Specifying Algorithms

We now continue with Part A of the textbook. Material in this document is from chapter 1 and sections 2.1 - 2.7.

Stages in software construction include requirements, specification and implementation. Verification consists of checking that the program follows the specification.

- Specifications describe transformations from input values to output values.
- A program transforms input values to output values in a particular way.

A specification consists of the following parts:

1. What the input will be.
2. What the outputs should be.
3. Environment in which the specification/program should work.

Inputs and outputs typically refer to things that can be observed:

- set of input variables
- set of output variables
- constraints on the values that the variables can take

Formalization of a specification consists of the following:

- declarative interface: static properties of the identifiers
- pre-conditions: assertion on input values that the program will be given

- post-condition: assertion on output values, possibly in relation to the same input values

The declarative interface can specify which values should not be changed by the code (these are declared as constants).

We can say that a specification is a contract: the software designer agrees to establish the post-condition if the program is started in a way that satisfies the pre-condition.

If the program is run in a context not covered by the pre-condition, it can run in any way without "breaking" the contract.

Note: In the following, we will develop logic-based techniques to verify correctness of small programs (algorithms), that is, we want to prove that a program does what a specification says it should do.

The techniques discussed here are, in general, too time consuming to be directly used with large software systems. Formal methods[1] used for verification encompass various software tools for reasoning about correctness. Such tools implement algorithms, typically, from theorem proving or model checking. The algorithms and their use require an understanding of foundations of program correctness discussed in Chapters 1–4 of the textbook.

We use logical formulas, called assertions, as comments in programs. We are asserting that the formula should be true when flow of control reaches it. The assertions are allowed to contain notation that cannot be used in Boolean expressions in programs, see Section 1.3 in the text. In particular, the assertions may include quantified formulas. It will be important to recall the notions of *free variables* and *bound variables* in quantified formulas.[2]

The assertions are not evaluated during program execution, but would be true if evaluated. The notation

[1] You will learn more about formal methods in 4th year.

[2] A very brief review will be done in class. If you do not recall the notions, please review material from CISC-204.

```
ASSERT(P)
S
ASSERT(Q)
```

where S is a sequence of program statements, has the following interpretation:

- If execution of S is begun in a state satisfying P, then it is guaranteed to terminate (in a finite amount of time) in a state satisfying Q.

The above condition is called total correctness.

The correctness statement

```
ASSERT(P)
S
ASSERT(Q)
```

is said to be partially correct if always when execution of S is begun in a state satisfying P it does not terminate (normally) in a state not satifying Q. Note that with partial correctness we do not exclude the possibility of nontermination, runtime errors, etc.

Consider assertions `P` and `Q`. If `P` implies `Q`, we say that "`P` is stronger than `Q`" or "`Q` is weaker than `P`". When the terms "stronger" and "weaker" have the above meaning, note that any assertion `P` is stronger (or weaker) than itself. A general rule is that the pre-condition may be strengthened and the post-condition may be weakened:

(`P {S} Q` and `[P' => P]`) implies `P' {S} Q`

(`P {S} Q` and `[Q => Q']`) implies `P {S} Q'`

Above `P {S} Q` is an abbreviation for the correctness statement

```
ASSERT(P)
S
ASSERT(Q)
```

Example. To illustrate the notion of correctness statements and their validity we consider the specification for array-searching code and some programs for array searching as presented in textbook Section 2.1.

We begin verification from simplest possible program fragments by considering simple assignment statements. The first idea could be, perhaps, to reason in the forward direction but this turns out to be inconvenient.

Axiom scheme:

```
V==I { V = E; } V == [E](V ↦ I)
```

Here `[E](V ↦ I)` is the expression obtained from E by replacing occurrences of V by I.

Required side condition above: identifier I must be distinct from variable V. Why is this needed?

Example.

```
x==7 { x = 6*x + 15; } x==57
```

The use of the above scheme is restricted because the precondition has to be an equality test for the left side of an assignment.

A simpler and more general approach does the reasoning backwards. Based on the post-condition determine the most general pre-condition that guarantees the post-condition holds after the assignment.

Hoare's axiom scheme for assignments:

```
[Q]( V ↦ E ) { V = E; } Q
```

See section 2.3 for a detailed explanation of the symbols. For post-condition Q, Hoare's axiom scheme gives the most general pre-condition that guarantees that post-condition holds after assignment.

Note that in Hoare's axiom scheme the post-condition can be any assertion.

Example.

```
x-y >= 0 { x = x-y; } x >= 0
```

Example. Verify, using also pre-condition strengthening,

```
ASSERT( y === 3 )
x = y-1;
ASSERT( x >= 2 )
```

Issues concerning substitutions

When substituting an expression E for an identifier I we should take care of the following:

1. Add parentheses when necessary.
2. Only free occurrences of I in the assertion are to be replaced by E.
3. As a result of the substitution free occurrences of an identifier in E should not become bound. To prevent this, the names of bound identifiers in the assertion can be changed, when necessary. (The names of free variables cannot be changed.)

Example. What is the result of the below substitution?

```
[ ForAll(x) Exists(y) x < z implies x < y < z ]( z ↦ x + 1 )
```

Note that bound occurrences of x should be replaced by some "fresh" identifier.

The inference rule for *statement sequencing* is:

$$\frac{\mathtt{P\{C_0\}Q} \quad \mathtt{Q\{C_1\}R}}{\mathtt{P\{C_0C_1\}R}}$$

A limitation is that the post-condition of the first correctness statement has to be the same as the pre-condition of the second correctness statement.

Using the fact that a pre-condition may be strengthened (or post-condition weakened), we see that also the following more general inference rule is valid:

$$\frac{\texttt{P}\{\texttt{C}_0\}\texttt{Q} \quad \texttt{Q}'\{\texttt{C}_1\}\texttt{R} \quad \texttt{Q implies Q}'}{\texttt{P}\{\texttt{C}_0\texttt{C}_1\}\texttt{R}}$$

Proof tableaux

A formal proof with respect to a set of axioms and inference rules consists of a sequence of statements where each statement is an axiom or a conclusion of one of the inference rules all of whose premises have already been proved.

In program verification we use *proof tableaux* as short hand notation for formal proofs of correctness. A proof tableau consists of program code, pre- and post-conditions and *intermediate assertions.* Axioms are instances of the Hoare axiom scheme. The correctness of the intermediate assertions is implied by pre-condition strengthening (post-condition weakening), the inference rule for statement sequencing, and inference rules for conditional statements and loops (to be discussed in the following). Since Hoare axiom scheme works "backwards", a proof tableau is typically constructed starting from the bottom.

Example. (Exercise 2.26, p. 46) Construct proof tableau for

```
ASSERT( x == x0 && y == y0 )
x = x-y;
y = y+x;
x = y-x;
ASSERT( x == y0 && y == x0 )
```

Example. Construct proof tableau for

```
ASSERT( x >= 2 || x <= -2 )

y = 2*x;

z = x-y;

y = x+z;

ASSERT( x+y >= 1 || z >= 2 )
```