CIS 301: Lecture Notes on Program Verification

Torben Amtoft
Department of Computing and Information Sciences
Kansas State University

May 5, 2003

These notes are written as a supplement to [1, Sect. 16.5], but can be read independently. The proof rules are inspired by the presentation in [3, Chap. 4]. Section 5 is inspired by Chapter 16 in [2], an excellent treatise on the subject of program construction.

1 Hoare Triples

To reason about correctness we shall consider *Hoare triples*, of the form

$$\{\phi\}\ P\ \{\psi\}$$

saying that if ϕ (the *precondition*) holds prior to executing program code P then ψ (the *postcondition*) holds afterwards.

Actually, the above description is ambiguous: what if P does not terminate? Therefore we shall distinguish between

partial correctness: *if* P terminates *then* ψ holds;

total correctness: P *does* terminate *and then* ψ holds.

In these notes, we shall interpret a Hoare triple as denoting *partial correctness* unless stated otherwise.

2 Software Engineering

In light of the notion of Hoare triples, one can think of software engineering as a 3-stage process:

1. *Translate* the demands D of the user into a *specification* (ϕ_D, ψ_D).
2. *Write* a program P that satisfies the specification constructed in 1.
3. *Prove* that it in fact holds that

$$\begin{array}{l}\{\phi_D\}\\ \quad P\\ \{\psi_D\}\end{array}$$

When it comes to software practice, 1 is a huge task (involving numerous discussions with the users) and hardly ever done completely. While 2 obviously has to be done, 3 is almost never carried out.

When it comes to academic discourse, 1 is an interesting task but only briefly touched upon (Section 3) in CIS 301. Instead, we shall focus on 3 (Section 4), but also give a few basic heuristics for how to do 2 and 3 *simultaneously* (Section 5).

3 Specifications

Square root

Suppose the user demands

> Compute the square root of `x` and store the result in `y`

As a first attempt, we may write the specification

$$\begin{array}{l}\quad P\\ \{\mathtt{y}^2 = \mathtt{x}\}\end{array}$$

We now remember that we cannot compute the square root of negative numbers and therefore add a precondition:

$\{\mathtt{x} \geq 0\}$
$\quad P$
$\{\mathtt{y}^2 = \mathtt{x}\}$

Then we realize that if `x` is not a square then we have to settle for an approximation (since we are working with integers):

$\{\mathtt{x} \geq 0\}$
$\quad P$
$\{\mathtt{y}^2 \leq \mathtt{x}\}$

On the other hand, this is too liberal: we could just pick `y` to be zero. Therefore, we must also specify that `y` is the largest number that does the job:

$\{\mathtt{x} \geq 0\}$
$\quad P$
$\{\mathtt{y}^2 \leq \mathtt{x} \wedge (\mathtt{y}+1)^2 > \mathtt{x}\}$

which seems a sensible specification of the square root program. (Observe that it entails that `y` has to be non-negative. Why?)

Factorial

Now assume that the user demands

> Ensure that `y` contains the factorial of `x`.

(Remember that the factorial function is defined by

$$\begin{aligned} \mathrm{fac}(0) &= 1 \\ \mathrm{fac}(n+1) &= (n+1)\mathrm{fac}(n) \text{ for } n \geq 0 \end{aligned}$$

and thus

$$\mathrm{fac}(0) = 1,\ \mathrm{fac}(1) = 1,\ \mathrm{fac}(2) = 2,\ \mathrm{fac}(3) = 6,\ \mathrm{fac}(4) = 24,\ \text{etc.})$$

It therefore seems that this should produce the specification

$\{\mathtt{x} \geq 0\}$
$\quad P$
$\{\mathtt{y} = \mathrm{fac}(\mathtt{x})\}$

Well, it's not hard to write a program satisfying this specification:

$\{\mathtt{x} \geq 0\}$
$\quad \mathtt{x} := 4;$
$\quad \mathtt{y} := 24$
$\{\mathtt{y} = \mathrm{fac}(\mathtt{x})\}$

The user may respond:

> Hey, that's cheating! You were not allowed to change `x`.

Well, if not that better has to be part of the specification! But how to incorporate such demands? We shall need the concept of *logical* variables: these do *not* occur in programs, only in specifications, and are written with a subscript.

Using the logical variable x_0 to denote the initial (and un-changed) value of `x`, a program computing factorials can be specified as follows:

$\{\mathtt{x} = x_0 \geq 0\}$
$\quad P$
$\{\mathtt{y} = \mathrm{fac}(x_0) \wedge \mathtt{x} = x_0\}$

Similarly, the specification of the square root program can be modified.

4 Proving Correctness

We shall now discuss how to verify a claim that

$\{\phi\}$
$\quad P$
$\{\psi\}$

where P is a program written in a language with the following simple[1] syntax:

$$\begin{array}{rcl} C & ::= & x := E \\ & | & C_1;\ C_2 \\ & | & \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2\ \texttt{fi} \\ & | & \texttt{while}\ B\ \texttt{do}\ C\ \texttt{od} \end{array}$$

where B stands for *boolean tests* of the form $E_1 < E_2$, $E_1 \leq E_2$, $E_1 \neq E_2$, etc; and E stands for *integer expressions* of the form n (a constant), x (a variable), $E_1 + E_2$, $E_1 - E_2$, etc. Programs are thus constructed from assignments, sequential composition, conditionals, and while-loops[2].

4.1 Invariants

For the purpose of verification, the notion of *invariants* is crucial.

4.1.1 Motivating example

We look at the following program for computing the factorial function[3] (cf. Section 3).

$\{\texttt{x} \geq 0\}$
    $\texttt{y} := 1;$
    $\texttt{z} := 0;$
    $\texttt{while}\ \texttt{z} \neq \texttt{x}\ \texttt{do}$
        $\texttt{z} := \texttt{z} + 1;$
        $\texttt{y} := \texttt{y} * \texttt{z}$
    $\texttt{od}$
$\{\texttt{y} = \text{fac}(\texttt{x})\}$

[1] Many desirable language features (such as procedures) are absent from our language which, however, is "Turing-complete" in that it can *encode* all other features one can imagine!

[2] Note that we use the symbol `od` to end while-loops, rather than a curly bracket as this symbol is used for writing pre- and post-conditions. Similarly, we use `fi` as a delimiter for conditionals.

[3] Since the program does not change the value of `x`, we can safely write its specification without employing a logical variable x_0.

There are many mistakes we could have made when writing that program: for instance we could have reversed the two lines in the loop body (in which case y would be assigned zero and keep that value forever), or we could have written the loop test as $\mathtt{z} \leq \mathtt{x}$ (in which case y would end up containing $\mathrm{fac}(x+1)$).

Let us now convince ourselves that what we wrote is correct. We might first try a simulation: if say $x = 4$, the situation at the entry of the loop is:

	x	y	z
After 0 iterations	4	1	0
After 1 iterations	4	1	1
After 2 iterations	4	2	2
After 3 iterations	4	6	3
After 4 iterations	4	24	4

and then $\mathtt{z} = \mathtt{x}$ so that the loop terminates, with y containing the desired result $24 = \mathrm{fac}(\mathtt{x})$. This may boost our confidence in the program, but still a general proof is needed. Fortunately, the table above may help us in that endeavor. For it suggests that it is always the case that $\mathtt{y} = \mathrm{fac}(\mathtt{z})$.

Definition 4.1. A property which holds each time the loop test is evaluated is called an *invariant* for the loop. □

Equivalently, a property ψ is an invariant for a loop iff ψ holds after any number (≥ 0) of loop iterations.

We now annotate the program with our prospective invariant:

$\{\mathtt{x} \geq 0\}$
    y := 1;
    z := 0;
$\{\mathtt{y} = \mathrm{fac}(\mathtt{z})\}$
    while $\mathtt{z} \neq \mathtt{x}$ **do**
        z := z + 1;
        y := y ∗ z
    od
$\{\mathtt{y} = \mathrm{fac}(\mathtt{x})\}$

Of course, we must prove that what we have found is indeed an invariant:

Proposition 4.2. *For all $k \geq 0$, after k iterations of the loop it holds that* $\mathtt{y} = \mathrm{fac}(\mathtt{z})$. □

This proposition almost begs for a proof by induction!

Establishing the invariant. The base step amounts to checking that $\mathtt{y} = \text{fac}(\mathtt{z})$ after 0 iterations. But since the preamble assigns `y` the value 1 and assigns `z` the value 0, the claim follows from the fact that $\text{fac}(0) = 1$.

Maintaining the invariant. The inductive step amounts to checking that if $\mathtt{y} = \text{fac}(\mathtt{z})$ holds after k iterations, then it also holds after $k+1$ iterations. With $\mathtt{y}'$ denoting the value of `y` after $k+1$ iterations, and $\mathtt{z}'$ denoting the value of `z` after $k+1$ iterations, this follows since

$$\mathtt{y}' = \mathtt{y}\mathtt{z}' = \mathtt{y}(\mathtt{z}+1) = \text{fac}(\mathtt{z})(\mathtt{z}+1) = \text{fac}(\mathtt{z}+1) = \text{fac}(\mathtt{z}')$$

where for the third equality we have used the induction hypothesis, and for the fourth equality we have used the definition of the factorial function.

Completing the correctness proof. We have shown that every time the loop test is evaluated, it holds that $\mathtt{y} = \text{fac}(\mathtt{z})$. If (when!) we eventually exit the loop then the loop test is `false`, that is $\mathtt{z} = \mathtt{x}$. Therefore, if (when) the program terminates it holds that $\mathtt{y} = \text{fac}(\mathtt{x})$. This shows that our program satisfies its specification!

Recall from Section 1, however, that our focus is on partial correctness. Still, in this case total correctness is not hard to prove: since `x` is initially[4] non-negative, and since `z` is initialized to zero and incremented by one at each iteration, eventually `z` will equal `x`, causing the loop to terminate.

4.1.2 Proof Principles for Invariants

From the previous subsection we see that three steps are involved when proving that a certain property ψ is indeed an invariant for a loop:

1. we must show that the code before the loop establishes ψ;
2. we must show that ψ is maintained after each iteration;
3. we must show that if the loop test evaluates to `false`, ψ is sufficient to establish the desired postcondition.

[4]It is interesting that the proof of partial correctness did not use the precondition $\mathtt{x} \geq 0$.

4.2 Annotated Programs

We have argued that annotating a program with loop invariants is essential for the purpose of verification (and also to understand how the program works!) It is often beneficial to provide more fine-grained annotations.

EXAMPLE 4.3. For our factorial program, a fully annotated version looks like

$\{\mathtt{x} \geq 0\}$ (A)
$\{1 = \mathrm{fac}(0)\}$ (B)
    $\mathtt{y} := 1;$
$\{\mathtt{y} = \mathrm{fac}(0)\}$ (C)
    $\mathtt{z} := 0;$
$\{\mathtt{y} = \mathrm{fac}(\mathtt{z})\}$ (D)
    `while` $\mathtt{z} \neq \mathtt{x}$ `do`
$\{\mathtt{y} = \mathrm{fac}(\mathtt{z}) \wedge \mathtt{z} \neq \mathtt{x}\}$ (E)
$\{\mathtt{y}(\mathtt{z}+1) = \mathrm{fac}(\mathtt{z}+1)\}$ (F)
        $\mathtt{z} := \mathtt{z} + 1;$
$\{\mathtt{yz} = \mathrm{fac}(\mathtt{z})\}$ (G)
        $\mathtt{y} := \mathtt{y} * \mathtt{z}$
$\{\mathtt{y} = \mathrm{fac}(\mathtt{z})\}$ (H)
    `od`
$\{\mathtt{y} = \mathrm{fac}(\mathtt{z}) \wedge \mathtt{z} = \mathtt{x}\}$ (I)
$\{\mathtt{y} = \mathrm{fac}(\mathtt{x})\}$ (J)

In the next section, we shall see that this program is in fact *well-annotated*. □

4.3 Well-Annotated Programs and Valid Annotations

We first define what it means for an annotation to be *valid*. There are several cases:

Logical consequence. If the annotation $\{\psi\}$ immediately follows the annotation $\{\phi\}$, and ψ is a logical consequence of ϕ, then ψ is valid.

Trying to conform with the notation used in [1], we can write this rule as

$\{\phi\}$
$\triangleright$ $\{\psi\}$ **Implies** (if ψ logical consequence of ϕ)

saying that the marked annotation is valid.

Of course, in order to trust that ψ holds, we must at some point also establish that ϕ is valid!

Example 4.4. Referring back to Example 4.3, note that thanks to this rule

- annotation (B) is valid, since it is a logical consequence of annotation (A);
- annotation (F) is valid, since it is a logical consequence of annotation (E);
- annotation (J) is valid, since it is a logical consequence of annotation (I).

□

Rule for While loops. We have the rule

$\{\psi\}$
`while` B `do`
▷ $\{\psi \wedge B\}$ **WhileTrue**
$\ldots$
$\{\psi\}$
`od`
▷ $\{\psi \wedge \neg B\}$ **WhileFalse**

saying that if ψ is a loop invariant then

- at the beginning of the loop body, the loop test has just evaluated to `true` and therefore $\psi \wedge B$ will hold;
- immediately after the loop, the loop test has just evaluated to `false` and therefore $\psi \wedge \neg B$ will hold.

Note that we are still left with the obligation to show that the two ψ annotations (one before the loop, the other at the end of the loop body) are valid.

Example 4.5. Referring back to Example 4.3, note that annotations (E) and (I) are valid, thanks to this rule. □

Rule for Conditionals. We have the rule

$\{\phi\}$
`if` B
`then`
$\triangleright$ $\{\phi \wedge B\}$ **IfTrue**
$\ldots$
$\{\psi\}$
`else`
$\triangleright$ $\{\phi \wedge \neg B\}$ **IfFalse**
$\ldots$
$\{\psi\}$
`fi`
$\triangleright$ $\{\psi\}$ **IfEnd**

saying that if ϕ holds before a conditional statement then

- at the beginning of the `then` branch, $\phi \wedge B$ will hold;
- at the beginning of the `else` branch, $\phi \wedge \neg B$ will hold;

and also saying that ψ holds after the conditional statement if ψ holds at the end of both branches.

Again, we are left with the obligation to show that the initial ϕ annotation is valid, and that the ψ annotations concluding each branch are valid.

Observe that this rule is quite similar to the rule $\vee$ **Elim** from propositional logic!

Rule for Assignments We would surely expect that it for instance holds that

$\{\mathtt{y} = 5\}$
$\quad \mathtt{x} := \mathtt{y} + 2$
$\{\mathtt{x} = 7 \wedge \mathtt{y} = 5\}$

and it seems straightforward to go from precondition to postcondition. But now consider

$\{\mathtt{y} + 2\mathtt{z} \leq 3 \wedge \mathtt{z} \geq 1\}$
$\quad \mathtt{x} := \mathtt{y} + \mathtt{z}$
$\{???\}$

where it is by no means a simple mechanical procedure to fill in the question marks: what does the precondition imply concerning the value of $\mathtt{y} + \mathtt{z}$?

It turns out that we shall formulate the proper rule *backwards*: if we assign $\mathtt{x}$ the expression E, and we want $\psi(x)$ to hold *after* the assignment, we better demand that $\psi(E)$ holds *before* the assignment! This motivates the rule[5]

$\{\psi(E)\}$
$\quad \mathtt{x} := E$
$\triangleright \quad \{\psi(x)\}$ **Assignment**

Referring back to our first example, we have

$\{\mathtt{y} = 5\}$
$\{\mathtt{y} + 2 = 7 \wedge \mathtt{y} = 5\}$ **Implies**
$\quad \mathtt{x} := \mathtt{y} + 2$
$\{\mathtt{x} = 7 \wedge \mathtt{y} = 5\}$ **Assignment**

And referring back to our second example, we have

$\{\mathtt{y} + 2\mathtt{z} \leq 3 \wedge \mathtt{z} \geq 1\}$
$\{\mathtt{y} + \mathtt{z} \leq 2\}$ **Implies**
$\quad \mathtt{x} := \mathtt{y} + \mathtt{z}$
$\{x \leq 2\}$ **Assignment**

since it is easy to check that if $y + 2z \leq 3$ and $z \geq 1$ then $y + z \leq 2$.

Example 4.6. Referring back to Example 4.3, note that annotations (C), (D), (G), and (H) are valid, thanks to this rule. □

Well-annotation. We are now done with all the rules for validity. Note that there is no need for a rule for sequential composition $C_1; C_2$, since in an annotation

[5] As usual, we let $\psi(x)$ denote a formula where x is possibly free, and let $\psi(E)$ denote the result of substituting E for the free occurrences of x.

$\{\phi\}$
$\quad C_1;$
$\{\phi_1\}$
$\quad C_2$
$\{\phi_2\}$

the validity of each ϕ_i $(i = 1, 2)$ must be established using the form of C_i. But there is a rule for all other language constructs, and also a rule **Implies** that is not related to any specific language construct.

We are now ready to assemble the pieces:

Definition 4.7. We say that an annotated program

$\{\phi\}$
$\quad \ldots$
$\{\psi\}$

is *well-annotated* iff all annotations, *except* for the precondition ϕ, are valid. □

Theorem 4.8. *Assume that the annotated program*

$\{\phi\}$
$\quad \ldots$
$\{\psi\}$

is in fact well-annotated. Then the program satisfies its specification (ϕ,ψ). □

EXAMPLE 4.9. The program in Example 4.3 is well-annotated. This follows from Examples 4.4, 4.5, and 4.6. We can write

$\{\mathtt{x} \geq 0\}$	
$\{1 = \mathrm{fac}(0)\}$	**Implies**
$\mathtt{y} := 1;$	
$\{\mathtt{y} = \mathrm{fac}(0)\}$	**Assignment**
$\mathtt{z} := 0;$	
$\{\mathtt{y} = \mathrm{fac}(\mathtt{z})\}$	**Assignment**
`while` $\mathtt{z} \neq \mathtt{x}$ `do`	
$\{\mathtt{y} = \mathrm{fac}(\mathtt{z}) \wedge \mathtt{z} \neq \mathtt{x}\}$	**WhileTrue**
$\{\mathtt{y}(\mathtt{z}+1) = \mathrm{fac}(\mathtt{z}+1)\}$	**Implies**
$\mathtt{z} := \mathtt{z} + 1;$	
$\{\mathtt{yz} = \mathrm{fac}(\mathtt{z})\}$	**Assignment**
$\mathtt{y} := \mathtt{y} * \mathtt{z}$	
$\{\mathtt{y} = \mathrm{fac}(\mathtt{z})\}$	**Assignment**
`od`	
$\{\mathtt{y} = \mathrm{fac}(\mathtt{z}) \wedge \mathtt{z} = \mathtt{x}\}$	**WhileFalse**
$\{\mathtt{y} = \mathrm{fac}(\mathtt{x})\}$	**Implies**

□

5 Developing a Correct Program

In Section 4, we considered the situation where we must prove the correctness of a program which has *already* been written for a given specification. This two-step approach has some drawbacks:

- it gives us no clue about how actually to *construct* programs;
- if the program in question has been developed in an unsystematic way, perhaps by someone else, it may be hard to detect the proper invariant(s).

In this section, we shall illustrate that it is often possible to write a program *together* with the proof of its correctness.

For that purpose, we look at the square root specification[6] from Section 3:

[6]We have not bothered to employ the device of logical variables, and must therefore solemnly promise that the program to be constructed will not modify the value of `x`.

$\{\mathtt{x} \geq 0\}$
$\quad P$
$\{\mathtt{y}^2 \leq \mathtt{x} \wedge (\mathtt{y}+1)^2 > \mathtt{x}\}$

It seems reasonable to assume that P should be a loop, possibly with some preamble. With ϕ the (yet unknown) invariant of that loop, we have the skeleton

$\{\mathtt{x} \geq 0\}$
$\quad ???$
$\{\phi\}$
$\quad \mathtt{while}\ B\ \mathtt{do}$
$\{\phi \wedge B\}$ **WhileTrue**
$\quad ???$
$\{\phi\}$
$\quad \mathtt{od}$
$\{\phi \wedge \neg B\}$ **WhileFalse**
$\{\mathtt{y}^2 \leq \mathtt{x} \wedge (\mathtt{y}+1)^2 > \mathtt{x}\}$

We now face the main challenge: to come up with a suitable invariant ϕ, the form of which will direct the remaining construction process. In order to justify the postcondition, we must ensure that

$$\mathtt{y}^2 \leq \mathtt{x} \wedge (\mathtt{y}+1)^2 > \mathtt{x} \text{ is a logical consequence of } \phi \wedge \neg B \qquad (1)$$

There are at least two ways to achieve that, to be described in the next two subsections.

5.1 Deleting a Conjunct

A simple way to satisfy (1) is to define

$$\begin{aligned} \phi &= \mathtt{y}^2 \leq \mathtt{x} \\ B &= (\mathtt{y}+1)^2 \leq \mathtt{x} \end{aligned}$$

That is, we follow the following general recipe:

- let the loop test be the negation of one of the conjuncts of the postcondition;
- let the loop invariant be the remaining conjuncts of the postcondition.

Our prospective annotated program now looks like

$\{\mathtt{x} \geq 0\}$
    ???
$\{\mathtt{y}^2 \leq \mathtt{x}\}$
    `while` $(\mathtt{y}+1)^2 \leq \mathtt{x}$ `do`
$\{\mathtt{y}^2 \leq \mathtt{x} \wedge (\mathtt{y}+1)^2 \leq \mathtt{x}\}$ **WhileTrue**
        ???
$\{\mathtt{y}^2 \leq \mathtt{x}\}$
    `od`
$\{\mathtt{y}^2 \leq \mathtt{x} \wedge (\mathtt{y}+1)^2 > \mathtt{x}\}$ **WhileFalse**

The obvious way to establish the loop invariant is to initialize `y` to zero, leaving us with the now nearly completed program

$\{\mathtt{x} \geq 0\}$
$\{0^2 \leq \mathtt{x}\}$ **Implies**
    $\mathtt{y} := 0$
$\{\mathtt{y}^2 \leq \mathtt{x}\}$ **Assignment**
    `while` $(\mathtt{y}+1)^2 \leq \mathtt{x}$ `do`
$\{\mathtt{y}^2 \leq \mathtt{x} \wedge (\mathtt{y}+1)^2 \leq \mathtt{x}\}$ **WhileTrue**
        ???
$\{\mathtt{y}^2 \leq \mathtt{x}\}$
    `od`
$\{\mathtt{y}^2 \leq \mathtt{x} \wedge (\mathtt{y}+1)^2 > \mathtt{x}\}$ **WhileFalse**

And the obvious way to maintain the loop invariant is to increment `y` by one, resulting in the well-annotated program

$\{\mathtt{x} \geq 0\}$
$\{0^2 \leq \mathtt{x}\}$ **Implies**
    $\mathtt{y} := 0$
$\{\mathtt{y}^2 \leq \mathtt{x}\}$ **Assignment**
    `while` $(\mathtt{y}+1)^2 \leq \mathtt{x}$ `do`
$\{\mathtt{y}^2 \leq \mathtt{x} \wedge (\mathtt{y}+1)^2 \leq \mathtt{x}\}$ **WhileTrue**
$\{(\mathtt{y}+1)^2 \leq \mathtt{x}\}$ **Implies**
        $\mathtt{y} := \mathtt{y}+1$
$\{\mathtt{y}^2 \leq \mathtt{x}\}$ **Assignment**
    `od`
$\{\mathtt{y}^2 \leq \mathtt{x} \wedge (\mathtt{y}+1)^2 > \mathtt{x}\}$ **WhileFalse**

This program will clearly always terminate, but is rather inefficient. We shall now describe a method which in this case results in a more efficient program.

5.2 Replacing an Expression By a Variable

Let us consider another way of satisfying (1). First observe that the postcondition involves the expression $\mathtt{y}$ as well as the expression $\mathtt{y}+1$. It might be beneficial to loosen the connection between these two entities, by introducing a new variable $\mathtt{w}$ which eventually should equal $\mathtt{y}+1$ but in the meantime may roam more freely. Note that the postcondition is implied by the formula

$$\mathtt{y}^2 \leq \mathtt{x} \wedge \mathtt{w}^2 > \mathtt{x} \wedge \mathtt{w} = \mathtt{y}+1$$

containing three conjuncts. It is thus tempting to apply the previous technique of "deleting a conjunct", resulting in

$$\begin{aligned} \phi &= \mathtt{y}^2 \leq \mathtt{x} \wedge \mathtt{w}^2 > \mathtt{x} \\ B &= \mathtt{w} \neq \mathtt{y}+1 \end{aligned}$$

Our prospective annotated program now looks like

$\{\mathtt{x} \geq 0\}$
???
$\{\mathtt{y}^2 \leq \mathtt{x} \wedge \mathtt{w}^2 > \mathtt{x}\}$
`while` $\mathtt{w} \neq \mathtt{y}+1$ `do`
$\{\mathtt{y}^2 \leq \mathtt{x} \wedge \mathtt{w}^2 > \mathtt{x} \wedge \mathtt{w} \neq \mathtt{y}+1\}$ **WhileTrue**
???
$\{\mathtt{y}^2 \leq \mathtt{x} \wedge \mathtt{w}^2 > \mathtt{x}\}$
`od`
$\{\mathtt{y}^2 \leq \mathtt{x} \wedge \mathtt{w}^2 > \mathtt{x} \wedge \mathtt{w} = \mathtt{y}+1\}$ **WhileFalse**
$\{\mathtt{y}^2 \leq \mathtt{x} \wedge (\mathtt{y}+1)^2 > \mathtt{x}\}$ **Implies**

To establish the loop invariant, we must not only initialize $\mathtt{y}$ to zero but also initialize $\mathtt{w}$ so that $\mathtt{w}^2 > \mathtt{x}$: clearly, $\mathtt{x}+1$ will do the job.

$\{\mathtt{x} \geq 0\}$
$\{0^2 \leq \mathtt{x} \wedge (\mathtt{x}+1)^2 > \mathtt{x}\}$ **Implies**
    $\mathtt{y} := 0$
$\{\mathtt{y}^2 \leq \mathtt{x} \wedge (\mathtt{x}+1)^2 > \mathtt{x}\}$ **Assignment**
    $\mathtt{w} := \mathtt{x}+1$
$\{\mathtt{y}^2 \leq \mathtt{x} \wedge \mathtt{w}^2 > \mathtt{x}\}$ **Assignment**
    `while` $\mathtt{w} \neq \mathtt{y}+1$ `do`
        ...

For the loop body, it seems a sensible choice to modify *either* `y` *or* `w`. This can be expressed as a conditional of the form

`if` B' `then`
    $\mathtt{y} := E_1$
`else`
    $\mathtt{w} := E_2$
`fi`

We now plug that into the main program. Even without knowing E_1, E_2, or B', several extra annotations can be provided:

        ...
$\{\mathtt{y}^2 \leq \mathtt{x} \wedge \mathtt{w}^2 > \mathtt{x}\}$
    `while` $\mathtt{w} \neq \mathtt{y}+1$ `do`
$\{\mathtt{y}^2 \leq \mathtt{x} \wedge \mathtt{w}^2 > \mathtt{x} \wedge \mathtt{w} \neq \mathtt{y}+1\}$ **WhileTrue**
        `if` B' `then`
$\{\mathtt{y}^2 \leq \mathtt{x} \wedge \mathtt{w}^2 > \mathtt{x} \wedge \mathtt{w} \neq \mathtt{y}+1 \wedge B'\}$ **IfTrue**
$\{\mathtt{E_1}^2 \leq \mathtt{x} \wedge \mathtt{w}^2 > \mathtt{x}\}$ (A)
            $\mathtt{y} := E_1$
$\{\mathtt{y}^2 \leq \mathtt{x} \wedge \mathtt{w}^2 > \mathtt{x}\}$ **Assignment**
        `else`
$\{\mathtt{y}^2 \leq \mathtt{x} \wedge \mathtt{w}^2 > \mathtt{x} \wedge \mathtt{w} \neq \mathtt{y}+1 \wedge \neg B'\}$ **IfFalse**
$\{\mathtt{y}^2 \leq \mathtt{x} \wedge \mathtt{E_2}^2 > \mathtt{x}\}$ (B)
            $\mathtt{w} := E_2$
$\{\mathtt{y}^2 \leq \mathtt{x} \wedge \mathtt{w}^2 > \mathtt{x}\}$ **Assignment**
        `fi`
$\{\mathtt{y}^2 \leq \mathtt{x} \wedge \mathtt{w}^2 > \mathtt{x}\}$ **IfEnd**
    `od`
$\{\mathtt{y}^2 \leq \mathtt{x} \wedge \mathtt{w}^2 > \mathtt{x} \wedge \mathtt{w} = \mathtt{y}+1\}$ **WhileFalse**
$\{\mathtt{y}^2 \leq \mathtt{x} \wedge (\mathtt{y}+1)^2 > \mathtt{x}\}$ **Implies**

We still have to justify annotations (A) and (B). But they will follow by **Implies**, provided there exists an expression E such that

$$\begin{aligned} E_1 &= E \\ E_2 &= E \\ B' &= E^2 \leq \mathtt{x} \end{aligned}$$

We have thus constructed the well-annotated program

$\{\mathtt{x} \geq 0\}$
$\{0^2 \leq \mathtt{x} \wedge (\mathtt{x}+1)^2 > \mathtt{x}\}$ **Implies**
  $\mathtt{y} := 0$
$\{\mathtt{y}^2 \leq \mathtt{x} \wedge (\mathtt{x}+1)^2 > \mathtt{x}\}$ **Assignment**
  $\mathtt{w} := \mathtt{x}+1$
$\{\mathtt{y}^2 \leq \mathtt{x} \wedge \mathtt{w}^2 > \mathtt{x}\}$ **Assignment**
  `while` $\mathtt{w} \neq \mathtt{y}+1$ `do`
$\{\mathtt{y}^2 \leq \mathtt{x} \wedge \mathtt{w}^2 > \mathtt{x} \wedge \mathtt{w} \neq \mathtt{y}+1\}$ **WhileTrue**
    `if` $\mathtt{E}^2 \leq \mathtt{x}$
    `then`
$\{\mathtt{y}^2 \leq \mathtt{x} \wedge \mathtt{w}^2 > \mathtt{x} \wedge \mathtt{w} \neq \mathtt{y}+1 \wedge \mathtt{E}^2 \leq \mathtt{x}\}$ **IfTrue**
$\{\mathtt{E}^2 \leq \mathtt{x} \wedge \mathtt{w}^2 > \mathtt{x}\}$ **Implies**
      $\mathtt{y} := E$
$\{\mathtt{y}^2 \leq \mathtt{x} \wedge \mathtt{w}^2 > \mathtt{x}\}$ **Assignment**
    `else`
$\{\mathtt{y}^2 \leq \mathtt{x} \wedge \mathtt{w}^2 > \mathtt{x} \wedge \mathtt{w} \neq \mathtt{y}+1 \wedge \mathtt{E}^2 > \mathtt{x}\}$ **IfFalse**
$\{\mathtt{y}^2 \leq \mathtt{x} \wedge \mathtt{E}^2 > \mathtt{x}\}$ **Implies**
      $\mathtt{w} := E$
$\{\mathtt{y}^2 \leq \mathtt{x} \wedge \mathtt{w}^2 > \mathtt{x}\}$ **Assignment**
    `fi`
$\{\mathtt{y}^2 \leq \mathtt{x} \wedge \mathtt{w}^2 > \mathtt{x}\}$ **IfEnd**
  `od`
$\{\mathtt{y}^2 \leq \mathtt{x} \wedge \mathtt{w}^2 > \mathtt{x} \wedge \mathtt{w} = \mathtt{y}+1\}$ **WhileFalse**
$\{\mathtt{y}^2 \leq \mathtt{x} \wedge (\mathtt{y}+1)^2 > \mathtt{x}\}$ **Implies**

We are left with deciding which E to use. For partial correctness, we have just seen that any choice will do. But of course, we want to ensure termination, and hopefully a quick such! For that purpose, we pick

$$E = (\mathtt{y} + \mathtt{w})\ \mathtt{div}\ 2$$

where a `div` b (for positive b) is the largest integer c such that $bc \leq a$. With that choice, it is not difficult to see that `y` and `w` will get closer to each other

for each iteration, until eventually $\mathtt{w} = \mathtt{y} + 1$. This shows total correctness. Even more, the program runs much faster than our first attempt!

References

[1] Jon Barwise and John Etchemendy. *Language, Proof and Logic.* CSLI Publications, 1999.

[2] David Gries. *The Science of Programming.* Springer-Verlag, 1981.

[3] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and reasoning about systems.* Cambridge University Press, 2000.

A Typical Exam Questions

Question I (Fall 2002)

Given a positive integer x, we may define its integer logarithm as the largest integer y with the property that $2^y \leq x$. (Example: the integer logarithm of 10 is 3, since $2^3 = 8 \leq 10$ but $2^4 = 16 > 10$.)

We claim that if the program below terminates then `y` will denote the integer logarithm of `x`. Prove this claim by well-annotating the program. This includes

a. formalizing the desired postcondition of the program;

b. coming up with a suitable invariant for the `while` loop.

You can ignore the implicit demand that the value of `x` should not change (that is, don't bother about introducing a logical variable x_0).

```
y := 0;

w := 2;

while w ≤ x do

    y := y + 1;

    w := 2 * w

od
```

Proposed Answer for I

The desired postcondition is

$$\{2^{\mathtt{y}} \leq \mathtt{x} \wedge 2^{\mathtt{y}+1} > \mathtt{x}\}$$

and is satisfied, as shown by the well-annotation

$\{\mathtt{x} \geq 1\}$	Precondition
$\{2^0 \leq \mathtt{x} \wedge 2 = 2^{0+1}\}$	**Implies**
$\mathtt{y} := 0;$	
$\{2^{\mathtt{y}} \leq \mathtt{x} \wedge 2 = 2^{\mathtt{y}+1}\}$	**Assignment**
$\mathtt{w} := 2;$	
$\{2^{\mathtt{y}} \leq \mathtt{x} \wedge \mathtt{w} = 2^{\mathtt{y}+1}\}$	**Assignment**
`while` $\mathtt{w} \leq \mathtt{x}$ `do`	
$\{2^{\mathtt{y}} \leq \mathtt{x} \wedge \mathtt{w} = 2^{\mathtt{y}+1} \wedge \mathtt{w} \leq \mathtt{x}\}$	**WhileTrue**
$\{2^{\mathtt{y}+1} \leq \mathtt{x} \wedge 2\mathtt{w} = 2^{\mathtt{y}+1+1}\}$	**Implies**
$\mathtt{y} := \mathtt{y} + 1;$	
$\{2^{\mathtt{y}} \leq \mathtt{x} \wedge 2\mathtt{w} = 2^{\mathtt{y}+1}\}$	**Assignment**
$\mathtt{w} := 2 * \mathtt{w}$	
$\{2^{\mathtt{y}} \leq \mathtt{x} \wedge \mathtt{w} = 2^{\mathtt{y}+1}\}$	**Assignment**
`od`	
$\{2^{\mathtt{y}} \leq \mathtt{x} \wedge \mathtt{w} = 2^{\mathtt{y}+1} \wedge \mathtt{w} > \mathtt{x}\}$	**WhileFalse**
$\{2^{\mathtt{y}} \leq \mathtt{x} \wedge 2^{\mathtt{y}+1} > \mathtt{x}\}$	**Implies**