This is an informal presentation of Chapter 10 in Induction, Recursion and Programming*, omitting the underlying mathematical details.*

Proving Programs

The logical calculus used in PVS is extended with inference rules for reasoning about sequential programs. The introduction below presents these rules and shows how a statement of the form *"Program S is correct,"* can be reduced to pure logical propositions, or *verification conditions* that can be verified in a proof validation system like PVS.

This section begins by introducing a notation for making true-false assertions about programs. The next section gives inference rules

A Simple Algorithmic Language

The *Language of Statements*, STMT, is a simple, sequential programming language, similar in form to many languages that exist today, such as C and Java. There are just four kinds of statements.

1. The *assignment statement*, has the form

   $$v \texttt{ := } \textsc{Term}$$

   The object to the left of the assignment symbol is called an *identifier*, or sometimes *program variable* (but never just *"variable"*). To the right is an expression, TERM, whose value is calculated and then associated with the program variable from that point on, or until another value is assigned to the same identifier. Identifiers may be simple names, such as `x` and `answer`, or array references, such as `a[i]` and `b[5, j]`.

2. A *conditional statement* has the form

   $$\texttt{if } \textsc{Test} \texttt{ then } S_1 \texttt{ else } S_2$$

   Where S_1 and S_2 are, themselves, statements. If the TEST holds then statement S_1 executes; otherwise, statement S_2 is executes. Tests are unquantified propositions and do not have side effects.

3. A *repetition statement* has the form

   $$\texttt{while } \textsc{Test} \texttt{ do } S$$

   Statement S is repeatedly executed so long as the TEST remains true.

4. Finally, a *compound statement* has the form

   $$\texttt{begin}\, S_1 \texttt{ ; } S_2 \texttt{ ; } \ \ldots\ \texttt{ ; } S_n \texttt{ end}$$

   Statements S_1, S_2, ..., S_n are executed in order, from left to right.

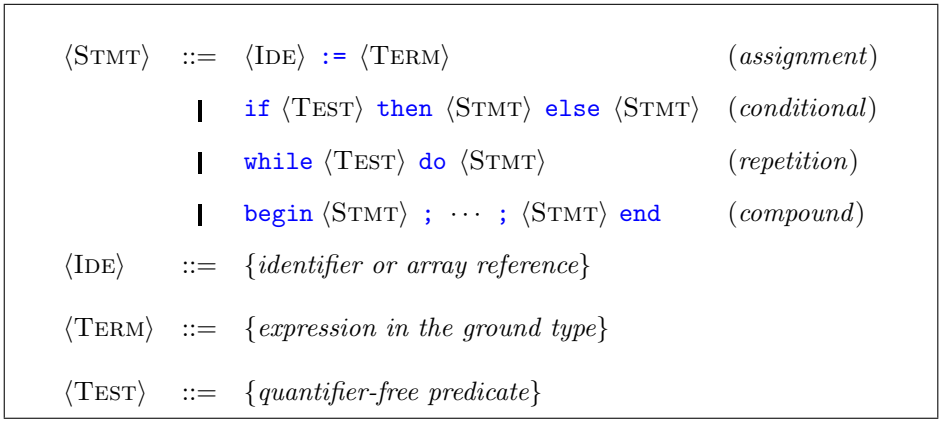

Figure 1: Partial description of the Stmt programming language.

Figure 1 shows a concise specification of the Stmt language using *Backus-Naur* notation ("BNF"). The BNF grammar says nothing about what statements *mean*, only what they look like.

There are no input/output operations in Stmt. We can talk about the result of a program in terms of its intial and final content. Memory is viewed abstractly as a function from program identifiers to values of the appropriate type. In a more complete memory model, identifiers are associated with *addresses*, addresses with memory *content*, and content with representations of *values*; but we don't need all that detail for our purposes. Figure 2 is an example of a program in Stmt. Stmt actually describes a family of languages as determined by the "data" over which they operate, in this case, natural numbers.

The program labels, $\mathcal{P}$, ℓ_1 and ℓ_2 are not part of the language—there is no `goto` construct, so labels aren't needed, but they are used to refer to points of the program.

Comments written between braces, $\{ \cdots \}$, are called *assertions*. For now, comments are optional, but they become a formal part of the language syntax, used to reason logically about a program's data state.

Program Correctness Assertions

Informal Definition. Let P and Q be predicate formulas over program states and let S be a program fragment. The *partial correctness assertion*

$$\{\ P\ \}\ S\ \{\ Q\ \}$$

Is a predicate that says, *"If P holds initially, then just after S executes, Q holds."* We call formula P a *precondition* of statement S; and we call formula Q a *postcondition*.

```
P:   { A, B ∈ ℕ }
     begin
     x := A ;
     y := B ;
     z := 0 ;
     { z + xy = AB }
ℓ1:  while x ≠ 0 do
       if even?(x)
       then
         begin x := ½x ; y := 2y end
       else
ℓ2:      begin x := x − 1 ; z := z + y end
     end
     { z = AB }
```

Figure 2: A program in the programming language STMT

Implicit in this definition is the assumption that S terminates. if S contains an infinite loop and does not terminate, the assumption is false, making $\{\ P\ \}\ S\ \{\ Q\ \}$ vaccuously true. In Figure 2 three assertions are introduced.

1. The precondition is $\{\ A, B \in \mathbb{N}\ \}$;
2. the postcondition is $\{\ z = AB\ \}$;
3. and there is an *intermediate assertion* $\{\ z + xy = AB\ \}$.

These can be read "Program $\mathcal{P}$ computes the product of natural numbers A and B."

Reasoning Rules for Statements

There is a primitive inference rule for each kind of STMT:

Assignment Rule

$$\frac{P \ \Rightarrow\ Q\left[\begin{smallmatrix} t \\ v \end{smallmatrix}\right]}{\{\ P\ \}\ v\ \texttt{:=}\ t\ \{\ Q\ \}}$$

$Q\left[\begin{smallmatrix} t \\ v \end{smallmatrix}\right]$ denotes the formula obtained by substituting term t for all free occurences of variable v in formula Q.

EXAMPLE 1. *Prove* $\{\ z + (x+1)y = A\ \}\ z\ \texttt{:=}\ z + y\ \{\ z + xy = A\ \}$.
By the Assignment Rule, it suffices to prove

$$(z + (x+1)y = A)\ \textit{implies}\ \big(z + xy = A\big)\left[\begin{smallmatrix} z+y \\ z \end{smallmatrix}\right]$$

Distributing on the left, and substituting on the right, we get

$$(z + y + xy = A) \textit{ implies } (z + y + xy = A)$$

which is trivially true.

Compound Rule

$$\frac{\{\, P \,\}\ S_1\ \{\, Q \,\} \quad \diamond \quad \{\, Q \,\}\ S_2\ \{\, R \,\}}{\{\, P \,\}\ \texttt{begin}\ S_1\ \texttt{;}\ S_2\ \texttt{end}\ \{\, R \,\}}$$

This rule says that in order to prove an assertion about two sequential statements, it suffices to find an *intermediate assertion*, Q that serves *both* as a postcondition for the first statement *and* as a precondition for the second.

Of course, the problem with this rule is that it seems to require the prover to guess what the intermediate assertion is. We shall see later that this is not the case!

EXAMPLE 2. *Prove* $\{\, z + xy = A \wedge x > 0 \,\}$
$\texttt{begin}\ x\ \texttt{:=}\ x - 1\ \texttt{;}\ z\ \texttt{:=}\ z + y\ \texttt{end}$
$\{\, z + xy = A \,\}$

Let intermediate assertion Q be $z + (x + 1)y = A$. By the Compound Rule, is suffices to prove

$$\{\, z + xy = A \wedge y > 0 \,\}\ x\ \texttt{:=}\ x - 1\ \{\, z + (x + 1)y = A \,\}$$
and
$$\{\, z + (x + 1)y = A \,\}\ z\ \texttt{:=}\ z + y\ \{\, z + xy = A \,\}$$

The second PCA proved in Example 1. As for the first, by the Assignment Rule, the PCA on the right is true if

$$\big(z + (x + 1)y = A \wedge y > 0\big)\ \textit{implies}\ \big(z + x\,y = A\big)\big[{}^{x-1}_{x}\big]$$

Substituting on the right we get

$$\big(z + xy = A \wedge y > 0\big)\ \textit{implies}\ \big(z + \big[\,(x - 1) + 1\big]\,y = A\big)$$

The 1s cancel on the right making the proposition true. The precondition $x > 0$ is not explicity involved in the logic, but it is necessary to assure that $x - 1$ is a natural number.

Conditional Rule

$$\frac{\{\, P \wedge B \,\}\ S_1\ \{\, Q \,\} \quad \diamond \quad \{\, P \wedge \neg B \,\}\ S_2\ \{\, Q \,\}}{\{\, P \,\}\ \texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \{\, Q \,\}}$$

The rule for `if` says simply to add B to the precondition for the `then` branch, and $\neg B$ to the precondition for the `else` branch, and evaluate them seperately. Within the `then` branch you know that the test B has succeeded, and for the `else` branch you know it has failed.

medskipExample 3. *Prove* $\{\,(x \neq 0) \wedge (z + xy = A)\,\}$
`if` *even*?(x)
`then`
`begin` x `:=` $\frac{1}{2}x$ `;` y `:=` $2y$ `end`
`else`
`begin` x `:=` $x - 1$ `;` z `:=` $z + y$ `end`
$\{\,z + xy = A\,\}$

By the Conditional Rule, it suffices to prove

(a) $\{\,(x \neq 0) \wedge (z + xy = A) \wedge \mathit{even}?(x)\,\}$
`begin` x `:=` $\frac{1}{2}x$ `;` y `:=` $2y$ `end`
$\{\,z + xy = A\,\}$

The proof, using the Compound rule with intermediate assertion $z = x(2y)$ is left as an exercise. *[Do you see why this intermediate assertion was choosen?]*

(b) $\{\,(x \neq 0) \wedge (z + xy = A) \wedge \neg \mathit{even}?(x)\,\}$
`begin` x `:=` $x - 1$ `;` z `:=` $z + y$ `end`
$\{\,z + xy = A\,\}$

This subgoal was proven in Example 2.

While Rule

$$\frac{\{\,P \wedge B\,\}\ S\ \{\,P\,\}}{\{\,P\,\}\ \texttt{while}\ B\ \texttt{do}\ S\ \{\,P \wedge \neg B\,\}}$$

The subgoal above the line says assertion P is "reinstated" by the body of the `while`-loop. That is, if P holds before S executes, then just after S terminates, P again holds. P need not hold throughout S's execution, just at the very beginning and just after the very end. Test B may be added to the precondition because S only executes when the `while`-test succeeds.

An assertion like P that satisfies the subgoal is called an *invariant* of the loop. As we shall soon see, invariants are the essence of program proving. Since Stmt has no `goto`s, S must execute an integral number of times, and each time P is reinstated. Hence, if and when the `while`-loop terminates, P is true and we also know at that point B is false.

EXAMPLE 4. *Prove* $\{\, z + xy = A \,\}$
```
while x ≠ 0 do
  if even?(x)
  then
    begin x := ½x ; y := 2y end
  else
    begin x := x − 1 ; z := z + y end
```
$\{\, z = A \,\}$

Exercises 1–3 prove that with precondition $x \neq 0$ and $z + xy = A$, $z + xy = A$ the `if`-statement establishes postcondition $z + xy = A$. By the While Rule we may infer that the `while`-loop establishes postconditions $z + xy = A$ and $x = 0$. Hence

$$A = z + xy = z + 0y = z$$

Using the rules

In using these rules it is often the case that what the rule produces is not exactly what is wanted. For instance, in Example 4 the desired postcondition is $z = A$ but the While Rule yields postconditions $x \neq 0$ and $z + xy = A$. Since these two conditions imply $z = A$, it is reasonable to deduce that $z = A$ also holds. This intuition is summarized in the Strengthening Rule, introduced next.

The Compound Rule says that in order to prove

$$\{\, P \,\} \texttt{ begin } S_1 \texttt{ ; } S_2 \texttt{ end } \{\, R \,\}$$

one must provide an intermediate assertion, Q, that holds between S_1 and S_2. Except in one case, there is no need to *guess* what that intermediate assertion is. What Q should be depends on what kind of statement $S - 2$ is.

Strengthening Rules

$$\frac{P \;\Rightarrow\; P' \quad \diamond \quad \{\, P' \,\}\; S\; \{\, Q' \,\} \quad \diamond \quad Q' \;\Rightarrow\; Q}{\{\, P \,\}\; S\; \{\, Q \,\}}$$

It follows from the definition of $\{\, P \,\}\; S\; \{\, Q \,\}$ that it is valid replace a postcondition, P with a *stronger* condition, P' that it implies. Similarly, postcondition Q may be replaced by a *weaker* condition, Q', provided $Q' \;\Rightarrow\; Q$.

Block Flattening

$$\frac{\{\, P \,\} \texttt{ begin } S_1 \texttt{ ; begin } S_2 \texttt{; } S_3 \texttt{ end end } \{\, Q \,\}}{\{\, P \,\} \texttt{ begin } S_1 \texttt{ ; } S_2 \texttt{ end ; } S_3 \texttt{ end end } \{\, Q \,\}}$$

Compound `begin-end` blocks are associative. No matter how they are nested, if the left-right order of the statements S_i is the same, the programs do the same thing. The double line is used above to indicate that the rule is valid in both directions. By this rule, we may include any number of statement in a block,

$$\texttt{begin}\ S_1\ \texttt{;}\ S_1\ \texttt{;}\ \ldots\ \texttt{;}\ S_n\ \texttt{end}$$

Assignment Elimination, Left

$$\frac{\{\,P\,\}\ S\ \{\,Q\big[{}^{t}_{v}\big]\,\}\quad\diamond\quad Q\big[{}^{t}_{v}\big]\ \Rightarrow\ Q\big[{}^{t}_{v}\big]}{\{\,P\,\}\ \texttt{begin}\ S\ \texttt{;}\ v\ \texttt{:=}\ t\ \texttt{end}\ \{\,Q\,\}}$$

If the rightmost statement in a compound block is an assignment statement the obvious choice of intermediate assertion is $Q\big[{}^{t}_{v}\big]$. By the Assignment Rule, this reduces one subgoal to a tautology

Assignment Elimination, Right

$$\frac{\{\,P \wedge (v = t)\,\}\ S\ \{\,Q\,\}}{\{\,P\,\}\ \texttt{begin}\ v\ \texttt{:=}t\ \texttt{;}\ S\ \texttt{end}\ \{\,Q\,\}}$$

provided v does not occur free in t or P

It is intuitive that an assignment statement will accomplish its assignment and make the program variable value equal to that of the assigning term. However, this rule may be invalid in cases where the assigned variable is "used" in the assigning term or in the precondition. For example, the assignment `x := x + 1` does not accomplish $x = x + 1$, which is impossible. This rule is typically used in program initialization, but it is better not to use it unless the side condition is well understood.

Conditional Elimination

$$\frac{\{\,P\,\}\ S_1\ \{\,B\ \Rightarrow\ Q \wedge \neg B\ \Rightarrow\ Q'\,\}\ \diamond\ \{\,Q\,\}\ S_2\ \{\,R\,\}\ \diamond\ \{\,Q'\,\}\ S_3\ \{\,R\,\}}{\{\,P\,\}\ \texttt{begin}\ S_1\ \texttt{;}\ \texttt{if}\ B\ \texttt{then}\ S_2\ \texttt{else}\ S_3\ \{\,R\,\}}$$

The idea of this rule is to analyze the two branches independently, as though the subgoals were

$$\{\,P\,\}\ \texttt{begin}\ S_1\ \texttt{;}\ S_2\ \texttt{end}\ \{\,R\,\}$$

and

$$\{\,P\,\}\ \texttt{begin}\ S_1\ \texttt{;}\ S_3\ \texttt{end}\ \{\,R\,\}$$

This will result in intermediate assertions Q and Q' for the `then` and `else` branchs, respectively. The intermediate assertion for the conditional, then, must assure that the right intermediate assertion holds, depending on the outcome of the test.

While Elimination

$$\frac{\{\,P\,\}\ S_1\ \{\,I\,\} \quad \diamond \quad \{\,I\,\}\ \texttt{while}\ B\ \texttt{do}\ S_2\ \{\,Q\,\}}{\{\,P\,\}\ \texttt{begin}\ S_1\ \texttt{;}\ \texttt{while}\ B\ \texttt{do}\ S_2\ \{\,Q\,\}}$$

The previous four elimination rules provide an intermediate assertions for all cases but those involving `while` statements. Such an intermediate assertion must be a loop *invariant* to enable use of the While Rule. It is sometimes but not always possible to "guess" what the invariant is.

We will, instead, require the programmer to provide the invariants! Who else is better able to explain the purpose of the loop?

Change the syntax of the STMT programming language in Figure 1, replacing the clause for `while` statements with

$$\begin{array}{lll} \langle \textsc{Stmt} \rangle & ::= & \\ & \vdots & \\ & \mid & \texttt{while}\ \langle \textsc{Test} \rangle\ \texttt{inv}\ \{\,\langle \textsc{Assertion} \rangle\,\}\ \texttt{do}\ \langle \textsc{Stmt} \rangle \\ & \vdots & \end{array}$$

With inviants now required, we have enough elimination rules to mechanically (i.e., without guessing) reduce any program correctness assertion to a set of purely logical subgoals, called *verification conditions*. If all the verification conditions prove true, then the PCA we started with is true as well.

The onus is on the programmer to provide the key invariant assertions. In essence, the invariants explain how the loop "reaches" its postcondition. These, together with the precondition and postcondition are all that is needed to render an assertion about a program to mathematical propositions.

Including *formal* invariants is a learned skill, but think about it this way: they say the least that *has* to be said for someone to understand the program.