Specifying Algorithms

We now continue with Part A of the textbook.

- 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.

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.

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

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

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

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

- 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 pre- and post-conditions

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

are 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.

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

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

Axiom scheme for assignment statements:

`V==I { V = E; } V == [E](V ↦ I)`

See section 2.3 for explanation of notations.

Required side condition above: I must be distinct from 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 is an equality test for the left side of an assignment.

Hoare's scheme for assignments:

```
[Q]( V ↦ E ) { V = E; } Q
```

See section 2.3 for an explanation of the symbols.

Example.

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

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.

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:

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

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:

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

The inference rule for if-statements is as follows:

$$\frac{\texttt{P\&\&B}\{\texttt{C}_0\}\texttt{Q} \quad \texttt{P\&\&!B}\{\texttt{C}_1\}\texttt{Q}}{\texttt{P}\{\text{if}(\texttt{B})\texttt{C}_0 \text{ else } \texttt{C}_1\}\texttt{Q}}$$

Example. Prove the following correctness statement:

```
ASSERT( y==y0 && z==z0)
if (y <= z) { y=z+1; z=z+2; } else { z=y+2; y=y+1;}
ASSERT( max( y0, z0) < y < z )
```