A framework for software verification [H&R, p263]

The principle behind verifying a procedural program is: *proving that a sequence of commands establishes appropriate relationships between the machine states pre- and post- execution*. This principle will be illustrated as it applies to the simple "core programming language" introduced previously. This PL is minimalist (so that e.g. all program variables have integer values, and all comparators are derived from 'less than (<)' using logical connectives) but it is sufficiently expressive to be representative.

Verification promotes an alternative perspective on the interpretation of an executing program, focused on the invariant relationships between variables in the program rather than the changing values. This approach was pioneered by Hoare and Dijkstra in the 1970s. A key definition:

A **Hoare triple** takes the form:

> **{precondition} command {postcondition}**

where a special syntax is used to bracket the three components of the triple.

The **precondition** and **postcondition** are expressed by predicate logic formulas. They refer respectively to the state *before* and *after* the program executes.

The **command** is expressed in terms of the core programming language constructs introduced above.

The informal interpretation of a Hoare triple is "if the command is executed in a state that satisfies the precondition then the state resulting from this execution satisfies the post-condition". (Note that this is vacuously true if the command is executed in a state that does not satisfy the precondition.)

We can use a Hoare triple to express the idea that the program `Fac1`, introduced previously:

```
y = 1;
z = 0;
while (z != x) {
        z = z+1;
        y = y*z;
}
```

is *intended* to compute the factorial x! of a natural number x. For this purpose, we can specify that (subject to verification!):

```
{x≥0} Fac1 {y=x!}
```

[Note the use of {}s where H&R use special bracket symbols.]

H&R motivate Hoare triples by pointing out that it is frequently necessary in specifying a program requirement to place conditions on the initial state. For instance, consider how:

> *Compute a number y whose square is less than the input x.*

might naturally be refined to:

> *If the input x is a positive number, compute a number y whose square is less than the input x.*

The specification of a program P to meet the latter requirement would then be:

```
{x > 0} P {y · y < x}
```

This specification of P does not prescribe it fully. The program that merely assigns `y=0` fulfils the specification, as does:

```
y = 0;
while (y * y < x) {
        y = y + 1;
}
y = y-1;
```

Validating a Hoare triple involves inferring that a postcondition ψ follows from a precondition φ upon executing the program P. Proving such a result in some respects resembles traditional logical deduction, but differs in that the precondition and postcondition relate to distinct states, and the reasoning activity must be applied to the program P. A different kind of proof calculus is then required. An important feature of the proofs to be developed in this calculus is that they are **compositional** in that the properties of a program P are inferred from the properties of the parts of P.

The proof of correctness of a program in general has to make use of standard logic (which is concerned with relationships between the values of program variables and external values associated with the machine execution) as well as of proof rules associated with state-changing actions specified in the program. This makes it a form of hybrid reasoning in which it may be necessary to refer to the values of *logical* variables in addition to the values of program variables in specifying the predicates about which we reason. Logical variables do not occur within the program - in particular, their values cannot be modified by the program execution,

To motivate logical variables, consider the following alternative program, called `Fac2`, that is also intended to compute the factorial x! of a natural number x:

```
y = 1;
while (x != 0) {
        y = y * x;
        x = x - 1;
}
```

This program consumes its input x - but we can't characterise its behaviour by

$\{x \geq 0\}$ `Fac2` $\{y=x!\}$

since x is 0 on termination. To overcome this problem, we must refer to original value of x (say x_0) prior to the execution of the program. The variable x_0 is then a logical variable that is interpreted as as being universally quantified in the precondition. The appropriate specification is expressed by a Hoare triple:

$\{x = x_0 \wedge x \geq 0\}$ `Fac2` $\{y=x_0!\}$

This is to be interpreted as:

> *For all* integers x_0, if x initially equals x_0 and is non-negative and we run the program `Fac2` such that it terminates, then the resulting state will satisfy y equals x_0!

Note that x_0 cannot be modified by `Fac2` as it does not occur as a program variable in `Fac2`.

Being compositional in nature, the proof calculus that is used to verify programs has a proof rule associated with each of the primitive programming constructs: composition, assignment, alternative (`if`) and iterative (`while`). The application of these proof rules is concerned with reasoning about state change. There is a complementary need to reason about the relationships between the values of program variables and logical variables, and to capture the semantics of the domain. For the simple core programming language discussed in this context, in which the only non-logical values of interest are integers, it is enough to assume that there is deductive apparatus in place to deal with predicate logic enlarged with the basic facts of arithmetic that are required to reason about arithmetic expressions. A single virtual proof rule, denoted by `Implied`, is used to indicate where deduction of this nature is used in verifying a program. The notation $\vdash_{AR}$ is used for sequents whose validity is established in this way. The `Implied` rule acts as an interface between predicate logic with arithmetic and program logic.

To capture the formal semantics of a Hoare triple, we must define what is being referred to informally as "the state of the machine" and "the external values associated with the machine execution". This is done by specifying a look-up function *l* which assigns an integer value to each (program and logical) variable x. Note that the

assignments to program variables that occur during execution will normally affect the values associated with variables in a dynamic fashion. For this reason, the look-up function also changes dynamically as a program executes. By convention, the look-up function that pertains prior to execution (and is used in interpreting the pre-condition) is denoted by *l*, and the look-up function that pertains at the end of the execution (and is used in interpreting the post-condition) is denoted by *l′*.

Inferences associated with executing programs are subject to encounter a problem that does not arise in a traditional deductive system - a program may or may not *terminate*. A Hoare triple can express useful information about a program that terminates, but if there is no termination, then there is no state in which to interpret the post-condition. This motivates two different notions of correctness that can be applied when interpreting a Hoare triple: **partial** correctness, and **total** correctness.

- **Partial correctness** asserts that if the preconditions are met, then the post-condition will be satisfied *provided that the program terminates*.

- **Total correctness** asserts that if the preconditions are met, then the post-condition will be satisfied and that *there is proof that the program will terminate*.

Note that in proving total correctness it is only necessary to strengthen the proof rules in respect of one of the program constructs, viz. the `while`-construct. This is the only construct that can potentially execute indefinitely. The symbols $\vdash_{par}$ and $\vdash_{tot}$ are used to denote specifications that express partial and total correctness respectively.

There are two different ways in which the proof of a program can be conceptualised:

- **As a proof tree:** The proof can be viewed as a tree, in which each node is annotated by a Hoare triple. The root of such a tree is annotated with the specification that was to be established. The edges emanating from each node point to child nodes of a node that are annotated with the antecedents of the proof rule that is being applied to establish the validity of the Hoare triple associated with that node.

- **As a proof tableau:** A program can be viewed as a sequence of primitive commands (assignments, `if`-statements or `while`-statements). If the program P is decomposed into

  $C_1; C_2; \ldots C_n;$

  in this way, we can express a proof of the correctness of P with respect to the specification

  $\{\varphi_0\}\ P\ \{\varphi_n\}$

  by interleaving formulae $\varphi_0, \varphi_1, ..., \varphi_n$ with the primitive commands to derive a tableau:

  $\{\varphi_0\}\ C_1; \{\varphi_1\}\ C_2; \{\varphi_2\}\ \ldots\ \{\varphi_{n-1}\}\ C_n; \{\varphi_n\}$

  in such a way that we can establish the validity of each of the Hoare triples

  $\{\varphi_i\}\ C_{i+1}\ \{\varphi_{i+1}\}.$

Proofs expressed by trees are too unwieldy to set down on paper for any but the simplest of programs. For this reason, a proof tableau representation is generally preferred.

Whether we express a proof via a tree or a tableau, human ingenuity and creativity is involved in its construction. In the case of a proof tableau, the challenge is to devise appropriate logical formulae $\varphi_0, \varphi_1, ..., \varphi_n$. Though it eventually makes sense to interpret a proof tableau by working through the program execution from the initial state to the final state (*presuming* termination in the case of a partial correctness proof), this is not the most appropriate way to construct such a tableau. For that purpose, the main heuristic is to work from the final state to the initial state, seeking to find suitable logical formulae φ_i for i = n-1, n-2, …, 1. This motivates the concept of "weakest precondition" (introduced by E W Dijkstra in the 1970s).

Weakest precondition [H&R, 276]: The process of obtaining φ_i from φ_{i+1} is called computing the weakest

precondition of C_{i+1} given the postcondition φ_{i+1}. That is to say, finding the logically weakest formula whose truth at the beginning of the execution of C_{i+1} is enough to guarantee φ_{i+1}.

To say that φ is logically weaker than ψ in this context means that $\vdash_{AR} \psi \rightarrow \varphi$. A weaker precondition is to be preferred because it puts fewer constraints on the preceding code.

Finding the weakest precondition of an assignment for a given postcondition is straightforward:

```
{ψ[E/x]} x=E {ψ}
```

Finding the weakest precondition of an `if`-statement for a given postcondition is likewise straightforward. To compute the weakest precondition φ to achieve the postcondition ψ for the statement:

```
if (B) {C1} else {C2}
```

we compute the weakest precondition φ_i of C_i given postcondition ψ for i=1 and 2, and set

$$\varphi = (B \rightarrow \varphi_1) \wedge (\neg B \rightarrow \varphi_2).$$

In reasoning about `while`-statements, an important role is played by *invariants*. An **invariant** of the body C of the `while`-statement `while B {C}` is a formula η with the property that: provided the boolean guard B is true, if η is true before C is executed, and C terminates, then it is also true after C has been executed.

It is easy to see that an invariant η of the `while`-statement `while B {C}` establishes the validity of the Hoare triple:

```
{η} while B {C} {η ∧ ¬B}
```

where the post-condition is the same as the pre-condition conjoined with ¬B. (This is merely to assert that the `Partial-while` proof rule is sound.) To establish the validity of a more general Hoare triple, such as:

```
{φ} while B {C} {ψ}
```

it is necessary to *find* an invariant η of the `while-` statement such that:

1. $\vdash_{AR} \varphi \rightarrow \eta$
2. $\vdash_{AR} \eta \wedge \neg B \rightarrow \psi$

are both valid.

The discovery of appropriate invariants is the aspect of program verification that demands human ingenuity (cf. the mechanical rules for determining weakest preconditions in respect of assignments and `if`-statements). A *useful* invariant expresses a relationship between the variables manipulated in the body of the `while`-statement which is preserved by the execution of the body, even though the values of the variables themselves must change [H&R, p284].

As a simple example, consider the `while`-statement in the program `Fac1`:

```
y = 1;
z = 0;
while (z != x) {
        z = z+1;
        y = y*z;
}
```

An invariant for this is given by $\eta \equiv y = z!$. The role that this plays in the proof of partial correctness of `Fac1` is evident from the fact that

$$\vdash_{AR} (y=1 \wedge z=0) \rightarrow \eta \text{ and } \vdash_{AR} \eta \wedge \neg(z \neq x) \rightarrow y=x!$$

are valid.

The above discussion supplies all the ingredients necessary to prove the partial correctness of the `Fac1` program, which is derived in H&R in two stages (see Example 4.17 on p.286).

To establish *total* correctness it is necessary to go further, and supply a proof that a program will terminate. For this purpose, a second proof rule associated with the `while`-statement (`Total-while`) has to be invoked (cf. H&R, p292). This involves embellishing the `Partial-while` proof rule by introducing an expression E whose value before and after each execution of the body of the `while`-statement is a non-negative integer, and whose value can be proved to decrease on each execution. E is called a **variant** of the `while-` statement. It is easy to verify that the expression

$$E \equiv x\text{-}z$$

is a variant for the `while`-statement in `Fac1`, thus establishing total correctness for the program.