Chapter 1

Proving Sequential Programs

Assume we are working in a PVS THEORY that defines the signature of some data type, \( \mathcal{A} \) containing a type of values, \( \mathcal{A} \).

1.1 Syntax

A \( \langle \text{TERM} \rangle \) is an expression involving “simple” operators in \( \mathcal{A} \). For simplicity, assume these are non-recursive, first-order functions on constants and variables. However, we shall allow array and record references, so long as the indices do not involve arrays or records.

A \( \langle \text{QFF} \rangle \), or \( \text{test} \) is a quantifier-free predicate. That is, a comparison of some kind that does not imply looping or searching.

IVS, is a set of program variable symbols.

The sequential language \( \langle \text{STMT} \rangle \) of statements over \( \mathcal{A} \) is defined inductively as follows:

\[
\langle \text{STMT} \rangle ::= \langle \text{IVS} \rangle := \langle \text{TERM} \rangle \quad (\text{assignment})
\]

\[
\text{begin} \langle \text{STMT} \rangle ; \langle \text{STMT} \rangle \text{ end} \quad (\text{compound})
\]

\[
\text{if} \langle \text{QFF} \rangle \text{ then } \langle \text{STMT} \rangle \text{ else } \langle \text{STMT} \rangle \quad (\text{conditional})
\]

\[
\text{while} \langle \text{QFF} \rangle \text{ do } \langle \text{STMT} \rangle \quad (\text{repetition})
\]

A program correctness assertion is a predicate saying what a program does. It has the form

\[
\{ \langle \text{PreCondition} \rangle \} \ S \ \{ \langle \text{PostCondition} \rangle \}
\]

in which PreCondition and PostCondition are logical predicates (possibly quantified), and \( S \in \langle \text{STMT} \rangle \). The assertion says, “If PreCondition holds before \( S \) executes, then PostCondition holds after \( S \) executes.”
1.2 Semantics

If is any mapping from program variables to values, $ENV : [IVS \rightarrow A]$, let
$T : ENV \times \langle \text{TERM} \rangle \rightarrow A$ give the value of terms, and
$I : ENV \times \langle \text{QFF} \rangle \rightarrow \{T, F\}$ give the value of a quantifier-free formula.

The operational interpretation of $\langle \text{STMT} \rangle$ is defined:

$$\mathcal{M} : ENV \times \text{STMT} \rightarrow ENV$$

(a) $\mathcal{M} \sigma [v := t] = \sigma \setminus \{(v, \sigma(v))\} \cup \{(v, T \sigma[t])\}$

(b) $\mathcal{M} \sigma [\text{begin } S_1 ; S_2 \text{ end}] = \mathcal{M} \sigma'[S_2]$ where $\sigma' = \mathcal{M} \sigma [S_1]$

(c) $\mathcal{M} \sigma [\text{if } Q \text{ then } S_1 \text{ else } S_2] = \begin{cases} \mathcal{M} \sigma [S_1] & \text{if } I \sigma[Q] = \text{true} \\ \mathcal{M} \sigma [S_2] & \text{if } I \sigma[Q] = \text{false} \end{cases}$

(d) $\mathcal{M} \sigma [\text{while } Q \text{ do } S] = \begin{cases} \sigma, & \text{if } I \sigma[Q] = \text{false} \\ \mathcal{M} \sigma'[\text{while } Q \text{ do } S] & \text{if } I \sigma[Q] = \text{true} \end{cases}$

where $\sigma' = \mathcal{M} \sigma [S]$,

$\mathcal{M}$ is defined according to its effect on the program state, $\sigma$, so we can think of a program $S$ as “executing”.

(a) The interpretation of assignment says to take the function $\sigma \in ENV$, and replace the binding for program variable $v$ with one that binds $v$ to the value of $t$.

(b) The compound-statement interpretations says, execute statement $S_1$, and then execute $S_2$ using this resulting memory, $\sigma'$.

(c) The rule for if-then-else says execute just one of the branches according to the outcome of the test.

(d) The rule for while-statements, unlike all the other rules, does not reduce interpretation to some proper substatement. We can infer from this that execution of certain programs does not terminate.

We will not use this operational interpretation very much, but now that it has been defined, let us prove an interesting fact about compound statements.

**Proposition 1.1** The compound operator, ‘; ’ is associative in the sense that for all environments $\sigma$ and statements $S_1, S_2$, and $S_3$,

$$\mathcal{M} \sigma [\text{begin } S_1 ; \text{begin } S_2 ; S_3 \text{ end end}] = \mathcal{M} \sigma [\text{begin begin } S_1 ; S_2 \text{ end ; } S_3 \text{ end}]$$
1.2. SEMANTICS

PROOF: Apply Definition ??.

Thus, it is not ambiguous to write

\begin{verbatim}
begin S_1; S_2; S_3 end
\end{verbatim}

because it doesn’t matter how \texttt{begin-ends} are associated.

\[
\mathcal{I}_\sigma[P] \text{ implies } \mathcal{I}_\sigma'[Q] \text{ where } \sigma' = \mathcal{M}_\sigma[S]
\]

We are now in a position to define the meaning of a partial correctness assertion. Let \( \mathcal{F}: ENV \times (FOF) \rightarrow \{T, F\} \) give the truth-value of a first-order predicate (First-Order Formula):

\[
\begin{array}{c}
\{P\} S \{Q\} \\
\text{iff for all } \sigma \in ENV, \\
\mathcal{F}_\sigma[P] \text{ implies } \mathcal{F}_\sigma'[Q] \text{ where } \sigma' = \mathcal{M}_\sigma[S]
\end{array}
\]

Notice that by to this definition, \( \{P\} S \{Q\} \) is vacuously true if \( S \) fails to terminate, because \( \sigma' \) does not exist.

1.2.1 Reasoning Rules for Statements

Here is a fundamentally important result, called the \textit{Law of Assignment}.

\textbf{Proposition 1.2} For all environments \( \sigma \), the partial correctness assertion \( \{P\} \ v := t \ {\{Q\}} \)
holds iff

\[
P \Rightarrow Q^{t}_v
\]

where \( Q^{t}_v \) denotes substituting term \( t \) for variable \( v \) in formula \( Q \).

\textbf{PROOF:} Omitted, but straightforward.

We will express facts like this as \textit{inference rules}

\[
\begin{array}{c}
P \Rightarrow Q^t_v \\
\{P\} v := t \ {\{Q\}} \\
\end{array}
\]

Assignment Rule

In general, there may be more than one statement above the line. What lies below the line represents a goal that we are trying to prove. What lies above the line tell us what fact, or set of facts in general, that we must show in order to establish the goal.
Example

Ex 1.1 Prove: \{z + xy - y = AB\} x := x-1 \{z + xy = AB\}

Performing the substitution we get

\[ z + zy - y = AB \Rightarrow z + (x - 1)y = AB \]

But this is clearly true, since \( z + (x - 1)y = z + xy - y \).

There is a reasoning rule for each phrase-type of the language of statements.
These are shown in Figure 1.2.5 and discussed individually below. The assignment rule has already been discussed.

1.2.2 The Compound Rule

The rule

\[
\begin{array}{c}
\{P\} \quad S_1 \quad \{Q\} \quad S_2 \quad \{R\} \\
\text{COMPOUND RULE}
\end{array}
\]

says that you can break down the proof of a compound statement by finding an intermediate assertion to place between them. As we shall see in the next section it is never a problem to determine what this assertion should be.

Example

Ex 1.2 Prove: \{z + xy = A \land y > 0\}

\[
\text{begin } x := x - 1 ; \quad z := z + y \text{ end} \quad \{z + xy = A\}
\]

Let intermediate assertion \( Q \equiv (z + y) + xy = A \). You may be able to see shy this \( Q \) was chosen, but if not, it will become clear in a moment. The compound rule says that to prove the desired program correctness assertion, it suffices to prove the following two assertions:

\[
\{z + xy = A \land y > 0\} \quad x := x - 1 \quad \{Q\}
\]

and

\[
\{Q\} \quad z := z + y \quad \{z + xy = A\}
\]

The second of these is trivial. By the assignment rule, it reduces to

\[
Q \Rightarrow (z + xy = A)^{z+y}
\]

After performing the substitution we get the tautology

\[
(z + y) + xy = A \quad \Rightarrow \quad (z + y) + xy = A
\]

To prove the first correctness assertion, we again use the assignment rule. We are to assume that \( z + xy = A \) and \( y > 0 \) and prove \( Q_z^{-1} \). After performing the substitution we can derive

\[
(z + y) + (x - 1)y = z + y + xy - y = z + zy = A
\]

by our assumption of the precondition.
1.3. USING THE RULES

1.2.3 The Conditional Rule

The rule

\[ \begin{array}{c}
\{P \land B\} \quad S_1 \quad \{Q\} \\
\{P \land \neg B\} \quad S_2 \quad \{Q\}
\end{array} \]

\[ \{P\} \text{ if } B \text{ then } S_1 \text{ else } S_2 \quad \{Q\} \]

**CONDITIONAL RULE**

says that you can argue separately about the two branches, while using the fact that the test \( B \) succeeded, in the case of the then-part, or failed, in the case of the else part.

1.2.4 The Repetition Rule

The rule

\[ \begin{array}{c}
\{P \land B\} \quad S \quad \{I\}
\end{array} \]

\[ \{I\} \text{ while } B \text{ do } S \quad \{I \land \neg B\} \]

**REPETITION RULE**

Is a formal statement of Theorem ?? from Chapter ??.

1.2.5 The Relaxation Rule

The specific assertions within programs do not always match exactly with the subgoals generated by the rules. The relaxation rule says that once we have proven a program correctness assertion, we can replace the precondition by a stronger statement and we can replace the postcondition by a weaker statement:

\[ \begin{array}{c}
P' \Rightarrow P \\
\{P\} \quad S \quad \{Q\} \\
Q \Rightarrow Q'
\end{array} \]

\[ \{P'\} \quad S \quad \{Q'\} \]

**RELAXATION RULE**

An instance of this rule is the invariant rule for while statements:

\[ \begin{array}{c}
P \Rightarrow I \\
\{I \land B\} \quad S \quad \{I\} \\
(I \land \neg B) \Rightarrow Q
\end{array} \]

\[ \{P\} \text{ while } B \text{ do } S \quad \{Q\} \]

**INVARIANT RULE**

1.3 Using the Rules

Given an assertion \( \{P\} \quad S \quad \{Q\} \), how do we go about determining whether it is true? The only thing to do is look for a rule that applies. Since there is just one rule for each kind of phrase in the language, there always just one choice. Since each inference rule reduces an argument about a statement to subarguments about its parts, we will eventually reduce the program correctness assertion to a collection of purely logical formulas. These are called the program’s verification conditions. If the verification conditions are all true, so is the original correctness assertion. In other words, the program is correct.
Unfortunately, knowing which rule to apply is not always enough. Consider the assertion

\[ \{x = 2\} \begin{array}{l} S_1 \; ; \; S_2 \end{array} \{y = 4\} \]

Obviously, we need to use the compound rule, but in order to use it, we must come up with an intermediate assertion \( Q \) such that

\[
\begin{array}{c}
\{x = 2\} \begin{array}{l} S_1 \; {\{Q\}} \; S_2 \; {\{x = 4\}} \\
\{x = 2\} \begin{array}{l} \text{begin} \; S_1 \; ; \; S_2 \; \text{end} \; {\{y = 4\}} \\
\end{array}
\end{array}
\]

We cannot choose \( Q \) at random, but we can almost always find the right formula. Just two of the four rules, repetition and compound, require us to invent an assertion in this way.

For while statements, we must find an invariant formula. In general, this is hard to do all but one person: the person who wrote the loop. In fact, a loop’s invariant reveals the essence of what the loop is doing. Let us, therefore, require the program to contain an invariant for every loop. This is not unreasonable, since, as we have just claimed, with practice you can write down invariant the instant you conceive of the loop. Furthermore, there are often ways to derive the invariant from the surrounding program specification.

From now on, a while statement must have the form

\[ \text{while } B \text{ \{inv:} I \text{\} do } S \]

and our reasoning rule becomes

\[
\begin{array}{c}
P \Rightarrow I \quad (I \land B) \quad S \quad (I \land \neg B) \Rightarrow Q \\
P \text{ while } B \text{ \{inv:} I \text{\} do } S \quad \{Q\}
\end{array}
\]

The situation for compound statements isn’t nearly so bad. For any compound statement we can determine the intermediate assertion by analyzing the component statements.

**Technique 1.3** To prove \( \{P\} \begin{array}{l} \text{begin} \; S_1 \; ; \; v := t \end{array} \{R\} \), choose \( Q \) to be \( R_v^t \).

If we do this we get the following proof structure:

\[
\begin{array}{c}
\{P\} \begin{array}{l} S_1 \; \{R_v^t\} \end{array} \quad \begin{array}{c}
R_v^t \Rightarrow R_v^t \\
\{R_v^t\} \quad v := t \quad \{R\}
\end{array}
\end{array}
\]

Thus, the right-hand verification condition reduces to a trivial tautology.

**Technique 1.4** To prove \( \{P\} \begin{array}{l} \text{begin} \; v := t \; ; \; S_2 \end{array} \{R\} \) when \( P \) and \( t \) do not contain the variable \( v \), choose \( Q \) to be \( P \land (v = t) \).
1.3. USING THE RULES

In this case the choice of \( Q \) trivializes the left-hand verification condition to \( P \Rightarrow (P \land t = t) \):

\[
\begin{array}{c}
P \Rightarrow (P \land v = t) \\
\{P\} \ v := t \ \{P \land v = t\} \\
\{P\} \begin{array}{c}
S_1 \\
\{R\}
\end{array}
\end{array}
\]

If the precondition \( P \) or term \( t \) do contain the program variable \( v \), this tactic fails miserably. For example, consider the assertion

\[
\{x = 2\} \begin{array}{c}
\{x := 3\} \\
S_2 \end{array} \ \{R\}
\]

The intermediate assertion would be \((x = 2) \land (x = 3)\), which is contradictory.

**Technique 1.5** To prove

\[
\{P\} \begin{array}{c}
S_1 \\
while \ B \ \{inv:\ I\} \ do \\
S_2 \ \{R\}
\end{array}
\]

choose intermediate assertion \( Q \) to be \( I \).

See if you can develop a technique for dealing with conditional statements.

**Exercises 1.3**

1. \( \{x = A \land y = B \land A \geq B\} \)
   while \( y \neq 0 \) do \( \{(x - y = A - B) \land (x \geq y)\} \)
   begin
   \( x := x - 1; \)
   \( y := y - 1 \)
   end
   \( \{x = A - B\} \)

2. \( \{x = A \land y = B\} \)
   begin
   \( z := 1; \)
   while \( y \neq 0 \) do \( \{z \cdot x^y = A^B\} \)
   begin
   \( z := z \cdot x; \)
   \( y := y - 1 \)
   end
   end
   \( \{z = x^y\} \)
CHAPTER 1. PROVING SEQUENTIAL PROGRAMS

\[
\begin{align*}
&\text{Assignment Rule} \\
&P \Rightarrow Q^t_v \\
&\{P\} \; v := t \; \{Q\}
\end{align*}
\]

\[
\begin{align*}
&\text{Compound Rule} \\
&\{P\} \; S_1 \; \{Q\} \; \{Q\} \; S_2 \; \{R\} \\
&\{P\} \; \text{begin} \; S_1 \; \text{; } S_2 \; \text{end} \; \{Q\}
\end{align*}
\]

\[
\begin{align*}
&\text{Conditional Rule} \\
&\{P \land B\} \; S_1 \; \{Q\} \; \{P \land \neg B\} \; S_2 \; \{Q\} \\
&\{P\} \; \text{if } B \; \text{then} \; S_1 \; \text{else} \; S_2 \; \{Q\}
\end{align*}
\]

\[
\begin{align*}
&\text{Repetition Rule} \\
&\{P \land B\} \; S \; \{I\} \\
&\{I\} \; \text{while } B \; \text{do } S \; \{I \land \neg B\}
\end{align*}
\]

Figure 1.1: Basic reasoning rules for the language of statements

\[
\begin{align*}
&\text{While Rule} \\
&P \Rightarrow I \; \{I \land B\} \; S \; \{I\} \; (I \land \neg B) \Rightarrow Q \\
&\{P\} \; \text{while } B \; \{\text{inv:} I\} \; \text{do } S \; \{Q\}
\end{align*}
\]

\[
\begin{align*}
&\text{Assignment-right} \\
&\{P\} \; S \; \{Q^t\} \\
&\{P\} \; \text{begin} \; S \; ; \; v := t \; \text{end} \; \{Q\}
\end{align*}
\]

\[
\begin{align*}
&\text{Assignment-left} \\
&\{P \land (v = t)\} \; S \; \{Q\} \\
&\{P\} \; \text{begin} \; v := t \; ; \; S \; \text{end} \; \{Q\}
\end{align*}
\]

\[
\begin{align*}
&\text{Initialization Rule} \\
&P \Rightarrow S_1 \; \{I\} \; \{I\} \; \text{while } B \; \{\text{inv:} I\} \; \text{do } S_2 \; \{Q\} \\
&\{P\} \; \text{begin} \; S_1 \; \text{; while } B \; \{\text{inv:} I\} \; \text{do } S_2 \; \{Q\}
\end{align*}
\]

Figure 1.2: Derived reasoning rules for the language of statements
1.3. USING THE RULES

3. \( \{x = A \land y = B\} \)
   
   begin
   \( q := 0 \);
   \( r := x \);
   while \( r \geq y \) do \( \{q \cdot y + r = A\} \)
   
   begin
   \( q := q + 1 \);
   \( r := r - y \)
   end
   
   \( \{(q \cdot y + r = A) \land (r < y)\} \)

4. \( \{x = A\} \)

   begin
   \( z := 1 \);
   while \( x \neq 1 \) do \( \{z \cdot x! = A!\} \)
   
   begin
   \( z := z \cdot x \);
   \( x := x - 1 \)
   end
   
   \( \{z = A!\} [z \text{ is } "A \text{ factorial.}"\]  

5. Assume \( \text{even?} \) is a primitive test for even numbers.

\[
\{x = A \land y = B\}
\]

begin
\( z := 1 \);
while \( y \neq 0 \) do \( \{z \cdot x^y = A^B\} \)

if \( \text{even?}(y) \)
   then begin \( y := y/2; \ x := x \cdot x \) end
else begin \( y := y - 1; \ z := z \cdot x \) end

\( \{z = A^B\} \)

Copyright © 2006 Steven D. Johnson        DRAFT MATERIAL         February 27, 2006
6. \(\{x = A \land y = B\}\)
   
   \[
   \begin{align*}
   & \text{begin} \\
   & \quad z := 1; \\
   & \quad \text{while } y \neq 0 \text{ do } \{z \cdot x^y = A^B\} \\
   & \quad \quad \text{begin} \\
   & \quad \quad \quad \text{while } \text{even}(y) \text{ do } \{z \cdot x^y = A^B \land y \neq 0\} \\
   & \quad \quad \quad \quad \text{begin} \\
   & \quad \quad \quad \quad \quad y := y/2 \\
   & \quad \quad \quad \quad \quad x := x \cdot x \\
   & \quad \quad \quad \quad \quad \text{end}; \\
   & \quad \quad \quad z := z \cdot x; \\
   & \quad \quad \quad y := y - 1 \\
   & \quad \quad \quad \text{end} \\
   & \quad \text{end} \\
   & \{z = A^B\}
   \end{align*}
   \]

7. \textbf{NOTE}: \(GCD(x, y)\) stands for the greatest common divisor of \(x\) and \(y\).

\[
\begin{align*}
\{x = A \land y = B\} \\
\text{begin} \\
\text{while } x \neq y \text{ do} \\
\quad \{GCD(x, y) = GCD(A, B)\} \\
\quad \text{if } x < y \\
\quad \quad \text{then } y := y - x \\
\quad \quad \text{else } x := x - y \\
\text{end} \\
\{x = GCD(A, B)\}
\end{align*}
\]