : ∀ n,
  dec_correct (dfib n).
(* FILL IN HERE *) Admitted.
```
☐

Exercise: 5 stars, advanced, optional (improve_dcom)

The formal decorated programs defined in this section are intended to look as similar as possible to the informal ones defined earlier in the chapter. If we drop this requirement, we can eliminate almost all annotations, just requiring final postconditions and loop invariants to be provided explicitly. Do this — i.e., define a new version of dcom with as few annotations as possible and adapt the rest of the formal development leading up to the `verification_correct` theorem.

```
(* FILL IN HERE *)
```
☐

Exercise: 4 stars, advanced, optional (implement_dcom)

Adapt the parser for Imp presented in chapter ImpParser of *Logical Foundations* to parse decorated commands (either ours or, even better, the ones you defined in the previous exercise).

```
(* FILL IN HERE *)
```
☐