0. Introduction

This course deals with the problem of *program correctness*. Informally, a program is correct, iff it solves the problem it is supposed to solve. When students start writing programs, the issue of correctness is usually dealt with in a rather informal way. They are told in English, what the program should accomplish, then they write the program and test it, in order to find any errors that might be present. However, it is clear that this is not a perfect method, since a program may fail for other input data than those for which it was tested.

To be sure, there are guidelines as to how to do the testing, so as to increase the chance of catching errors. In particular, the test data should be such that every piece of code is used. But even so it is not possible to establish correctness with absolute certainty—indeed, it happens time and again that a programming error is detected after the program has already been in use for years.

For this reason it makes sense to prove correctness of a program S in a rigorous formal mathematical way—and this is exactly what *program verification* is about. However, only precise mathematical statements can be shown mathematically, hence the first step is to express the correctness of S in a mathematically precise way. In general, a specification has the form

$$\{\phi\} \quad S \quad \{\psi\}, \tag{0.1}$$

where ϕ and ψ are mathematical formulas in which the program variables may occur as free variables. ϕ and ψ are referred to as *precondition* and *postcondition* respectively. Its meaning is as follows: 'If the assertion ϕ is true before the execution of S, then ψ holds after this execution.' This kind of assertion may still not appear mathematically precise at the current stage, inasmuch as it refers to the action of S, which in turn is usually dealt with informally when the underlying programming language $\mathcal{L}$ is taught. It becomes precise, though, as soon as the semantics of $\mathcal{L}$ is specified formally. This issue will be the topic of chapter 2.

However, a different, axiomatic approach will be followed first (chapter 1): We provide *proof rules* from which assertions like (0.1) can be formally deduced. Under these circumstances a proof of (0.1) can be accomplished even without interpreting this formula, without associating a meaning with it. Of course, the interpretation of specifications is still important as a motivation—without it the proof rules would appear arbitrary and would not make sense.

On the one hand, proof rules provide specifications for certain atomic statements like assignments, which cannot be decomposed further. On the other hand, they allow a obtain a specification for a complex instruction (or sequence of instructions) from corresponding specifications for its constituents. It is obvious then that it does not suffice to consider a specification for a complete program P which we wish to use in practice and for which verification is desired. For a proof one also has to deal with specifications of the form (0.1), where S is some part of P. Indeed, in the specifications to be dealt with, S will contain no program heading, no declarations of variables etc., and usually no input- and output instructions. Hence S alone will not qualify as a program in a language like PASCAL or FORTRAN, though it may be a program of some abstractly defined programming language, as it is considered in computability theory.

As we develop the methods for program verification, type declarations will be ignored; it will just be assumed that the type of each variable is known in each example. read- and write statements will be ignored for the most part, but in chapter 4 we show how to deal with files, and this involves read and write as well.

One point about the meaning of (0.1) has to be noted. Of course, ψ may hold or fail upon termination, but there is a third possibility: The execution of S may not terminate. Accordingly we have to distinguish *total* and *partial* correctness. Assertion (0.1), as it stands, means partial correctness, so the meaning should be specified more precisely as follows:

> If the system of values of the variables before execution of S is such that ϕ holds, and if the execution terminates, then ψ holds upon termination.

Thus (0.1) does not claim that S does indeed terminate, in contrast to the assertion of total correctness, which is specified by including a downward arrow after S. Thus

$$\{\phi\} \quad S\downarrow \quad \{\psi\} \tag{0.2}$$

has the following meaning:

> If the system of values of the variables before execution of S is such that ϕ holds, then the execution terminates, and ψ holds upon termination.

Chapter 1 contains an example for program verification where no details are omitted. Here we go back to the very proof rules and point out every application of them. This is analogous to an exercise in mathematical logic, where a simple theorem is to be shown from the axioms and rules of inference, without applying any of the shortcuts used in common mathematical reasoning.

Naturally, this way of proceeding is very awkward and unpractical and makes sense only as an exercise. In program verification, too, certain shortcuts are applied without abandoning mathematical precision. Even so, verification is a very tedious task except for very short programs. This is why in real life verification is usually carried out only for exceptionally important programs, which will typically be used over and over. On the other hand, many steps in the verification process can be done mechanically, without any particular creative effort. Hence it is not surprising that there exist software systems to support verification. Such systems work interactively, and the user only has to supply a few pieces of information in addition to the specification to be shown. In particular, the user has to supply *loop invariants*, namely conditions which hold before and/or after every execution of some loop.

Example 0.1: We wish to show that the following program P computes $K := \lfloor N/D \rfloor$ for given N and D:

$K := 0;$
$S := D;$
while $S \leq N$ **do**
    begin
        $K := K + 1;$
        $S := S + D$
    end

Of course, this program serves only as an illustration and has no practical value, since real-life programming languages have operations which yield $\lfloor N/D \rfloor$ directly. All variables are of type **integer**, which is the default assumption. What is the right precondition? Namely, what condition must N and D satisfy initially in order that the program works? Surely, N must not be negative. Indeed, if $N < 0 < D$, then the **while**-loop will not be executed, and the program will set K to 0, which is not $\lfloor N/D \rfloor$. Likewise, D must not be negative; indeed, if $N \geq 0 \geq D$, then we have $S \leq 0 \leq N$ before and after every execution of the **while**-loop, so the program will not terminate. We might omit the clause $D > 0$ from the precondition if we only wish to obtain a specification which asserts partial correctness. This makes little sense, though: the writer of this program surely did not intend to use it for $D \leq 0$. Finally, we check if the program works for $N = 0$ and $D > 0$. In this case the **while**-condition will already fail when it is checked for the first time, the loop will not be executed, and the final value of K will be 0, which is correct. Thus we get the precondition $N \geq 0 \,\&\, D > 0$.

Since the goal is to show that P computes $K := \lfloor N/D \rfloor$, we may take $K = \lfloor N/D \rfloor$ as the postcondition. Moreover, the discussion above indicates P terminates if our precondition is satisfied before execution, which means that the program is totally rather just partially correct. So the following specification comes to mind:

$$\{N \geq 0 \,\&\, D > 0\} \quad P\!\downarrow \quad \{K = \lfloor N/D \rfloor\}. \tag{0.3}$$

This specification is indeed correct, but unsatisfactory; do you see why? What we wish to show is that P computes the quotient of the *original* values of N and D, whereas (0.3) asserts $K = \lfloor N/D \rfloor$ holds upon termination for the *final* values of N and D. Of course, N and D are not changed by the program, but this fact is not expressed by (0.3). Indeed, (0.3) would also be true if P were the program

$$N := 0; \;\; D := 1; \;\; K := 0;$$

this program does not in any way accomplish what is desired. In order to obtain a more appropriate specification, one has to introduce *mathematical* variables n, d which denote the original values of N and D. Then the right specification is

$$\{N = n \geq 0 \,\&\, D = d > 0\} \quad P\!\downarrow \quad \{K = \lfloor n/d \rfloor\}. \tag{0.4}$$

This assertion must be understood as being true for all n and d, but we follow the usual convention of omitting universal quantifiers in mathematical formulas. (0.4) means:

> Whatever initial values n, d the variables N and D have before execution of P, if $n \geq 0$ and $d > 0$, then the execution of P will terminate, and K will have value $\lfloor n/d \rfloor$ upon termination.

Let ϕ and ψ denote the precondition and postcondition in (0.4), P_1 the body of the **while**-loop. What is an appropriate loop invariant? We need a condition ρ which holds before and after each execution of P_1, and which allows to deduce ψ upon termination. Trivially, the conditions $N = n \geq 0$ and $D = d > 0$ always hold, since N and D do not change. Next note that there is a simple relation between K and S. Indeed, after the first execution of P_1, the values of K and S are 1 and $2d$, after the second they are 2 and $3d$, and in general after the n-th execution n and $(n+1)d$. Thus $S = (K+1)d$ holds before and after each execution of P_1, and clearly it is not only possible but also necessary to include this clause in ρ. Finally, we have to express that K and S do not become too large. S will exceed n after the last execution of P_1, but n is an upper bound for $Kd = S - d$. Accordingly we choose the following loop invariant ρ:

$$N = n \geq 0 \,\&\, D = d > 0 \,\&\, S = (K+1)d \,\&\, Kd \leq n.$$

Upon termination the **while**-condition fails, so $S > N$ holds in addition to ρ. Thus we have

$$N = n \geq 0 \,\&\, D = d > 0 \,\&\, S = (K+1)d \,\&\, Kd \leq n \,\&\, S > N,$$

implying $Kd \leq n < (K+1)d$, which in turn implies the postcondition ψ. Of course, this argument has to be made more formal, and for a proof of (0.4) we also have to show that ρ holds before the first execution of P_1, that each execution of P_1 preserves ρ, and that the execution of P terminates if ϕ is satisfied initially. How all this can be done will be shown in chapter 1.

How can we find an appropriate loop invariant in general? This is one of those steps in the verification process which cannot be done mechanically, and there exists no recipe which works in every case. But if one understands how the program works, then it is usually not too hard to see what conditions hold at each point. If verification does not succeed after a formula ρ has been tried as loop invariant, then it may be necessary to modify ρ, for instance, by adding some 'trivial' clauses (like $N = n \geq 0$ in our example).

We conclude the introduction by making a few conventions. We use capital roman letters for program variables, corresponding lowercase roman letters for their values. For instance, if X is a variable, then x will denote the value of X at some stage of the computation, most typically at the beginning. Programs and program segments are also denoted by roman capitals, and formulas by Greek letters. Calligraphic letters will be used for certain other kinds of objects like *environments*, see chapter 2. As noted above, the program segments in the specifications to be considered contain no declarations of variables etc. Usually the data type of a variable X will be clear, and if necessary it will be stated in the context. If the value of X is a number, then the default assumption is that X is of type **integer** rather than **real**.

We provide no formal definition of the programming language on which these lecture notes are based, but it resembles PASCAL closely. Most constructs are well-known anyway and occur in several languages. Whenever there may be some doubt about the semantics, an explanation will be given.

1. Foundations of program verification

1.1. The basic proof rules.

In this section we provide proof rules which yield specifications of the form (0.1) for various types of statements S, that is, only partial correctness is considered. We start with the two types of 'atomic' statements which do not contain simpler ones inside. Denoting the empty statement (or program) by Λ, we get the following *rule for the empty statement*:

$$\text{(R1)} \qquad \{\phi\} \quad \Lambda \quad \{\phi\}.$$

Thus if ϕ holds initially, then ϕ also holds after the execution of Λ, that is, after doing nothing. This is of course trivial.

The *rule for assignment statements* (*assignment rule*) is more significant. Consider the instruction $X := A$, where A is some term. It turns out that an appropriate precondition ϕ can be determined easily from a given postcondition ψ. Thus we ask: what is the condition ϕ which has to hold before execution of $X := A$ in order that ψ holds afterwards? ϕ has to assert that the old values of the variables are such that ψ holds for the new ones. Now the new values coincide with the old ones except that the new value of X is given by (the old value of) A. Hence ϕ asserts that 'ψ holds with X replaced by A,' and ϕ is obtained from ψ by substituting A for X. We use the following notation for substitution in general:

Notation: $E\langle t_1/x_1, \ldots, t_n/x_n\rangle$ is the result of simultaneously substituting t_1 for x_1, $\ldots$, t_n for x_n in the expression E (which may be a formula or a term).

Presently we only need the case where a single variable is substituted. We obtain the following assignment rule:

$$\text{(R2)} \qquad \{\psi\langle A/X\rangle\} \quad X := A \quad \{\psi\}.$$

We say that the precondition $\psi\langle E/X\rangle$ is obtained by backward substitution.

The remaining proof rules to be introduced in this chapter use hypothesis, that is, they assert that certain specifications can be deduced from facts already known. Again we introduce a general

Notation: If Φ_1, $\ldots$, Φ_n are assertions (formulas), then

$$\frac{\Phi_1, \ldots, \Phi_n}{\Phi}$$

means that Φ can be deduced from Φ_1, $\ldots$, Φ_n.

The *implication rule* is

$$\text{(R3)} \qquad \frac{\phi \Longrightarrow \phi', \quad \{\phi'\} \quad S \quad \{\psi'\}, \quad \psi' \Longrightarrow \psi}{\{\phi\} \quad S \quad \{\psi\}};$$

this is of course obvious. For $\phi = \phi'$ or $\psi = \psi'$ respectively we get the special cases

$$\frac{\{\phi\} \quad S \quad \{\psi'\}, \quad \psi' \Longrightarrow \psi}{\{\phi\} \quad S \quad \{\psi\}}, \qquad \frac{\phi \Longrightarrow \phi', \quad \{\phi'\} \quad S \quad \{\psi\}}{\{\phi\} \quad S \quad \{\psi\}}.$$

The *rule for statement sequences* shows how to obtain a specification for a program segment S of the form $S_1; \ldots S_k$; of course the segments S_i are executed in order. As an axiom it suffices to consider the case $k = 2$:

$$\text{(R4)} \qquad \frac{\{\phi\} \quad S \quad \{\rho\}, \quad \{\rho\} \quad S' \quad \{\psi\}}{\{\phi\} \quad S; S' \quad \{\psi\}}.$$

Here S and S' may in turn be sequences of statements, not necessarily single statements. By applying the above rule repeatedly we obtain a corresponding rule for sequences of k program segments. The rule for the empty statement may be regarded as a special case with $k = 0$.

Sometimes a statement sequence S has to be delimited by **begin** ... **end**. However, this serves only to assure that the program containing S is parsed correctly; it does not alter the effect of the execution of S. Accordingly we get the following *rule for compound statements*:

$$\text{(R5)} \qquad \frac{\{\phi\} \quad S \quad \{\psi\}}{\{\phi\} \quad \mathbf{begin}\ S\ \mathbf{end} \quad \{\psi\}}.$$

Next we turn to conditional statements. In order to deduce

$$\{\phi\} \quad \mathbf{if}\ B\ \mathbf{then}\ S\ \mathbf{else}\ S' \quad \{\psi\},$$

two facts have to be established: (i) If in addition to ϕ, B holds prior to execution, then ψ holds after execution of S (since S is executed in this case). (ii) If $\neg B$ holds in addition to ϕ prior to execution, then ψ holds after execution of S'. So the first *rule for conditional statements* is

$$\text{(R6)} \qquad \frac{\{\phi \mathbin{\&} B\} \quad S \quad \{\psi\}, \quad \{\phi \mathbin{\&} \neg B\} \quad S' \quad \{\psi\}}{\{\phi\} \quad \mathbf{if}\ B\ \mathbf{then}\ S\ \mathbf{else}\ S' \quad \{\psi\}}.$$

The second rule for conditional statements deals with the case where the **else**-part is missing, or, equivalently, where S' is empty. Now the second hypothesis reduces to $\{\phi \mathbin{\&} \neg B\}\ \Lambda\ \{\psi\}$, which is clearly equivalent to $\phi \mathbin{\&} \neg B \implies \psi$. (The implication $[\phi \mathbin{\&} \neg B \implies \psi] \implies [\{\phi \mathbin{\&} \neg B\}\ \Lambda\ \{\psi\}]$ can be deduced by the rule for the empty statement and the implication rule.) Thus we get:

$$\text{(R7)} \qquad \frac{\{\phi \mathbin{\&} B\} \quad S \quad \{\psi\}, \ \phi \mathbin{\&} \neg B \implies \psi}{\{\phi\} \quad \mathbf{if}\ B\ \mathbf{then}\ S \quad \{\psi\}}.$$

Finally, we have to consider **while**-loops. As noted in the introduction, a loop invariant for **while** B **do** S cannot be computed mechanically—we have to find it by inspection. However, the claim that ϕ is a loop invariant can be represented as a specification: it holds if each execution of S preserves condition ϕ. This is expressed by the specification $\{\phi \mathbin{\&} B\}\ S\ \{\phi\}$, since prior to the execution of S, B holds in addition to ϕ. From $\{\phi \mathbin{\&} B\}\ S\ \{\phi\}$ the following may be deduced: if ϕ holds before the first execution of S, that is, before the execution of the **while**-statement as a whole, then ϕ will still hold after the last execution, and in addition $\neg B$ will hold at this point (since an exit from the loop has just occurred). Thus we obtain the following **while**-*rule:*

$$\text{(R8)} \qquad \frac{\{\phi \mathbin{\&} B\} \quad S \quad \{\phi\}}{\{\phi\} \quad \mathbf{while}\ B\ \mathbf{do}\ S \quad \{\phi \mathbin{\&} \neg B\}}.$$

Besides conditionals and **while**-loops there exist other control structures in various languages. PASCAL contains **case**-statements, **repeat-until**-loops and **for**-loops, and we will consider these constructs in section 1.4. These control structures are less fundamental than the ones considered so far, since they can be avoided if desired: every program can be translated into an equivalent one without **case**, **repeat-until** and **for**. In principle it would even be possible to avoid conditional statements and simulate them by **while**, but this would be very unpractical.

1.2. Two simple examples.

Example 1.1: As promised, we now return to the program P from example 0.1, and we prove partial correctness in full detail, using only the proof rules from section 1.1 and pointing out every application of them. The precondition ϕ, postcondition ψ and loop invariant ρ have already been introduced:

$$\begin{aligned} &\phi\colon \quad N = n \geq 0 \mathbin{\&} D = d > 0;\\ &\psi\colon \quad K = \lfloor n/d \rfloor;\\ &\rho\colon \quad N = n \geq 0 \mathbin{\&} D = d > 0 \mathbin{\&} S = (K+1)d \mathbin{\&} Kd \leq n. \end{aligned}$$

Our goal is to prove:

$$\{\phi\} \quad P \quad \{\psi\}, \tag{1.1}$$

which is the same as (0.4), except that only partial correctness is asserted. (How to prove *total* correctness will be shown in section 1.3.) At the top level we have to apply the rule for statement sequences, since P is such a sequence. We divide P into the two segments P_0 and P_2, where P_0 consists of the two assignments before the **while**-loop, and P_2 is the **while**-statement itself. The *body* of the **while**-loop was already denoted by P_1; we now stipulate that P_1 does not include **begin** and **end**. The condition which supposedly holds after the execution of P_0 and before the execution of P_2 is the loop invariant ρ. Hence we have to show:

$$\{\phi\} \quad P_0 \quad \{\rho\}, \quad \text{and} \tag{1.2}$$

$$\{\rho\} \quad P_2 \quad \{\psi\}. \tag{1.3}$$

From (1.2) and (1.3), (1.1) follows by (R4).

The assignment rule suggests the following general method for proving a specification of the form

$$\{\eta\} \quad S_1; \ldots; S_n \quad \{\theta\}, \tag{1.4}$$

where each S_i is an assignment statement $X_i := A_i$. Starting from the postcondition θ, carry out a sequence of substitutions corresponding to the assignments S_i: $\eta_{n-1} = \theta\langle A_n/X_n\rangle$, $\eta_{n-2} = \eta_{n-1}\langle A_{n-1}/X_{n-1}\rangle$, $\ldots$, $\eta_0 = \eta_1\langle A_1/X_1\rangle$. Thus η_i is the assertion which supposedly holds before the execution of S_{i+1} and (except for $i = 0$) after the execution of S_i, and (R2) yields $\{\eta_i\}$ S_{i+1} $\{\eta_{i+1}\}$. Finally, prove $\eta \Longrightarrow \eta_0$ (which is a purely mathematical assertion). After this has been accomplished, (1.4) follows by one application of (R3) and several applications of (R2) and (R4).

Applying this method for (1.2), backward substitution yields the formulas

$$\begin{aligned} &\eta_1\colon \quad N = n \geq 0 \mathbin{\&} D = d > 0 \mathbin{\&} D = (K+1)d \mathbin{\&} Kd \leq n,\\ &\eta_0\colon \quad N = n \geq 0 \mathbin{\&} D = d > 0 \mathbin{\&} D = (0+1)d \mathbin{\&} 0d \leq n. \end{aligned}$$

(R2) yields

$$\{\eta_0\} \quad K := 0 \quad \{\eta_1\}, \quad \text{and} \tag{1.5}$$

$$\{\eta_1\} \quad S := D \quad \{\rho\}, \tag{1.6}$$

whereupon (R4) yields

$$\{\eta_0\} \quad P_0 \quad \{\rho\}. \tag{1.7}$$

Next we have to show the implication

$$\phi \Longrightarrow \eta_0, \tag{1.8}$$

that is, $N = n \geq 0 \,\&\, D = d > 0 \Longrightarrow N = n \geq 0 \,\&\, D = d > 0 \,\&\, D = (0+1)d \,\&\, 0d \leq n$, which is trivial. Then (1.2) follows by (1.7) and (1.8) using (R3).

Now we turn to (1.3). In order to apply the **while**-rule, we have to show that ρ is indeed a loop invariant, that is,

$$\{\rho \,\&\, S \leq N\} \quad P_1 \quad \{\rho\}. \tag{1.9}$$

The proof proceeds analogously to (1.2). This time we denote the results of each backward substitution by θ_1 and θ_0 and get

$$\theta_1\colon \quad N = n \geq 0 \,\&\, D = d > 0 \,\&\, S + D = (K+1)d \,\&\, Kd \leq n,$$
$$\theta_0\colon \quad N = n \geq 0 \,\&\, D = d > 0 \,\&\, S + D = ((K+1)+1)d \,\&\, (K+1)d \leq n.$$

As before, (R2) yields

$$\{\theta_0\} \quad K := K+1 \quad \{\theta_1\} \quad \text{and} \tag{1.10}$$

$$\{\theta_1\} \quad S := S + D \quad \{\rho\}, \tag{1.11}$$

and (R4) yields

$$\{\theta_0\} \quad P_1 \quad \{\rho\}. \tag{1.12}$$

We have to show:

$$\rho \,\&\, S \leq N \Longrightarrow \theta_0, \tag{1.13}$$

that is,

$$N = n \geq 0 \,\&\, D = d > 0 \,\&\, S = (K+1)d \,\&\, Kd \leq n \,\&\, S \leq N$$
$$\Longrightarrow N = n \geq 0 \,\&\, D = d > 0 \,\&\, S + D = ((K+1)+1)d \,\&\, (K+1)d \leq n.$$

This implication, like (1.8) is obvious, for instance, $(K+1)d \leq n$ follows from $N = n \,\&\, S = (K+1)d \,\&\, S \leq N$. From (1.12) and (1.13) we deduce (1.9), using (R3).

Using (R8), we can deduce from (1.9):

$$\{\rho\} \quad P_2 \quad \{\rho \,\&\, \neg(S \leq N)\}. \tag{1.14}$$

This step does not involve an application of (R5), as explaned in remark 1.1 below. In order to obtain (1.3), the implication rule has to be applied once more, and we have to show:

$$\rho \,\&\, \neg(S \leq N) \Longrightarrow \psi, \tag{1.15}$$

which is

$$N = n \geq 0 \,\&\, D = d > 0 \,\&\, S = (K+1)d \,\&\, Kd \leq n \,\&\, \neg(S \leq N) \Longrightarrow K = \lfloor n/d \rfloor.$$

The left-hand side of this implication implies $S = (K+1)d > N = n \,\&\, Kd \leq n$, hence $K \geq 0$ (otherwise $(K+1)d \leq 0 \leq N$), hence $K \leq n/d < K+1$, which is equivalent to $K = \lfloor n/d \rfloor$. This shows (1.15), whereupon (R3) yields (1.3), completing the proof.

Remark 1.1: Consider a **while**-statement P of the form **while** B **do** S, and assume that S is a statement sequence $S_1; \ldots; S_n$. Then P really has the form **while** B **do begin** $S_1; \ldots S_n$ **end**. Inserting **begin** ... **end** is part of the substitution operation; without it the body would consist only of S_1. This is analogous to substitution in mathematical expressions, when parenthesis sometimes have to be added. Conversely, assume that we wish to apply to a **while**-statement given in the form **while** B **do begin** $S_1; \ldots S_n$ **end**. Then we stipulate that the rôle of S in (R8) is played by $S_1; \ldots; S_n$ (without **begin** ... **end**). A consequence of this convention is that applications of (R5) are often unnecessary. The same remark applies for conditional statements and other control structures to be considered in section 1.4.

The verification just carried out illustrates that the proof rules really allow to reduce the proof of a specification to the proof of purely mathematical facts like (1.8), which, to be sure, are not always quite as simple as in the example above. As noted in the introduction, some shortcuts are usually applied when verification is carried out in practice. For one thing, whenever, a pre- or postcondition is computed, for instance by backward substitution, then the result may be simplified immediately; this involves an implicit application of (R3). When we wish to prove a specification for a sequence of assignments like (1.4), then we compute $\eta_{n-1}, \ldots, \eta_0$ by backward substitution as described, and prove $\eta \Longrightarrow \eta_0$; there is no need to mention the applications of (R2)–(R4) which are involved. This is how we proceed in the next example.

Example 1.2: The following program S represents what is a favorite example of an algorithm in textbooks—the Euclidean algorithm for computing the greatest common divisor of two integers x, y. First we clarify the definition of $\gcd(x, y)$. If x and y are positive, then the definition is clear and well-known, in fact, the expression 'greatest common divisor' is self-explanatory. In the case where one of the two numbers is 0, the other nonnegative, we have $\gcd(x, 0) = \gcd(0, x) = x$ for $x \geq 0$. Finally, the case where x and/or y is negative is covered by the rule $\gcd(x, -y) = \gcd(-x, y) = \gcd(-x, -y) = \gcd(x, y)$. This can be summarized by specifying that (i) $d = \gcd(x, y)$ is maximal among all common divisors of x and y with respect to the divisibility relation, that is, every other common divisor d' divides d ($d'|d$), and (ii) $d \geq 0$. Note that condition (i) alone would still leave the choice between d and $-d$.

In the program the values x, y are initially stored in the variables X, Y. We use the notation $|X|$ for the absolute value of X at the right side of an assignment statement, which of course is not allowed in real programming languages.

while $Y \neq 0$ **do**
  begin
    $Z := X \bmod Y$;
    $X := Y$;
    $Y := Z$
  end;
$X := |X|$

Similarly as in the previous example, the specification has to express that the program computes the greatest common divisor of the *origial* values of X and Y, which are given by x and y. The data type is **integer** by default, and no other assumption about x and y

is needed. Hence the precondition is

$$\phi: \quad X = x \,\&\, Y = y.$$

Before proceeding further, a general comment is in order, which will be discussed further in remark 1.2. Here, like in most examples in these lecture notes, programs are presented first, and verification is carried out afterwards. However, this is not the ideal way to proceed for serious programming tasks. Even in such a simple example as the current one, there are some points to discuss, which should be kept in mind while the program is written.

The most common version of the Euclidean algorithm is the one where

(A1) The input numbers x, y and all remainders are nonnegative.

If this is the case, then upon exit from the loop, Y contains the last remainder, which is 0, and X contains the next to last remainder, which is the desired value $\gcd(x, y)$. As will be seen shortly, this is also true without assumption (A1), except that X may have the wrong sign, so the assignment $X := |X|$ is needed to ensure that the result is positive. Since X contains the result upon termination, the postcondition is

$$\psi: \quad X = \gcd(x, y).$$

There are two reasons why (A1) may fail. The first, obvious one is that x and y were not assumed to be positive. Secondly, several definitions of mod are possible, and the result is not necessarily nonnegative. The following property holds for all reasonable definitions of mod, and for all definitions in common programming languages:

(1.16) If $v \neq 0$, then $r = u \bmod v$ is a remainder of u and v, that is, a number of the form $u - qv$ with integer q, and $|r| < |v|$.

This leaves several options for $r = u \bmod v$: r may be the smallest nonnegative remainder, which satisfies $0 \leq r < |v|$, or the absolutely smallest remainder, which satisfies $-\lfloor v/2 \rfloor < r \leq \lfloor v/2 \rfloor$, or, finally, the sign of r may be the same as that of u (or v) unless $r = 0$. The last of these alternatives holds in PASCAL, for instance, but each provides a unique characterization of r in combination with (1.16). In the case $v = 0$ the only reasonable definition is $u \bmod 0 = u$, or else $u \bmod 0$ may be undefined.

We will see that program S works for *each* definition of mod, and only (1.16) will be needed for the verification. Since programs corresponding to different definitions of mod must be considered different themselves (even if they consist of the same sequence of characters), we actually verify several programs at the same time. An alternative would be to omit the assignment $X := |X|$ at the end and put $X := |X|$; $Y := |Y|$ at the beginning. This works if mod yields the smallest nonnegative remainder; then we obtain once again the most common version of the Euclidean algorithm, where all remainders are nonnegative.

The property $|u \bmod v| < |v|$ implies that every execution of the **while**-loop decreases the value of $|Y|$; this guarantees termination, as shown in section 1.3. The fact that $u \bmod v$ is of the form $u - qv$ with $v \in \mathbb{Z}$ allows to show $\gcd(u, v) = \gcd(v, u \bmod v)$, which in turn means that each execution of the **while**-loop leaves the value $\gcd(X, Y)$ unchanged. This is the key idea of the algorithm; it suggests the following loop invariant:

$$\rho: \quad \gcd(X, Y) = \gcd(x, y).$$

No other clauses are needed.

We now turn to the verification. The specification we have to prove is of course

$$\{\phi\} \quad S \quad \{\psi\}. \tag{1.17}$$

Let S_0 be the body of the **while**-loop, S_1 the **while**-statement itself. A specification for S_1 is obtained by (R8); in order to apply this rule, we have to show:

$$\{\rho \,\&\, Y \neq 0\} \quad S_0 \quad \{\rho\}. \tag{1.18}$$

The proof of (1.18) will be given below, but first we complete the verification under the assumption that (1.18) holds. The **while**-rule yields

$$\{\rho\} \quad S_1 \quad \{\rho \,\&\, Y = 0\}. \tag{1.19}$$

Trivially, we have

$$\phi \Longrightarrow \rho; \tag{1.20}$$

together with (1.19) this yields by (R3):

$$\{\phi\} \quad S_1 \quad \{\rho \,\&\, Y = 0\}. \tag{1.21}$$

In order to obtain (1.17), we have to apply (R4) to S_1 and to the assignment $X := |X|$; for this purpose we need

$$\{\rho \,\&\, Y = 0\} \quad X := |X| \quad \{\psi\}. \tag{1.22}$$

Backward substitution from ψ yields $|X| = \gcd(x, y)$, so we have to show

$$\rho \,\&\, Y = 0 \Longrightarrow |X| = \gcd(x, y), \tag{1.23}$$

that is, $\gcd(X, Y) = \gcd(x, y) \,\&\, Y = 0 \Longrightarrow |X| = \gcd(x, y)$. This implication reduces to $Y = 0 \Longrightarrow |X| = \gcd(X, Y)$, or $|X| = \gcd(X, 0)$, which follows clearly from the definiton of gcd, as specified above. This completes the deduction of (1.17) from (1.18).

It remains to show (1.18). Starting from ρ, we apply backward substitution, which yields the following formulas in order:

$$\gcd(X, Z) = \gcd(x, y);$$
$$\gcd(Y, Z) = \gcd(x, y);$$
$$\gcd(Y, X \bmod Y) = \gcd(x, y).$$

Accordingly, it remains to prove the implication

$$\rho \,\&\, Y \neq 0 \Longrightarrow \gcd(Y, X \bmod Y) = \gcd(x, y), \tag{1.24}$$

or

$$\gcd(X, Y) = \gcd(x, y) \,\&\, Y \neq 0 \Longrightarrow \gcd(Y, X \bmod Y) = \gcd(x, y).$$

Trivially, (1.24) follows from

$$Y \neq 0 \Longrightarrow \gcd(X, Y) = \gcd(Y, X \bmod Y), \tag{1.25}$$

which is the key property of the function gcd mentioned above. Assume $Y \neq 0$, and let $d = \gcd(X, Y)$, $d' = \gcd(Y, X \bmod Y)$; we have to show $d = d'$. Since d is a common divisor of X and Y, d is also a common divisor of Y and $X \bmod Y$, since $X \bmod Y$ is of the form $X - qY$ (using (1.16)). Hence the definiton of $d' = \gcd(Y, X \bmod Y)$ implies $d|d'$. An analogous argument (using $X = X \bmod Y + qY$) shows $d'|d$. Finally, d and d' are both values of the function gcd and therefore nonnegative. Hence we have indeed $d = d'$, as claimed.

Remark 1.2: In practice it is recommended to do part of the verification not after, but *during* the development of the program. Thus, while the code is written, one should keep track of the conditions that are supposed to hold at critical places, in particular of loop invariants. This is typically done when algorithms are explanced in textbooks; such an explanation often contains phrases of the form 'At this point, this and this holds.' In these lecture notes we do not follow these recommendation and present programs directly, since program development is not the topic.

1.3. Proving total correctness.

Most of the constructs considered so far cannot cause nontermination of a program. For instance, a conditional statement **if** B **then** S **else** S' can fail to terminate only if an execution of S or S' fails to terminate, and the **if**-**then**-**else**-construction itself is not the culprit. This means that (R6) can be adjusted readily to deal with total correctness:

$$(\text{R6}') \qquad \frac{\{\phi \mathbin{\&} B\} \quad S{\downarrow} \quad \{\psi\},\ \{\phi \mathbin{\&} \neg B\} \quad S'{\downarrow} \quad \{\psi\}}{\{\phi\} \quad \textbf{if } B \textbf{ then } S \textbf{ else } S' \downarrow \quad \{\psi\}}.$$

Similarly, for the proof rules (R1), (R3), (R4), (R5) and (R7) there exist corresponding proof rules (R1$'$) etc. obtained by replacing every specification $\{\phi\}\ S\ \{\psi\}$ by $\{\phi\}\ S{\downarrow}\ \{\psi\}$.

For assignment statements $X := A$ the situation is more critical. Indeed, if A contains a function call $F(X)$, and if the computation of $F(X)$ fails to terminate, then the execution of the assignment does not terminate either. For instance, assume that the user declares the factorial function recursively in the form

> **function** Fact(X);
> **begin**
>   **if** $X = 0$ **then** **return**(1) **else** **return**$(X * \text{Fact}(X-1))$
> **end**

Then the assignment $Y := \text{Fact}(-1)$ will run into an infinite loop.

Our general convention is that $F(x)$ is undefined if the computation of $F(X)$ fails to terminate with x as the value of X. Then the question arises: Is an adjusted assignment rule for total correctness formally correct in the form

$$\{\psi\langle A/X\rangle\} \quad X := A \downarrow \quad \{\psi\}? \tag{1.26}$$

If the computation of A does not terminate, then the value of A is undefined, and in the precondition $\psi\langle A/X\rangle$ predicates may be applied to an undefined value. Under these circumstances it is not clear how the truth value of $\psi\langle A/X\rangle$ should be defined. There are several possibilities all of which fit our purpose; for instance, we may introduce a third truth value **undefined** besides **true** and **false**. The convention we use is as follows. Let ϕ be a formula of the form $P(e_1, \ldots, e_n)$, where P is a predicate and e_1, $\ldots$, e_n are expressions. Infix notation may also be used, as in $e_1 \leq e_2$. Then ϕ has truth value **false** if the value of one of the e_i is undefined, except that the formula $e_1 = e_2$ has truth value **true** if the values of e_1 and e_2 are both undefined.

Now it is clear that (1.26) is not a correct rule as it stands: If P is a predicate, then $\{\neg P(A)\}\ X := A \downarrow \{\neg P(X)\}$ is an instance of (1.26), and the precondition has truth value **true** in the case where the value of A is undefined, yet the assignment does not terminate. Nor would it help to modify our convention about the truth value of a formula involving undefined terms: If ψ contains neither X nor an undefined term, then $\{\psi\}\ X := A\ \{\psi\}$ is an instance of (1.26), which still fails if the value of A is undefined. Instead we use the following variant of (R2):

$$(\text{R2}')\qquad \{A\!\downarrow \& \ \psi\langle A/X\rangle\}\quad X := A\downarrow \quad \{\psi\}.$$

Here the notation $A\!\downarrow$ means that the value of A is defined, so that its computation terminates.

Besides nontermination, there is something else that can go wrong with an assignment: a run-time error may occur, for instance, in case of a division by 0. This is an issue which has to do with computing in practice, and as such it will be dealt with in chapter 3. From an abstract theoretical point of view it is more natural to discard run-time errors, and to assume that the computation of any expression either yields it value or does not terminate. For the time being we make no distinction between a run-time error and nontermination, so we reinterpret (0.1) as follows:

> If the system of values of the variables before execution of S is such that ϕ holds, and if the execution terminates without run-time error, then ψ holds upon termination.

The reinterpretation of (0.2) is obvious, and $A\!\downarrow$ also asserts that the computation of A leads to no run-time error (besides terminating).

Remark 1.3: Even though (R2$'$) is the right version of the assignment rule for total correctness, we may use the simpler form (1.26) in many cases. This is justified if all functions (operations) occurring in A are known to be total, so that the clause $A\!\downarrow$ trivially holds, or if $A\!\downarrow$ follows from the remaining clauses. This will be illustrated when the termination proof is done for example 1.2.

Among the constructs causing nontermination, **while**-loops are the ones which primarily come to mind. In example 0.1 it is obvious enough why the program terminates, if the precondition is initially satisfied: every execution of the **while**-loop increases the value of S, until it exceeeds that of N, whereupon an exit from the loop occurs. Equivalently, the value of $N - S$ continues to decrease until it becomes negative. Similarly, in example 1.2, it is the value $|Y|$ which decreases with every execution of the loop.

Before describing the general method, recall the following definition. A partial ordering $\preceq$ on a set A is *well-founded*, if every nonempty subset $B \subseteq A$ has a minimal element a (with respect to $\preceq$); this means that a satisfies $(\forall b \in B)(\neg b \prec a)$. If in addition $\preceq$ is linear, then $\preceq$ is called a *well-ordering*; in this case a minimal element a of $B \subseteq A$ is the *smallest* element of B, that is, it satisfies the stronger condition $(\forall b \in B)(a \leq b)$.

Some examples of partial orderings which are or are not well-founded will be given below. At the moment we only note the most important example: the usual ordering $\leq$ is a well-ordering on ω, the set of natural numbers including 0.

Now, in order to prove that a **while**-loop **while** B **do** S terminates, proceed as follows. Let ρ be the loop invariant, as in the proof of partial correctness. For one thing,

(i) $\{\rho \,\&\, B\}\ S\!\downarrow\ \{\rho\}$

must hold, analogously to (R8). Next, find an appropriate expression E, a subset W of the data type of E, and a *well-founded* partial ordering $\preceq$ on W such that

(ii) The value of E belongs to W before every execution of S, that is, $E \in W$ follows from $\rho \,\&\, B$, and

(iii) If the loop condition B still holds after some execution of S, then the value of E is smaller after than it was before that execution of S (with respect to $\preceq$).

If (i)–(iii) are satisfied, then the loop cannot be executed an infinite number of times, since otherwsie the sequence of values of E after each execution would form an infinite decreasing sequence of elements of W, contradicting the fact that $\preceq$ is well-founded. Accordingly, we obtain the following rule for the total correctness of **while**-statements:

$$\text{(R8}'\text{)} \qquad \frac{\phi \,\&\, B \Longrightarrow E \in W, \ \{\phi \,\&\, B \,\&\, E = e\} \quad S\!\downarrow \quad \{\phi \,\&\, (E < e \vee \neg B)\}}{\{\phi\} \quad \textbf{while } B \textbf{ do } S\downarrow \quad \{\phi \,\&\, \neg B\}}.$$

If W consists of the *whole* data type of E, then the first hypothesis of (R8′) is vacuously true. Moreover, it is possible to prove the second hypothesis in the stronger but simpler form $\{\phi \,\&\, B \,\&\, E = e\}\ S\!\downarrow\ \{\phi \,\&\, E < e\}$. As a matter of fact, it will be shown below that every total correctness proof for **while**-loops can be carried out in such a way that this special case applies. However, this would usually require to use a rather artificial well-founded partial ordering $\preceq$, so we prefer to present the proof rule in the above form.

Now it is time to redo the examples from section 1.2, proving total correctness this time. Example 1.2 is considered first. The precondition ϕ, postcondition ψ and loop invariant do not change, and as before we denote the body of the loop by S_0, the **while**-statement itself by S_1. Let (1.17′)–(1.25′) be the analogues of (1.17)–(1.25) for total rather than partial correctness; in those cases where $(1.n)$ is just a mathematical formula, not a specification, let $(1.n')$ be the same as $(1.n)$. The goal is to show (1.17′); this is deduced from (1.19′) in complete analogy to the proof of partial correctness. In view of remark 1.3, the assignment rule can be applied to $X := |X|$ in the form (1.26), since the only function at the right side is the absolute value-function, which is total, so the condition $|X|\!\downarrow$ always holds. More specifically, the precondition obtained from the postcondition ψ by (R2′) is $|X|\!\downarrow \,\&\, |X| = \gcd(x, y)$, which simplifies to $|X| = \gcd(x, y)$. Therefore $(1.23') = (1.23)$ is again sufficient to establish (1.22′).

Now we have to prove (1.19′), that is, the total correctness of the **while**-statement. We choose $|Y|$ as the expression E, since it was already noted that every execution of S_0 decreases its value. Also, we take ω as the set W, and the usual ordering $\leq$ as $\preceq$. In order to apply (R8′), we have to show:

$$\rho \,\&\, Y \neq 0 \Longrightarrow |Y| \in \omega, \tag{1.27}$$

$$\{\rho \,\&\, Y \neq 0 \,\&\, |Y| = e\} \quad S_0\!\downarrow \quad \{\rho \,\&\, |Y| < e\}. \tag{1.28}$$

Here (1.27) is trivial. For the third and second statement of S_0 we can again apply (1.26) by remark 1.3, which means that backward substitution can be applied as before. Only the assignment $Z := X \bmod Y$ requires to use (R2′) as it stands, since the function mod may be defined such that $x \bmod 0$ is undefined. Hence, working backwards from $\rho \,\&\, |Y| < e$, the following formulas are obtained:

$$\gcd(X, Z) = \gcd(x, y) \,\&\, |Z| < e,$$

$$\gcd(Y, Z) = \gcd(x, y) \,\&\, |Z| < e,$$

$$|X \bmod Y|\!\downarrow \,\&\, \gcd(Y, X \bmod Y) = \gcd(x, y) \,\&\, |X \bmod Y| < e.$$

In order to prove (1.28), it now remains to show the implication

$$\rho \mathbin{\&} Y \neq 0 \mathbin{\&} |Y| = e \Longrightarrow |X \bmod Y|\!\downarrow \mathbin{\&} \gcd(Y, X \bmod Y) = \gcd(x,y) \mathbin{\&} |X \bmod Y| < e. \tag{1.29}$$

Assume that the left-hand side of (1.29) holds. Then $|X \bmod Y|\!\downarrow$ follows, since $Y \neq 0$, $\gcd(Y, X \bmod Y) = \gcd(x,y)$ follows as in (1.24), and $|X \bmod Y| < e$ follows by (1.16). Thus (1.28) is established, and (R8′) allows to deduced (1.19′), completing the proof.

In this proof the condition $E \in W$ remained valid throughout, even upon exit from the loop. This is no longer the case when we prove total correctness in example 1.1, which we discuss briefly without going into details. Again we take $(\omega, \leq)$ as our well-ordering, and the expression E whose value is decreased by every execution of the loop is $N - S$. This time $N - S$ is negative when the execution of the **while**-statement is finished, but the first hypothesis in the application of (R8′), the implication $\rho \mathbin{\&} S \leq N \Longrightarrow E \in \omega$, still holds trivially. The second hypothesis to show is

$$\{\rho \mathbin{\&} S \leq N \mathbin{\&} E = e\} \quad P_1\!\downarrow \quad \{\rho \mathbin{\&} (E \prec e \vee S > N)\},$$

which is again straightforward. We used the notation $E \prec e$ rather than $E < e$ in order to emphasize that we apply the well-ordering which is defined only on ω. Thus if $E < 0$, then the condition $E \prec e$ fails, but in this case $S > N$ holds.

We claimed earlier that each termination proof for a **while**-statement can be carried out such that W is the whole data type of E. In the last example the data type of E is of course **integer**, so an appropriate well-founded ordering $\preceq$ on $\mathbb{Z}$, the set of integers, is asked for. It is defined by

$$x \preceq y \iff (x \leq y \mathbin{\&} y \geq 0) \vee x = y.$$

Thus $\preceq$ behaves on nonnegative numbers like $\leq$, negative numbers are smaller than nonnegative ones, and two negative numbers are incomparable with each other. Every $x < 0$ is a minimal element. It is easy to see that this $\preceq$ is indeed well-founded, and that each execution of P_1 decreases the value of E, so that

$$\{\rho \mathbin{\&} S \leq N \mathbin{\&} E = e\} \quad P_1\!\downarrow \quad \{\rho \mathbin{\&} E \preceq e\}$$

holds. This is due to the fact that the loop is executed only with $E \geq 0$.

We briefly consider some other examples of linear orderings. The common ordering $\leq$ is not a well-ordering on $\mathbb{Z}$ or on the set of nonnegative rational numbers, since the sets $\mathbb{Z}$ itself and $\{1, \frac{1}{2}, \frac{1}{3}, \ldots\}$ have no smallest elements. If A is finite, then every partial ordering on A is well-founded, so every linear ordering is a well-ordering. Let $(A, \preceq_1)$ and $(B, \preceq_2)$ be two well-orderings with $A \cap B = \emptyset$. Then a well-ordering $\preceq$ on $A \cup B$ can be defined by

$$x \preceq y \iff (x \in A \mathbin{\&} y \in A \mathbin{\&} x \preceq_1 y) \vee (x \in B \mathbin{\&} y \in B \mathbin{\&} x \preceq_2 y) \vee (x \in A \mathbin{\&} y \in B),$$

that is, in the ordering $\preceq$, elements of A come before elements of B. For instance, taking $A = \omega$ with the common ordering $\leq$, and $B = \{\infty\}$ with the trivial ordering, we obtain a well-ordering $\preceq$ on $\omega \cup \{\infty\}$, where ∞ is greater than all elements of ω. Without the condition $A \cap B = \emptyset$, another well-ordering $\preceq'$ can be defined on $A \times B$ (lexicographical ordering):

$$(x,y) \preceq' (x',y') \iff x \prec_1 x' \vee (x = x' \mathbin{\&} y \preceq_2 y').$$

In the examples in these lecture notes, we will always take $W = \omega$, with the ordinary well-ordering $\leq$. This choice also works often in more serious examples, as they occur in real life. An example where a more complicated well-ordering is needed is the algorithm for computing Gröbner basis for polynomial ideals.

1.4. Other control structures.

In this section we deal with three control structures not considered before, namely **case**-statements, **repeat**-**until**-loops and **for**-loops. It is rather obvious that they are unnecessary from a theoretical point of view, since **case**-statements can be replaced by nested conditionals, **repeat**-**until**- and **for**-loops by **while**-statements (even though in the latter case this transformation is not as trivial as it may appear). The syntax and semantics we use is essentially the one from PASCAL, which differs substantially from C, for instance.

A **case**-statement has the form

$$\textbf{case } E \textbf{ of } c_1\colon S_1; \ldots\ c_n\colon S_n \textbf{ end},$$

where E is an expression (**case**-selector), $c_1, \ldots, c_n$ are constants of the same data type as E, and $S_1, \ldots, S_n$ are statements. The interpretation is that S_i is executed if the value of E equals c_i; so a **case**-statement is like a conditional with multiple branching. We assume that no action takes place if the value of E does not belong to $\{c_1, \ldots, c_n\}$. In reality a run-time error may also happen in this case; this depends on the implementation of the language. We also allow an additional clause of the form **otherwise** S, which means that S is executed for $E \notin \{c_1, \ldots, c_n\}$. The following proof rules are straightforward analogues of (R6) and (R7), so no further comments are needed:

$$\text{(R9)} \qquad \frac{\{\phi \,\&\, E = c_1\} \quad S_1 \quad \{\psi\}, \ \ldots, \ \{\phi \,\&\, E = c_n\} \quad S_n \quad \{\psi\}, \qquad \{\phi \,\&\, E \notin \{c_1, \ldots, c_n\}\} \quad S \quad \{\psi\}}{\{\phi\} \quad \textbf{case } E \textbf{ of } c_1\colon S_1; \ldots\ c_n\colon S_n; \textbf{ otherwise } S \textbf{ end} \quad \{\psi\}}$$

$$\text{(R10)} \qquad \frac{\{\phi \,\&\, E = c_1\} \quad S_1 \quad \{\psi\}, \ \ldots, \ \{\phi \,\&\, E = c_n\} \quad S_n \quad \{\psi\}, \qquad \phi \,\&\, E \notin \{c_1, \ldots, c_n\} \Longrightarrow \psi}{\{\phi\} \quad \textbf{case } E \textbf{ of } c_1\colon S_1; \ldots\ c_n\colon S_n \textbf{ end} \quad \{\psi\}}$$

Corresponding proof rules (R9′), $(R10')$ for total correctness are obtained in the obvious way, analogously to (R1′), (R3′), …, (R7′).

A **repeat**-Until-statement is of the form

$$\textbf{repeat } S \textbf{ until } B.$$

It differs from a **while**-statement in two respects. For one thing, an exit from the loop occurs when B holds, not when B fails. More importantly, the loop condition B is tested after, not before every execution of S, and as a consequence, S is executed at least once. There exists apparently no reasonable proof rule for **repeat**-**until** with only one loop invariant; instead we need one invariant ϕ which holds before and another invariant ψ which holds after every execution of S. Two facts have to be shown: (i) If ϕ holds before some execution of S, then ψ holds afterwards, and (ii) If ψ holds after an execution of S and S is executed once more (meaning that $\neg B$ holds), then ϕ also holds before the next execution of S, that is, ϕ follows from $\psi \,\&\, \neg B$. Then a specification can be deduced analogously as in the **while**-rule.

$$\text{(R11)} \qquad \frac{\{\phi\} \quad S \quad \{\psi\},\ \psi \,\&\, \neg B \Longrightarrow \phi}{\{\phi\} \quad \textbf{repeat } S \textbf{ until } B \quad \{\psi \,\&\, B\}}$$

A rule for total correctness is obtained exactly as for **while**-statements. As in section 1.3 let E an appropriate expression whose value decreases with every execution of S, W a subset of the data type of E, and $\preceq$ a well-ordering on W. Then we have

$$\text{(R11}'\text{)} \qquad \frac{\{\phi \,\&\, E = e\} \quad S\!\downarrow \quad \{\psi \,\&\, (E \prec e \vee B)\},\ \psi \,\&\, \neg B \Longrightarrow \phi}{\{\phi\} \quad \textbf{repeat } S \textbf{ until } B \downarrow \quad \{\psi \,\&\, B\}}$$

Finally, we consider **for**-statements:

$$\textbf{for } I := E_1 \textbf{ to } E_2 \textbf{ do } S. \tag{1.30}$$

In this case the task of finding an appropriate proof rule is trickier, and there are several pitfalls to avoid. In general, when some construct causes trouble with respect to verification, then it is a good idea to rewrite the code so as to avoid it. In this way a proof rule for the construct may be derived, or at least verification may be carried out for any specific program where it occurs. Now at first glance it may appear that (1.30) is equivalent to the following program segment P, which uses a **while**-loop:

$I := E_1$;
while $I \leq E_2$ **do**
    begin S; $I := I + 1$ **end**

However, P is *not* necessarily equivalent to (1.30); do you see why? As noted above, we use the PASCAL-semantics of **for**-loops. This means in particular, that E_1 and E_2 are evaluated only once, and before each execution of S, I is compared with the *original* value of E_2. Hence the original values of E_1 and E_2 have to be kept in mathematical variables e_1, e_2.

Another difference between (1.30) and P is the following. Assume for simplicity that the execution of S does not alter the value of E_2, so that it remains e_2 throughout, and that the loop is executed at least once. Then the value of I after the execution of P will be $e_2 + 1$. But this is not necessarily the case after the execution of (1.30). Depending on (the implementation of) the underlying language, the value of I may also be undefined, or if e_2 happens to be the largest integer **maxint**, then the value may equal **maxint**. Actually, we will develop a proof rule for (1.30) where I does not occur in the postcondition, so no problem about the final value of I arises.

The loop invariant ρ will usually contain the control variable I. Since the dependence on I is of special importance, we write $\rho(I)$, and for any expression t, $\rho(t)$ denotes $\rho\langle t/I\rangle$. $\rho(e_1)$ is the condition which holds before the **for**-statement is executed, and in general, $\rho(i)$ holds before S is executed with i as the value of I. This value satisfies $e_1 \leq i \leq e_2$, and we have to show that $\rho(I+1)$ holds after the execution of S: $\{\rho(I) \,\&\, e_1 \leq I \leq e_2\}$ S $\{\rho(I+1)\}$. In the specification for (1.30) it is assumed that prior to execution the values of E_1 and E_2 are e_1 and e_2, and that $\rho(e_1)$ holds. What is the postcondition? If the last execution of S occurs for $I = e_2$, then $\rho(e_2)$ holds before and $\rho(e_2+1)$ after that execution. However, there is one case where $\rho(e_2+1)$ is not the condition which holds after the execution of (1.30), namely the one where $e_1 > e_2$, when no action takes place, and $\rho(e_1)$ continues to hold. The following proof rule takes both cases under consideration.

$$\text{(R12)}\qquad \frac{\{\rho(I)\ \&\ e_1 \le I \le e_2\}\quad S \quad \{\rho(I+1)\}}{\{E_1 = e_1\ \&\ E_2 = e_2\ \&\ \rho(e_1)\}\quad \textbf{for } I := E_1 \textbf{ to } E_2 \textbf{ do } S \quad \{\rho(\max(e_1, e_2+1))\}}$$

There also exists a statement analogous to (1.30), but with **downto** in place of **to**; in this case I is decreased rather than increased after every execution of S. A corresponding proof rule is obtained in full analogy to (R12):

$$\text{(R13)}\qquad \frac{\{\rho(I)\ \&\ e_2 \le I \le e_1\}\quad S \quad \{\rho(I-1)\}}{\{E_1 = e_1\ \&\ E_2 = e_2\ \&\ \rho(e_1)\}\quad \textbf{for } I := E_1 \textbf{ downto } E_2 \textbf{ do } S \quad \{\rho(\min(e_1, e_2-1))\}}$$

Going beyond PASCAL, we consider next a **for**-loop with arbitrary increment. In this case it is of course unnecessary to use the keyword **downto**. The increment is given by an expression E_3, whose value before execution is e_3. An exit from the loop occurs when the value of I is greater than or when it is smaller than e_2, depending on whether $e_3 \ge 0$ or $e_3 < 0$. For $e_3 = 0$ and $e_1 \le e_2$ we have the only case where the loop will not terminate. The condition before the loop is executed for $I = i$ is still denoted by $\rho(i)$, and after that execution it must be $\rho(i + e_3)$. The clause $\mathrm{betw}(e_1, e_2, e_3, I)$ asserts that I has one of the values for which the loop is executed, see below. The postcondition for the **for**-statement is of the form $\rho(e)$ for an appropriate value e which depends only on e_1, e_2 and e_3, and which we write as $\mathrm{lv}(e_1, e_2, e_3)$. Its determination is slightly tricky and will be done subsequently. The proof rule itself is

$$\text{(R14)}\qquad \frac{\{\rho(I)\ \&\ \mathrm{betw}(e_1, e_2, e_3, I)\}\quad S \quad \{\rho(I+e_3)\}}{\{E_1 = e_1\ \&\ E_2 = e_2\ \&\ \rho(e_1)\}\quad \textbf{for } I := E_1 \textbf{ to } E_2 \textbf{ step } E_3 \textbf{ do } S \quad \{\rho(\mathrm{lv}(e_1, e_2, e_3))\}}$$

It remains to provide the correct definitions for betw and lv. For instance, if $e_1 = 3$, $e_2 = 14$ and $e_3 = 5$, then the loop will be executed for $I = 3$, 8 and 13, and an exit occurs with $I = 18$. If instead $e_2 = -4$, then no execution takes place, whereas for $e_2 = -4$ and $e_3 = -5$ the loop will be executed for $I = 3$ and -2, with an exit for $I = -7$. In general, all values of I satisfy $I \equiv e_1 \bmod e_3$. These examples should illustrate clearly enough, how the definitions should be. First, we have

$$\begin{aligned}\mathrm{betw}(e_1, e_2, e_3, I) \iff I \equiv e_1 \pmod{e_3}\\ \&\ (e_3 \ge 0 \Longrightarrow e_1 \le I \le e_2)\ \&\ (e_3 < 0 \Longrightarrow e_2 \le I \le e_1).\end{aligned}$$

The value $\mathrm{lv}(e_1, e_2, e_3)$ is easier to describe in English. For $e_3 > 0$ ($e_3 < 0$), $\mathrm{lv}(e_1, e_2, e_3)$ is the smallest $e > e_2$ or the largest $e < e_2$ respectively such that $e \equiv e_1 \bmod e_3$. For $e_3 = 0$ we put $\mathrm{lv}(e_1, e_2, e_3) = e_1$.

Finally, we present corresponding rules (R12′)–(R14′) for total correctness. Loops with nonzero increment can never cause nontermination, hence (R12′) and (R13′) are obtained simply by adding $\downarrow$ in the hypothesis and in the conclusion. In (R14′) we have to take care of the one case where a **for**-loop may not terminate, which was already pointed out. Thus we have

$$\text{(R14}'\text{)}\qquad \frac{\{\rho(I)\ \&\ \mathrm{betw}(e_1, e_2, e_3, I)\}\quad S\!\downarrow \quad \{\rho(I+e_3)\}}{\begin{array}{l}\{E_1 = e_1\ \&\ E_2 = e_2\ \&\ \rho(e_1)\ \&\ (e_3 \ne 0 \lor e_1 > e_2)\}\\ \qquad\qquad \textbf{for } I := E_1 \textbf{ to } E_2 \textbf{ step } E_3 \textbf{ do } S\downarrow \quad \{\rho(\mathrm{lv}(e_1, e_2, e_3))\}\end{array}}$$

Example 1.3: Suppose we wish to prove

$$\{N \geq 0\} \quad P \quad \{S = N(N+1)/2\}, \tag{1.31}$$

where P is the following program segment:

$S := 0;$
for $I := 1$ **to** N **do**
    $S := S + I$

We continue to present toy examples of little practical use, since the purpose is just to illustrate how proofs of specifications can be constructed by means of the proof rules. The current example is unpractical for two reasons. For one thing, P is a very inefficient way to compute the sum of the first N numbers. Secondly, the specification should indicate that upon termination, $S = n(n+1)/2$, where n is the original value of N, see the comment on example 0.1. However, we give (1.31) in the above form in order to add a tiny extra twist to the problem.

What is the loop invariant $\rho(I)$? Before the assignment $S := S + I$ is carried out for $I = i$, the value of S is $1 + \cdots + (i-1)$, which is $i(i-1)/2$. However, a loop invariant of the form $S = I(I-1)/2$ is not quite sufficient. The rule then yields $S = n'(n'-1)/2$ as a postcondition for the **for**-statement, where n is the original value of N, which plays the rôle of e_2, and $n' = \max(1, n+1)$. In order to get $S = N(N+1)/2$ as a postcondition, we have to include the clause $N = n$ in the loop invariant, and we also have to include $n \geq 0$, so that n' simplifies to $n+1$; this is the extra twist. Thus $\rho(I)$ is the formula $N = n \geq 0 \,\&\, S = I(I-1)/2$.

The mathematical variables e_1, e_2 in (R12) are free variables, and the specifications are understood to hold for *all* values of e_1 and e_2. In the current example we substitute 1 for e_1, of course. According to (R12), we have to show:

$$\{N = n \geq 0 \,\&\, S = I(I-1)/2 \,\&\, 1 \leq I \leq n\} \quad S := S + I \quad \{N = n \geq 0 \,\&\, S = (I+1)I/2\}. \tag{1.32}$$

Backward substitution from the postcondition yields $N = n \geq 0 \,\&\, S + I = (I+1)I/2$, which is equivalent to the precondition (since $(I+1)I/2 - I = I(I-1)/2$). Thus (1.32) is established, and (R12) (applied literally) yields:

$$\{1 = 1 \,\&\, N = n \,\&\, N = n \geq 0 \,\&\, S = 1 \cdot 0/2\} \quad \textbf{for} \ \ldots \quad \{N = n \geq 0 \,\&\, S = n'(n'-1)/2\}, \tag{1.33}$$

where n' is as above. The postcondition simplifies to $N = n \geq 0 \,\&\, S = n(n+1)/2$, which implies $S = N(N+1)/2$. The precondition for the **for**-statement simplifies to $N = n \geq 0 \,\&\, S = 0$, and backward substitution with respect to the assignment $S := 0$ yields $N = n \geq 0 \wedge 0 = 0$. Thus we get:

$$\{N = n \geq 0\} \quad P \quad \{S = N(N+1)/2\}. \tag{1.34}$$

Finally, (1.31) is obtained by substituting the original value of N for the free variable n, whereupon the clause $N = n$ disappears. The same proof also works for total rather than partial correctness.

The last step in the deduction of (1.31) illustrates a general rule which was not mentioned before. Let v be a free mathematical variable in a specification $\{\phi\}\ S\ \{\psi\}$.

Then we may substitute any *mathematical* expression for v—this is just an instance of a general rule from mathematical logic, which allows to deduce $\phi\langle t/v\rangle$ from ϕ for any formula ϕ, variable v and term t. By contrast, a *program* expression E, that is, an expression containing program variables, may not be substituted for v in general, since the value of E may change in the course of the execution of S. For instance, $\{N = n\}\ S\ \{N = n^2\}$ cannot be replaced by an equivalent specification without a mathematical variable like n. On the other hand, the substitution of E for v is allowed if v occurs only in the pre- or only in the postcondition. The interpretation is that the value substituted for v is the value of E before (or after) the execution of S.

2. A mathematical basis for the proof rules

In chapter 1 the justification for the proof rules was somewhat informal, although it should be convincing enough. In the present chapter we indicate how they can be shown mathematically, though not all details will be given. For this purpose the meaning of specifications must be defined more rigorously, which is tantamount to defining the semantics of the underlying language, the topic of section 2.1. In section 2.2 we will show certain results which strengthen the proof rules. For instance, in the assignment rule $\{\psi\langle A/X\rangle\}\ X := A\ \{\psi\}$, the precondition $\psi\langle A/X\rangle$ is the *weakest* condition ϕ for which $\{\phi\}\ X := A\ \{\psi\}$ holds; every other such ϕ satisfies $\phi \Longrightarrow \psi\langle A/X\rangle$. This kind of investigation is not so easy to perform without a mathematical foundation for program correctness.

2.1. The semantics of programs.

Recall that the program segments in the specifications to be considered do not include declarations, and that the type of each variable is assumed to be known. Of course it must also be known in the first place whether any given identifier denotes a variable, constant or function etc., and what the value of each constant is. Accordingly, let $\mathcal{C}$ be the set of all constants, $\mathcal{V}$ the set of all variables, $\mathcal{C}(T)$ and $\mathcal{V}(T)$ the set of all constants or variables respectively of type T. Also, $\mathcal{F}$ is the set of all functions, and $\mathcal{F}(T_1, \ldots, T_n, T)$ the set of functions where $T_1, \ldots, T_n$ are the types of the arguments and T the type of the result. Binary symbols like $+$ are also included in $\mathcal{F}$, and when we speak of expressions of the form $F(A_1, \ldots, A_n)$, which are obtained by the application of a function, then it is understood that expressions like $A_1 + A_2$ are also included.

In general we assume that if X is a constant- or function symbol, then the symbol X also denotes the same constant or function in mathematical formulas. For instance, the constant 5 represents the same object both in programs and in mathematical writing, of course. The same is true if X is a function symbol like $+$ or sin, provided that the programs are executed on an idealized (mathematical) computer. When such a computer has to evaluate the expression $X + Y$, then it really computes the mathematical value $x + y$, where x and y, the values of X and Y, may be arbitrary integers (or real numbers, depending on the data type of X and Y). This is what we assume for the most part in these lecture notes.

In section 3.1, however, we will briefly discuss the verification of programs on *real* computers, where overflow, underflow and rounding errors may occur. In this case the symbol $+$ in a computer program represents a function $\tilde{+}$ which differs from the mathematical operation $+$, and the assignment rule has to be adjusted accordingly. Of course, the same is true for other functions.

$+$ and sin are examples of function symbols whose meaning in mathematical writing is standard. According to our general assumption, we allow the use of any such symbol in our sample programs, even if it is not allowed in real programming languages. For instance, the assignment $X := Y \cup Z$ is all right, if X, Y and Z are of a **set** data type. Conversely, if a constant- or function symbol X is declared by the user, then it is understood that common mathematical notation is extended by the symbol X, and that the meaning of X is the one specified by the declaration. Thus if the program contains the declaration

const C1=5,

then formulas may use C1 as an alternative symbol for 5. For functions, this convention is more relevant, of course.

If T is a type, then we write $\tilde{T}$ for the set of values of T. It would be possible to make a similar convention as for constants and functions and stipulate that T represents the object $\tilde{T}$ in programs and in mathematical formulas. However, such a convention appears less practical, since it does not correspond to common notation. For instance, if T is the type **integer**, then $\tilde{T} = \mathbb{Z}$.

The following is the key definition of this section:

Definition 2.1: An *environment* is a function E with domain $\mathcal{V}$ such that for every $V \in \mathcal{V}(T)$, $E(V) \in \tilde{T}$. (Intuitively, E specifies a system of values of the variables, as it occurs at any stage during a computation.) The set of all environments is denoted by $\mathcal{E}$.

We are now in a position to define the semantics $M(X)$ of terms and program segments X. For a term A, $M(A)$ is the value of A in dependence of the environment, therefore $M(A)$ is a mapping from $\mathcal{E}$ to $\tilde{T}$, where T is the type of A. We write $M(A, E)$ for $M(A)(E)$, the value of A under environment E. If A contains non-total functions, then $M(A, E)$ may be undefined; we write $M(A, E)\downarrow$ and $M(A, E)\uparrow$ for '$M(A, E)$ is defined' and '$M(A, E)$ is undefined' respectively. The formal definition follows; every term has one of the forms considered under (i)–(iv).

Definition 2.2: The semantics $M(A)$ of a term A is given recursively as follows:

(i) If C is a constant, then $M(C, E) = C$ for every $E \in \mathcal{E}$.

(ii) If V is a variable, then $M(V, E) = E(V)$, the value of V under environment E.

(iii) If A is of the form $F(A_1, \ldots, A_n)$, then $M(A, E) = F(M(A_1, E), \ldots, M(A_n, E))$; this value may also be undefined.

(iv) If A is of the form **if** B **then** A_1 **else** A_2, then

$$M(A, E) = \begin{cases} M(A_1, E) & \text{if } M(B, E) = \textbf{true}, \\ M(A_2, E) & \text{if } M(B, E) = \textbf{false}. \end{cases}$$

In analogy to remark 1.1, an application of part (iii) or (iv) of this definition is correct only if it corresponds to the parsing of the term involved. Assume, for instance, that expressions are evaluated left to right, so that the term $A = X - Y + Z$ is the same as $(X - Y) + Z$. Then of course it would be wrong to apply (iii) in the form $M(A, E) = M(A_1, E) - M(A_2, E)$ with $A_1 = X$ and $A_2 = Y + Z$; indeed, the correct substitution of X and $Y + Z$ for A_1 and A_2 in $A_1 - A_2$ would be $X - (Y + Z)$. The correct way to apply (iv) is obvious. The parsing of programs is not a topic in these lecture notes, but software systems for program verification have to deal with it.

The semantics of a program segment P specifies how the environment after depends on the environment before execution of P. Accordingly, $M(P)$ is a partial function from $\mathcal{E}$ to $\mathcal{E}$. Analogously as for terms, we write $M(P, E)$ for $M(P)(E)$, and $M(P, E, V)$ for $M(P)(E)(V)$. Note that $M(P)(E)$, if defined, is an environment E', so that $E'(V)$ is to be specified for each variable V. $M(P, E)$ is undefined, if P does not terminate in the case where E is the environment before execution.

Definition 2.3: Let P be a program which contains no other constructs than assignments, compound statements, conditionals and **while**-loops. The semantics of P is defined recursively as follows:

(i) $M(\Lambda, E) = E$.

(ii) If $M(A,E)\uparrow$, then $M(X := A, E)\uparrow$, otherwise

$$M(X := A, E, V) = \begin{cases} M(A,E) & \text{for } V = X, \\ E(V) & \text{for } V \neq X, \end{cases}$$

(iii) $M(P_1; P_2, E) = M(P_2, M(P_1, E))$.

(iv) $M(\textbf{begin } P_1 \textbf{ end}) = M(P_1)$;

(v)

$$M(\textbf{if } B \textbf{ then } P_1 \textbf{ else } P_2, E) = \begin{cases} M(P_1,E) & \text{if } M(B,E) = \textbf{true}, \\ M(P_2,E) & \text{if } M(B,E) = \textbf{false}. \end{cases}$$

(vi)

$$M(\textbf{if } B \textbf{ then } P_1) = \begin{cases} M(P_1,E) & \text{if } M(B,E) = \textbf{true}, \\ E & \text{if } M(B,E) = \textbf{false}. \end{cases}$$

(vii) $M(\textbf{while } B \textbf{ do } P, E) = \big(M(P)\big)^n(E)$, where $n \geq 0$ is minimal such that $M\big(B, (M(P))^n(E)\big) = \textbf{false}$; if no such n exists, then $M(\textbf{while } B \textbf{ do } P, E)\uparrow$. (Thus $M(\textbf{while } B \textbf{ do } P, E)$ is obtained from E by applying the mapping $M(P)$ repeatedly, until the value of B under the resulting environment is **false**.)

In this definition, and also in theorem 2.2 it is assumed for simplicity that the value of the Boolean expression B in **if**- and **while**-statements is defined whenever it has to be evaluated; this will be the case in all examples. Without this assumption the development of the theory has to be modified somewhat; this is straightforward but involves some extra work.

The definition of the semantics of **case**-, **repeat**-Until- and **for**-statements is left as an exercise for the reader. As noted in section 1.4, these control structures are less fundamental than **if** and **while**. Moreover, our goal is to describe the approach towards defining the semantics of programs, rather than giving all details.

There is an alternative, more elegant way to specify the semantics of **while**-loops, namely

$$M(\textbf{while } B \textbf{ do } P, E) = \begin{cases} E & \text{if } M(B,E) = \textbf{false}, \\ M\big(\textbf{while } B \textbf{ do } P, M(P,E)\big) & \text{if } M(B,E) = \textbf{true}. \end{cases} \tag{2.1}$$

This is a *recursive* definition of the function $H = M(\textbf{while } B \textbf{ do } P)$, and some remarks are in order concerning its precise meaning. The following example should clarify how recursive definitions of functions must be understood in general.

Example 2.1: Let

$$f(x) = \begin{cases} 0 & \text{if } x = 0, \\ f(x-2) + 2 & \text{if } x \neq 0. \end{cases} \tag{2.2}$$

We assume that the data type of the argument of f is **integer**, which is our default assumption. This does not mean, though, that $\mathrm{dom}(f) = \mathbb{Z}$; in general, recursive definitions specify *partial* functions. Is there a *unique* function f satisfying (2.2)? Certainly not. Regarding (2.2) as a functional equation, each of the following functions is a solution:

$$f_1(x) = x \qquad \text{(for all } x\text{);}$$

$$f_2(x) = \begin{cases} x & \text{if } x \geq 0 \text{ and } x \text{ is even,} \\ x - 3 & \text{if } x < 0 \text{ and } x \text{ is even,} \\ \uparrow & \text{if } x \text{ is odd;} \end{cases}$$

$$f_3(x) = \begin{cases} x & \text{if } x \geq 0 \text{ and } x \text{ is even,} \\ \uparrow & \text{if } x < 0 \text{ and } x \text{ is even,} \\ x + 2 & \text{if } x \text{ is odd;} \end{cases}$$

$$f_4(x) = \begin{cases} x & \text{if } x \geq 0 \text{ and } x \text{ is even,} \\ \uparrow & \text{otherwise.} \end{cases}$$

Which function do we really mean to define by (2.2)? The answer is f_4, since this is the *minimal* solution of the functional equation. This means that every other solution f is an extension of f_4. Also, f_4 is the function computed if (2.2) were translated into a recursive function declaration in a program. In this case the computation of $f(x)$ terminates only if x is even and nonnegative.

Notation: $g \subseteq g'$ means that g' is an extension of g, that is, $g'(x) = y$ holds whenever $g(x) = y$. (But $g'(x) = y$ may also hold if $g(x)\uparrow$.) Equivalently, g is a subset of g' if functions are regarded as sets of pairs.

In general, every recursive definition of a function f can be written in the form

$$f = \Phi(f), \tag{2.3}$$

where Φ is a function operator with certain properties, see below. In example 2.1, Φ maps f to the right side of (2.2), regarded as a function of x. The solutions of (2.3) are the fixpoints of Φ, but the minimal fixpoint is the function defined by (2.3). The existence of a minimal fixpoint can be derived from a general result of set theory, which has an analogue in computability theory. An operator Φ has a minimal fixpoint whenever it satisfies the following two properties, which are met if Φ is associated with a recursive definition:

(i) Φ is *monotonous*, that is, $F \subseteq G$ implies $\Phi(F) \subseteq \Phi(G)$.
(ii) Φ is *continuous*, that is, if $F_0 \subseteq F_1 \subseteq \cdots$, then $\Phi(\bigcup_{i\in\omega} F_i) = \bigcup_{i\in\omega} \Phi(F_i)$.

So this is how the recursive definition (2.1) must be understood. The equivalence of (2.1) with definition 2.3 (vii) follows immediately.

Finally, we are in a position to define the meaning of specifications in a mathematically precise way. The intuition should be clear.

Definition 2.4: (i) Let ϕ be a mathematical formula which may contain program variables and other (mathematical) variables. Then the assertion $\phi[E]$ (ϕ holds under environment E) means that ϕ holds with $E(V)$ substituted for V for every program variable V. $\phi[E]$ should be regarded as a mathematical formula where E itself, rather than the program variables, occurs free.

(ii) $\{\phi\}\ S\ \{\psi\}$ asserts $(\forall E \in \mathcal{E})(\phi[E]\ \&\ M(S,E)\!\downarrow\ \Longrightarrow \psi[M(S,E)])$. (In this formula only the mathematical variables of ϕ and ψ may occur free, similarly for (iii).)

(iii) $\{\phi\}\ S\!\downarrow\ \{\psi\}$ asserts $(\forall E \in \mathcal{E})(\phi[E] \Longrightarrow M(S,E)\!\downarrow\ \&\ \psi[M(S,E)])$.

2.2. Weakest preconditions and strongest postconditions.

In this section we have to deal with assertions about mathematical formulas as they occur in mathematical logic, so the following notation will be needed.

Notation: For a formula ϕ, $\models \phi$ means that ϕ is valid. Equivalently, $(\forall x_1, \ldots, x_n)(\phi)$ is a true sentence, where $x_1, \ldots, x_n$ are the free variables of ϕ. $\phi \equiv \psi$ means that ϕ and ψ are logically equivalent, that is, $\models (\phi \iff \psi)$.

Definition 2.5: (a) A formula ϕ is a *weakest precondition* for a program segment S and a postcondition ψ ($\phi \equiv S \to \psi$) iff

(i) $\models [\{\phi\}\ S\ \{\psi\}]$, and

(ii) For every formula ρ, if $\models [\{\rho\}\ S\ \{\psi\}]$, then $\models [\rho \Longrightarrow \phi]$.

(a) A formula ψ is a *strongest postcondition* for a program segment S and a precondition ϕ ($\psi \equiv \phi \leftarrow S$) iff

(i) $\models [\{\phi\}\ S\ \{\psi\}]$, and

(ii) For every formula ρ, if $[\models \{\phi\}\ S\ \{\rho\}]$, then $\models [\psi \Longrightarrow \rho]$.

Obviously, weakest preconditions, if they exist, are uniquely determined up to logical equivalence, and this is why we use the notation $\phi \equiv S \to \psi$ ('ϕ is equivalent to any weakest precondition for program segment S and postcondition ψ.') Similarly for strongest postconditions. As a matter of fact, weakest preconditions and strongest postconditions always exist, but we will not provide a complete proof.

The next task is to develop weakest preconditions and strongest postconditions for specific constructs. This will be done informally first; then the results will be summarized in a theorem and shown rigorously. It is obvious that the rule for the empty statement yields both a weakest precondition and a strongest postcondition for Λ. Also, the informal justification for the assignment rule from section 1.1 can be extended to show that (R2) yields a weakest precondition for assignment statements, provided that all functions in the expression assigned are total. If not, then $\{\phi\}\ X := A\ \{\psi\}$ holds iff ϕ implies $A\uparrow \vee \psi\langle A/X\rangle$. Indeed, if $A\uparrow$ prior to the assignment, then the execution of the assignment does not terminate, and the implication defining $\{\phi\}\ X := A\ \{\psi\}$ holds.

How about the strongest postcondition $\psi \equiv \phi \leftarrow X := A$? As an example, assume that A is the expression $X^3 + Y$, and the precondition ϕ is $X \geq 5$. If ϕ holds before the assignment $X := X^3 + Y$ is executed, what condition holds afterwards? Obviously, the new value of X must be of the form $x^3 + y$, where x is the old value of X, and y is the old and new value of Y. Here y can still be represented as Y in the postcondition ψ, but the old value of X is no longer available; what can be expressed is that there *exists* a value x such that the new value of X is $x^3 + Y$. In addition, this x must satisfy the precondition ϕ in place of X, so $x \geq 5$ must hold in our example. So the right postcondition $(\exists x)(x \geq 5\ \&\ X = x^3 + Y)$ in the example, and $(\exists x)(\phi\langle x/X\rangle\ \&\ X = A\langle x/X\rangle)$ in general.

It goes without saying that the newly introduced mathematical variable x must not be a variable which already occurs free in ϕ, so a different variable has to be chosen if necessary. It is all right to use x if x occurs bound in ϕ, but then the substitution may involve a renaming of bound variables, as is well-known from mathematical logic. See example 2.2.

Sequences of statements are easy to deal with. Let ϕ be a given precondition for $S_1; S_2$. Then $\rho \equiv (\phi \leftarrow S_1)$ is the strongest condition known to hold after execution of S_1, whereupon $\rho \leftarrow S_2$ is the strongest condition known to hold after execution of S_2; hence $(\phi \leftarrow S_1; S_2) \equiv ((\phi \leftarrow S_1) \leftarrow S_2)$. Weakest preconditions for statement sequences are obtained similarly. Here and in the sequel, weakest preconditions and strongest postconditions for nonatomic statements are expressed in terms of corresponding conditions for their constituents. It is also clear in view of (R5) that the weakest precondition and strongest postcondition for **begin** S **end** are the same as for S.

Finally, consider a conditional **if** B **then** S **else** S'. If ϕ holds before execution, what condition holds afterwards? If the **then**-clause applies, then $\phi \,\&\, B$ holds before, and $(\phi \& B) \leftarrow S$ after execution of S; otherwise $\phi \& \neg B$ holds before, and $(\phi \& \neg B) \leftarrow S'$ after execution of S'. Accordingly, the postcondition for the conditional asserts that one of the two postconditions (for S or S') holds.

Conversely, let a postcondition ψ be given. The validity of ψ after execution can be guaranteed in two ways. Prior to execution either B must hold along with the precondition $S \rightarrow \psi$, so that S is executed and ψ holds afterwards, or, analogously, $\neg B \,\&\, (S' \rightarrow \psi)$ must hold.

These arguments will be formalized in the proof of theorem 2.1. As a preparation we need

Lemma 2.1: (i) If $\phi \equiv S \rightarrow \psi$, then

$$\models \big[(\forall E \in \mathcal{E})\big(M(S,E)\!\uparrow \vee\, \psi[M(S,E)] \Longrightarrow \phi[E]\big)\big].$$

(ii) If $\psi \equiv \phi \leftarrow S$, then

$$\models \big[(\forall E \in \mathcal{E})\big(\psi[E] \Longrightarrow (\exists E' \in \mathcal{E})(\phi[E'] \,\&\, M(S,E')\!\downarrow = E)\big)\big].$$

Proof: Let $X_1, \ldots, X_m$ be the program variables, $z_1, \ldots, z_n$ the mathematical variables occurring in ϕ, ψ and S.

(i) Suppose that the assertion fails. Then there exist an environment E and values $\overline{z}_1, \ldots, \overline{z}_n$ such that $M(S,E)\!\uparrow$ or $\psi[M(S,E)]$ holds but $\phi[E]$ fails for $z_1 = \overline{z}_1, \ldots, z_n = \overline{z}_n$. The same is true for every $E' \in \mathcal{E}$ such that $E'(X_i) = E(X_i)$ for $i = 1, \ldots, m$; the values of other variables do not affect and are not affected by the execution of S. Putting $E(X_i) = x_i$, it follows that the formula

$$\phi': \quad X_1 = x_1 \,\&\, \cdots \,\&\, X_m = x_m \,\&\, z_1 = \overline{z}_1 \,\&\, \cdots \,\&\, z_n = \overline{z}_n$$

also satisfies $\models [\{\phi'\}\ S\ \{\psi\}]$, but does not satisfy $\models [\phi' \Longrightarrow \phi]$. This contradicts the assumption $\phi \equiv S \rightarrow \psi$.

(ii) Again suppose that the assertion fails. Then there exist an environment E and values $\overline{z}_1, \ldots, \overline{z}_n$ such that $\psi[E]$ holds but no $E' \in \mathcal{E}$ satisfies $\phi[E'] \,\&\, M(S,E')\!\downarrow = E$ for $z_i = \overline{z}_i$. Let ψ' be the negation of the formula ϕ' from the proof of (i). Then ψ' also satisfies $\models [\{\phi\}\ S\ \{\psi'\}]$, but does not satisfy $\models [\psi \Longrightarrow \psi']$. This contradicts the assumption $\psi \equiv \phi \leftarrow S$. $\square$

Theorem 2.1: (i) $\Lambda \to \psi \equiv \psi$.

(ii) $\phi \leftarrow \Lambda \equiv \phi$.

(iii) $(X := A \to \psi) \equiv A\uparrow \vee \psi\langle A/X\rangle$.

(iv) $(\phi \leftarrow X := A) \equiv (\exists x)(\phi\langle x/X\rangle \;\&\; X = A\langle x/X\rangle)$, where x is a (mathematical) variable which does not occur free in ϕ.

(v) $(S_1; S_2 \to \psi) \equiv S_1 \to (S_2 \to \psi)$. (Since we do not prove that weakest preconditions and strongest postconditions exist in general, it should be clarified how (v) has to be understood if $S_2 \to \psi$ or $S_1 \to (S_2 \to \psi)$ does not exist. In this case no claim is made about the form or existence of $S_1; S_2 \to \psi$. Similarly in the remaining parts of the theorem.)

(vi) $(\phi \leftarrow S_1; S_2) \equiv (\phi \leftarrow S_1) \leftarrow S_2$.

(vii) $(\textbf{begin } S \textbf{ end} \to \psi) \equiv S \to \psi$.

(viii) $(\phi \leftarrow \textbf{begin } S \textbf{ end}) \equiv \phi \leftarrow S$.

(ix) $(\textbf{if } B \textbf{ then } S \textbf{ else } S' \to \psi) \equiv (B \,\&\, (S \to \psi)) \vee (\neg B \,\&\, (S' \to \psi))$.

(x) $(\phi \leftarrow \textbf{if } B \textbf{ then } S \textbf{ else } S') \equiv ((\phi \,\&\, B) \leftarrow S) \vee ((\phi \,\&\, \neg B) \leftarrow S')$.

(xi) $(\textbf{if } B \textbf{ then } S \to \psi) \equiv (B \,\&\, (S \to \psi)) \vee (\neg B \,\&\, \psi)$.

(xii) $(\phi \leftarrow \textbf{if } B \textbf{ then } S) \equiv ((\phi \,\&\, B) \leftarrow S) \vee (\phi \,\&\, \neg B)$.

Proof: (i), (ii) These two parts are immediate consequences of the following fact: for any two formulas ϕ, ψ,

$$\models [\{\phi\} \quad \Lambda \quad \{\psi\}] \quad \text{if and only if} \quad \models [\phi \Longrightarrow \psi]. \tag{2.4}$$

By definition, $\models [\{\phi\}\ \Lambda\ \{\psi\}]$ means

$$\models \big[(\forall E \in \mathcal{E})(\phi[E] \,\&\, M(\Lambda, E)\!\downarrow \Longrightarrow \psi[M(\Lambda, E)])\big];$$

since $M(\Lambda, E)\!\downarrow = E$, this assertion simplifies to $\models \big[(\forall E \in \mathcal{E})(\phi[E] \Longrightarrow \psi[E])\big]$, which is clearly equivalent to $\models [\phi \Longrightarrow \psi]$. Thus (2.4) is established.

(iii) It suffices to show that for every formula ϕ,

$$\models [\{\phi\} \quad X := A \quad \{\psi\}] \quad \text{if and only if} \quad \models [\phi \Longrightarrow A\uparrow \vee \psi\langle A/X\rangle]. \tag{2.5}$$

Write X_0 for X, and let $X_1, \ldots, X_n$ be the remaining program variables occurring in ϕ, ψ or A. Thus we may write $\phi(X_0, \ldots, X_n)$ for ϕ, and similarly for ψ and A. Let E be an environment and $E(X_i) = x_i$. If $A(x_1, \ldots, x_n)\!\downarrow$, then $M(X := A, E, X_i) = x_i$ for $i \geq 1$, and $M(X := A, E, X_0) = M(A, E)$. Induction on the length of E shows immediately that $M(A, E)$ is given by the mathematical expression $A(x_0, \ldots, x_n)$. Hence we have

$$\begin{aligned}
&\models [\{\phi\} \quad X := A \quad \{\psi\}] \\
&\quad\Longleftrightarrow \models \big[(\forall E \in \mathcal{E})\big(\phi[E] \,\&\, M(X := A, E)\!\downarrow \Longrightarrow \psi[M(X := A, E)]\big)\big] \\
&\quad\Longleftrightarrow \models \big[(\forall x_0, x_1, \ldots, x_n)\big(\phi(x_0, \ldots, x_n) \,\&\, A(x_0, \ldots, x_n)\!\downarrow \\
&\qquad\qquad\qquad\qquad \Longrightarrow \psi(A(x_0, \ldots, x_n), x_1, \ldots, x_n)\big)\big] \\
&\quad\Longleftrightarrow \models \big[\phi(X_0, \ldots, X_n) \,\&\, A(X_0, \ldots, X_n)\!\downarrow \Longrightarrow \psi(A(X_0, \ldots, X_n), X_1, \ldots, X_n)\big] \\
&\quad\Longleftrightarrow \models [A\uparrow \vee (\phi \Longrightarrow \psi\langle A/X\rangle)],
\end{aligned}$$

as claimed.

(iv) We have to show that for every formula ψ,

$$\models [\{\phi\} \quad X := A \quad \{\psi\}] \quad \text{if and only if} \quad \models [(\exists x)(\phi\langle x/X\rangle \ \& \ X = A\langle x/X\rangle) \Longrightarrow \psi],$$

or, by (2.5),

$$\models \big[\phi \Longrightarrow A\uparrow \vee \psi\langle A/X\rangle\big] \quad \text{if and only if} \quad \models \big[(\exists x)(\phi\langle x/X\rangle \ \& \ X = A\langle x/X\rangle) \Longrightarrow \psi\big]. \tag{2.6}$$

To simplify notation, write $\phi(X)$, $\psi(X)$ and $A(X)$ for ϕ, ψ and A, so that $\phi\langle x/X\rangle$ can be written $\phi(x)$ etc. Thus the left side of (2.6) becomes

$$\models [\phi(X) \Longrightarrow A(X)\uparrow \vee \psi(A(X))], \tag{2.7}$$

and the right side becomes

$$\models \big[(\exists x)(\phi(x) \ \& \ X = A(x)) \Longrightarrow \psi(X)\big]. \tag{2.8}$$

First we show (2.8), assuming (2.7). Let x be such that $\phi(x) \ \& \ X = A(x)$ holds. Applying (2.7) with x in the rôle of X, we deduce $(A(x)\uparrow \vee \psi(A(x))) \ \& \ X = A(x)$, which implies $\psi(X)$. (Note that $X = A(x)$ implies $A(x)\downarrow$.) Thus (2.8) is established. Conversely, assume (2.8). In order to establish (2.7), assume $\phi(X)$. If $A(X)\uparrow$, then (2.7) follows. Otherwise apply (2.8) with $A(X)$ in place of X. Then the hypothesis $(\exists x)(\phi(x) \ \& \ A(X) = A(x))$ holds with $x = X$, hence the conclusion $\psi(A(X))$ holds, proving again (2.7). Thus (2.7) and (2.8) are indeed equivalent, as claimed.

(v) Let $\Phi_i = M(S_i)$ for $i = 1, 2$, $\rho \equiv S_2 \to \psi$, $\phi \equiv S_1 \to \rho$. In order to establish $\phi \equiv S_1; S_2 \to \psi$, we have to show that

$$\models [\{\phi\} \quad S_1; S_2 \quad \{\psi\}] \tag{2.9}$$

holds and that for every formula ϕ',

$$\models [\{\phi'\} \quad S_1; S_2 \quad \{\psi\}] \Longrightarrow \models [\phi' \Longrightarrow \phi]. \tag{2.10}$$

By definition of ρ and ϕ, the following implications are valid for all $E \in \mathcal{E}$:

$$\begin{aligned} \phi[E] \ \& \ M(S_1; S_2, E)\downarrow &\Longrightarrow \Phi_2\Phi_1(E)\downarrow \ \& \ \rho[\Phi_1(E)] \\ &\Longrightarrow \psi[\Phi_1\Phi_2(E)] \\ &\Longleftrightarrow \psi[M(S_1; S_2, E)]; \end{aligned}$$

this shows (2.9).

For (2.10), assume that $\models [\{\phi'\} \ S_1; S_2 \ \{\psi\}]$, that is

$$\models \big[(\forall E \in \mathcal{E})\big(\phi'[E] \ \& \ \Phi_2\Phi_1(E)\downarrow \Longrightarrow \psi[\Phi_2\Phi_1(E)]\big)\big]. \tag{2.11}$$

We claim that $\models [\{\phi'\} \ S_1 \ \{\rho\}]$ holds, that is,

$$\models \big[(\forall E \in \mathcal{E})\big(\phi'[E] \ \& \ \Phi_1(E)\downarrow \Longrightarrow \rho[\Phi_1(E)]\big)\big]. \tag{2.12}$$

Indeed, assume that $\phi'[E] \,\&\, \Phi_1(E)\!\downarrow$ holds, and let $E' = \Phi_1(E)$. Then by (2.11) we have $\Phi_2(E')\!\downarrow \implies \psi[\Phi_2(E')]$, hence, applying lemma 2.1 with ρ, E' and S_2 in the rôles of ϕ, E and S, it follows that $\rho[E']$ holds, as claimed. Now (2.12) implies $\models [\phi' \implies \phi]$, since $\phi \equiv S_1 \to \rho$. This completes the proof of (2.10) and of (v).

(vi) This part is left as an exercise; you may use lemma 2.1 (ii).

(vii), (viii) These are trivial consequences of $M(\textbf{begin } S \textbf{ end}) = M(S)$.

(ix) Let ϕ be the formula $(B \,\&\, (S \to \psi)) \vee (\neg B \,\&\, (S' \to \psi))$, $\hat{S}$ the conditional statement **if** B **then** S **else** S', $\Phi = M(S)$, $\Phi' = M(S')$ and $\hat{\Phi} = M(\hat{S})$. We have to show that $\models [\{\phi\}\ \hat{S}\ \{\psi\}]$, that is,

$$\models \big[(\forall E \in \mathcal{E})\big(\phi[E] \,\&\, \hat{\Phi}(E)\!\downarrow \implies \psi[\hat{\Phi}(E)]\big)\big] \tag{2.13}$$

holds and that for every formula ϕ',

$$\models [\{\phi'\} \quad \hat{S} \quad \{\psi\}] \implies \models [\phi' \implies \phi]. \tag{2.14}$$

For (2.13) let $E \in \mathcal{E}$, and assume that $\phi[E] \& \hat{\Phi}(E)\!\downarrow$ holds. If $B[E]$ also holds, then $(S \to \psi)[E]$ follows (as a tautological consequence of $\phi[E]$ and $B[E]$), and $\hat{\Phi}(E) = \Phi(E)\!\downarrow$. Hence, by definition of $S \to \psi$, we have $\psi[\hat{\Phi}(E)]$. Similarly, if $\neg B[E]$ holds, then we can deduce $(S' \to \psi)[E]$, $\hat{\Phi}(E) = \Phi'(E)$, and again $\psi[\hat{\Phi}(E)]$. Thus the latter formula follows in either case, proving (2.13).

For (2.14) assume $\models [\{\phi'\}\ \hat{S}\ \{\psi\}]$. Let E be an environment such that $\phi'[E]$ holds; it implies $\hat{\Phi}(E)\!\downarrow \implies \psi[\hat{\Phi}(E)]$. We have to show $\phi[E]$. If $B[E]$ holds, then $\hat{\Phi}(E) = \Phi(E)$, hence $\Phi(E)\!\downarrow \implies \psi[\Phi(E)]$, hence $(S \to \psi)[E]$ by lemma 2.1 (i). Analogously, if $\neg B[E]$ holds, then we can deduce $(S' \to \psi)[E]$. In either case $\phi[E]$ follows.

(x) This part is again left as an exercise.

(xi), (xii) These two assertions follow from (ix) and (x), using (i) and (ii) and the fact that $M(\textbf{if } B \textbf{ then } S)$ is the same as $M(\textbf{if } B \textbf{ then } S \textbf{ else } \Lambda)$. □

The correctness of proof rules (R1) and (R2) from section 1.1 follows directly from theorem 2.1 (i)–(iii), and a formal justification for (R4)–(R7) can be obtained easily by adjusting the argument for other parts. (R3) is entirely trivial once the meaning of $\{\phi\}\ S\ \{\psi\}$ is defined formally; no result about weakest pre- and strongest postconditions corresponds to this rule. Thus it remains to consider **while**-loops. Let S_1 be of the form **while** B **do** S. Is it possible to construct a weakest precondition and a strongest postcondition for S_1 from corresponding conditions for S? In principle the answer is yes, but this fact is of little practical use. A formula $\phi \equiv S_1 \to \psi$ is simply a translation of 'the values of the variables are such that ψ holds after execution of S_1' into first-order predicate logic. In order to prove a specification like $\{\rho\}\ S_1\ \{\psi\}$, one may compute the weakest precondition ϕ and prove $\models [\rho \implies \phi]$, but this will hardly be easier than proving $\{\rho\}\ S_1\ \{\psi\}$ directly (which usually involves finding an appropriate loop invariant). The same remarks also apply for strongest postconditions.

What we show is just the validity of the **while**-rule:

Theorem 2.2: If $\models [\{\phi \,\&\, B\}\ S\ \{\phi\}]$, then $\models [\{\phi\}\ \textbf{while } B \textbf{ do } S\ \{\phi \,\&\, \neg B\}]$.

Proof: Let S_1 be the statement **while** B **do** S, $\Phi = M(S)$ and $\Phi_1 = M(S_1)$. Assume

$$\models [\{\phi \,\&\, B\} \quad S \quad \{\phi\}], \tag{2.15}$$

and let E be an environment such that $\phi[E]\ \&\ \Phi_1(E)\downarrow$ holds. Thus we have to show $(\phi\ \&\ \neg B)[\Phi_1(E)]$. $\Phi_1(E)$ is of the form $\Phi^n(E)$, where n is such that $B[\Phi^m(E)]$ holds for $m < n$, but $B[\Phi^n(E)]$ fails. We show by induction on m that $\phi[\Phi^m(E)]$ holds for $m \leq n$. If $m = 0$, then $\Phi^m(E) = E$, and $\phi[E]$ holds by assumption. Now let $1 \leq m \leq n$, and assume $\phi[\Phi^{m-1}(E)]$. Since $m-1 < n$ we have $(\phi\ \&\ B)[\Phi^{m-1}(E)]$, hence the claim $\phi[\Phi^m(E)]$ follows by applying (2.15). In particular, $\phi[\Phi^n(E)]$ holds. By the choice of n we also have $\neg B[\Phi^n(E)]$, and $\Phi^n(E)$ equals $\Phi_1(E)$. Thus $(\phi\ \&\ \neg B)[\Phi_1(E)]$ holds, as claimed. □

Example 2.2: Theorem 2.1 can be applied mechanically to compute the strongest postcondition

$$\psi \equiv (X < Y \leq x+5) \leftarrow (\textbf{if } X < x \textbf{ then } X := X + Y^2 \textbf{ else if } Y < x \textbf{ then } Y := Y - x),$$

where x is a mathematical variable. Each value $\overline{x}$ of x gives rise to a different conditional statement, with $\overline{x}$ as a constant. Using theorem 2.1, we get

$$\begin{aligned}
\psi \equiv \big[&\big((X < Y \leq x+5\ \&\ X < x) \leftarrow X := X + Y^2\big)\\
&\vee \big((X < Y \leq x+5\ \&\ X \geq x) \leftarrow (\textbf{if } Y < x \textbf{ then } Y := Y - x)\big)\big]\\
\equiv \big[&(\exists u)(u < Y \leq x+5\ \&\ u < x\ \&\ X = u + Y^2)\\
&\vee \big(((X < Y \leq x+5\ \&\ X \geq x\ \&\ Y < x) \leftarrow Y := Y - x)\\
&\vee (X < Y \leq x+5\ \&\ X \geq x\ \&\ Y \geq x)\big)\big]\\
\equiv \big[&(\exists u)(u < Y \leq x+5\ \&\ u < x\ \&\ u = Y^2 - X)\\
&\vee \big((\textbf{false} \leftarrow Y := Y - x) \vee (x < X < Y \leq x+5)\big)\big]\\
\equiv \big[&(Y^2 - X < Y \leq x+5\ \&\ Y^2 - X < x)\\
&\vee (\exists y)(\textbf{false}\ \&\ Y = y - x) \vee x < X < Y \leq x+5\big]\\
\equiv \big[&(Y^2 - X < Y \leq x+5\ \&\ Y^2 - X < x) \vee x < X < Y \leq x+5\big].
\end{aligned}$$

The keywords **true**, **false** denote the condition which is always true (false); for instance, we have $\textbf{true} \equiv y = y$, and $\textbf{false} \equiv x^2 < 0$, assuming that x is type **real**. In our example we observe that the portion of the postcondition corresponding to the **then**-part inside the **else**-part evaluates to **false**. This indicates that this part of the conditional (namely $Y := Y - x$) is never executed if $X < Y \leq x+5$ holds prior to execution, a fact which is easy enough to recognize with hindsight. Therefore the computed postcondition is the same as if the **else**-part were omitted from the conditional statement.

Example 2.3: What about the weakest precondition

$$\phi \equiv \big(\textbf{while } X > 0 \textbf{ do begin } X := X - 1;\ Y := Y + 1 \textbf{ end}\big) \rightarrow Y < Z?$$

Since we have not provided an algorithm for computing weakest pre- and strongest postconditions for **while**-loops, we find ϕ by inspection. Let x, y and z be the values of X, Y and Z before the execution of the **while**-statement. The number of executions of the loop is $\max(x, 0)$, and afterwards the value of Y will be $y + \max(x, 0)$. Hence $Y < Z$ will hold afterwards if and only if $y + \max(x, 0) < z$. Therefore

$$\phi \equiv Y + \max(X, 0) < Z \equiv Y < Z\ \&\ X + Y < Z.$$

3. Applications of program verification

The first two sections of this chapter deal with topics which have been mentioned earlier. Both concern the question: what happens when programs are executed on real rather than mathematical computers? A different aspect of application is set out in section 3.3: It is shown that the verification methods can sometimes be used for the complexity analysis of algorithms, which is usually regarded as a different topic.

3.1. Operations on real computers.

Assume we wish to prove that a given program yields the right result when executed on a *real* computer; how should we proceed? One problem is that operations like $+$ do no longer agree with their mathematical counterparts; this was noted at the beginning of chapter 2. Denote the computer addition by $\tilde{+}$, and similarly for other operations. Then it must be understood that every occurrence of $+$ really means $\tilde{+}$, and the proof rules have to be adjusted accordingly. The modified assignment rule can be given in the form

$$(\text{R2}'') \qquad \{\psi\langle \tilde{E}/X\rangle\} \quad X := E \quad \{\psi\},$$

where $\tilde{E}$ is obtained from E by replacing every function f by the symbol $\tilde{f}$, which denotes the corresponding computer operation. The same kind of modification is needed for other proof rules (including those for *total* correctness), whenever an expression from a program segment has to be used in a mathematical formula, for instance:

$$(\text{R6}'') \qquad \frac{\{\phi \,\&\, \tilde{B}\} \quad S \quad \{\psi\},\ \{\phi \,\&\, \neg\tilde{B}\} \quad S' \quad \{\psi\}}{\{\phi\} \quad \textbf{if } B \textbf{ then } S \textbf{ else } S' \quad \{\psi\}}.$$

In this way the proof of a specification can be reduced to the proof of mathematical assertions, as before. In order to prove mathematical formulas involving computer operations, we have to investigate the mathematical properties of these operations, and this is what we do next.

Consider integer operations first. The domain of the mathematical $+$ is $\mathbb{Z} \times \mathbb{Z}$, but that of $\tilde{+}$ is $\tilde{\mathbb{Z}} \times \tilde{\mathbb{Z}}$, where $\tilde{\mathbb{Z}}$ denotes the set of values of the computer data type **integer**. Typically $\tilde{\mathbb{Z}}$ is of the form $\{\, x \in \mathbb{Z} \mid -2^m \leq x < 2^m \,\}$, where m depends on the computer used. The crucial mathematical property of $\tilde{+}$ is

$$(\forall x, y \in \tilde{\mathbb{Z}})(x + y \in \tilde{\mathbb{Z}} \Longrightarrow x \tilde{+} y = x + y), \tag{3.1}$$

so the computer result coincides with the mathematical result of the operation whenever the latter is computer-representable. The analogue of (3.1) also holds for all other *integer* operations.

A consequence of (3.1) is the following. Let $\mathcal{C}_M$ be a computation on a mathematical computer, and $\mathcal{C}_R$ the corresponding computation on a real computer for the same program and input. Then $\mathcal{C}_M$ and $\mathcal{C}_R$ yield the same result, *provided that all intermediate results belong to* $\tilde{\mathbb{Z}}$. This will be illustrated in example 3.1. So even for practical purposes it makes sense to carry out the verification for mathematical computers, if we can predict that intermediate results (of $\mathcal{C}_M$) are not too large.

If some intermediate result falls outside $\tilde{\mathbb{Z}}$, then an *overflow* occurs. Depending on the implementation of the language, two things can happen in this case:

(i) A run-time error may occur. We will show in section 3.2 how they can be dealt with.
(ii) If $\tilde{\mathbb{Z}}$ is of the form $\{ x \in \mathbb{Z} \mid -2^m \leq x < 2^m \}$ as above, then $x \tilde{+} y$ may be the unique element of $\tilde{\mathbb{Z}}$ satisfying $x \tilde{+} y \equiv x + y \bmod 2^{m+1}$.

Example 3.1: Let F be any integer function, perhaps one which has been declared by the user, n a constant ≥ 0, and let P be the program

$$\begin{array}{l} S := 0; \\ \textbf{for } I := 1 \textbf{ to } n \textbf{ do} \\ \qquad S := S + F(I) \end{array}$$

It is obvious that P computes the sum $F(1) + \cdots + F(n)$, if the intermediate results $F(i)$ and $F(1) + \cdots + F(i)$ $(1 \leq i \leq n)$ belong to $\tilde{\mathbb{Z}}$. We provide a formal proof. The specification to show is

$$\{(\forall i, 1 \leq i \leq n)(F(i) \in \tilde{\mathbb{Z}} \,\&\, F(1) + \cdots + F(i) \in \tilde{\mathbb{Z}})\} \quad P \quad \{S = F(1) + \cdots + F(n)\}. \tag{3.2}$$

Denote pre- and postcondition in (3.2) by ϕ and ψ. The loop invariant is

$$\rho(I): \quad \phi \,\&\, S = F(1) + \cdots + F(I-1);$$

the sum $F(1) + \cdots + F(I-1)$ becomes 0 for $I = 1$ (empty sum). Thus $\rho(1)$ is $\phi \,\&\, S = 0$, and the assignment rule yields immediately

$$\{\phi\} \quad S := 0 \quad \{\rho(1)\}. \tag{3.3}$$

In order to apply the **for**-rule (R12) (with 1 and n in place of e_1 and e_2), we have to show:

$$\{\rho(I) \,\&\, 1 \leq I \leq n\} \quad S := S + F(I) \quad \{\rho(I+1)\}. \tag{3.4}$$

The postcondition $\rho(I+1)$ is $\phi \,\&\, S = F(1) + \cdots + F(I)$; backward substitution according to (R2$''$) yields $\phi \,\&\, S \tilde{+} \tilde{F}(I) = F(1) + \cdots + F(I)$. Thus we have to show

$$\phi \,\&\, S = F(1) + \cdots + F(I-1) \,\&\, 1 \leq I \leq n \Longrightarrow \phi \,\&\, S \tilde{+} \tilde{F}(I) = F(1) + \cdots + F(I). \tag{3.5}$$

Now $\phi \,\&\, 1 \leq I \leq n$ implies $\tilde{F}(I) = F(I)$; this is just the analogue of (3.1) for the function F. (Of course, the values of I, n etc. belong to $\tilde{\mathbb{Z}}$, since the data type of these variables and constants is the computer type **integer**.) Also, applying (3.1) and ϕ with I in place of i, we get $F(1) + \cdots + F(I-1) \tilde{+} F(I) = F(1) + \cdots + F(I-1) + F(I)$, so (3.5) follows, implying (3.4). Then (R12) yields

$$\{\rho(1)\} \quad \textbf{for} \ \ldots \quad \{\rho(n+1)\}, \tag{3.6}$$

since the clauses $E_1 = e_1$ and $E_2 = e_2$ become $1 = 1$ and $n = n$, and $n + 1 \geq 1$. In general, the clauses $E_k = e_k$ $(k = 1, 2)$ disappear in the precondition of the conclusion of (R12) whenever E_1 and E_2 contain no program variables. From (3.3) and (3.6) we deduce the desired specification (3.2), since $\rho(n+1)$ implies ψ.

Two new features occur for computer operations on numbers of type **real**, namely *underflow* and *rounding errors*. Again we consider the mathematical operation $+$, this time on the set $\mathbb{R}$ of real numbers (in the mathematical sense), and the corresponding computer operation $\tilde{+}$ on the set $\tilde{\mathbb{R}}$, which is the set of floating-point numbers representable on the computer. Under 'normal' circumstances the application of $\tilde{+}$ causes only rounding errors, and $\tilde{+}$ satisfies

$$\begin{aligned} (1-\epsilon)(x+y) \leq x\tilde{+}y \leq (1+\epsilon)(x+y) \qquad & \text{for } x+y \geq 0, \\ (1+\epsilon)(x+y) \leq x\tilde{+}y \leq (1-\epsilon)(x+y) \qquad & \text{for } x+y < 0, \end{aligned} \tag{3.7}$$

where ϵ is a fixed small real number depending on the computer precision. Condition (3.7) fails if $x+y$ is greater than the largest floating-point number in $\tilde{\mathbb{R}}$; this leads to an overflow similarly as for integers. It also fails if $x+y$ is nonzero but absolutely smaller than the smallest positive $z \in \tilde{\mathbb{R}}$; in this case an *underflow* occurs. Of course, overflow and underflow happen much easier for multiplication than for addition.

In practice verification is rarely carried out directly for programs as executed on real computers. Instead one shows first that the right result is obtained on a mathematical computer (as we do in the rest of these lecture notes), secondly that the result on a real computer agrees with it 'to a sufficient extend.' If no floating point operations occur, then one has to examine how large the input may be so as to avoid an overflow. Finding good estimates for the size of intermediate results is a problem which belongs to the area *Analysis of Algorithms*. On the other hand, if floating point operations occur, then the effect of rounding errors has to be estimated as well, and this is the topic of *Numerical Mathematics*.

3.2. Run-time errors.

So far run-times errors and nontermination were lumped together. This makes sense inasmuch as the computation fails to yield a result in both cases. Still, from a practical point of view a distinction should certainly be made, so we now provide a framework for dealing with run-time errors.

For this purpose we introduce an *error condition* **err**, and also an *error value* **ERR**. Here **err** represents the situation where a run-time error has occurred. It may be associated with a special element *Error* to be included in the set $\mathcal{E}$, which we refer to as an environment, even though it does not fit definition 2.1. The specification

$$\{\phi\} \quad S \quad \{\mathbf{err}\}$$

means: if ϕ holds ititially, then the execution of S leads to a run-time error. If E is the environment before the execution of S and the execution causes a run-time error, then we have $M(S,E) = \mathit{Error}$ in the notation from seciton 2.1.

We take **ERR** to be the value of an expression whose computation leads a run-time error. Conversely, run-time errors are often, but not always caused by the attempt to evaluate such an expression.

Example 3.2: The following events usually give rise to run-time errors, although ultimately this depends on the underlying programming language.

(i) Division by 0. Thus X/Y has value **ERR** if Y has value 0. Also, $\sqrt{Y} = \mathbf{ERR}$ if Y is of type **real** and negative.

(ii) Array index out of range.

(iii) Attempt to read past the end of a file.

We take another look at situation (ii). Thus let A be an array, and assume that the current value of I lies outside the range of subscripts for A. Then **ERR** is of course the value of the expression $A[I]$, and the attempt to evaluate it causes the run-time error in the assignment $X := A[I]$. On the other hand, if we perform the assignment $A[I] := X$, then the resulting run-time error may or may not be regarded as being caused by the computation of an expression of value **ERR**. This depends on whether we consider A as a single variable, or each component of A (like $A[I]$) as a separate one. The issue will be discussed further in section 4.3.

How about events of type (iii)? Files are the topic of section 4.2; there it will be shown how reading and other file operations can be interpreted as sequences of assignments. If reading past the end of a file is analysed in the light of it, then it turns out once again that an expression of value **ERR** is computed. Namely, the operation **first** generally extracts the first element from a list, but in the situation considered it is applied to the empty sequence.

Next we describe how **ERR** and **err** behave under various operations. The definition of every function F has to be extended by specifying $F(A_1, \ldots, A_n)$ in the case where some A_i has value **ERR**. Usually $F(A_1, \ldots, A_n)$ should be defined to be **ERR**, since the computation of this expression requires that all A_i are evaluated first, giving rise to a run-time error. There are exceptions, however. For instance, let A be an array with index range $1 \ldots n$, assume that the current value of I is 0, and consider the Boolean expression

$$I > 0 \,\&\, A[I] \neq 0. \tag{3.8}$$

Then a run-time error occurs in PASCAL, since the subscript I is out of range. In C, by contrast, (3.8) is evaluated to **false** as soon as $I > 0$ is found to be **false**, so the definition of & differs in these two languages. The following general rule applies in C: a Boolean expression of the form $B_1 \,\&\, \cdots \,\&\, B_n$ evaluates to **false** as soon as some B_i is encountered which evaluates to **false**; it does not matter if $B_j = \mathbf{ERR}$ for some $j > i$. Nor does it matter if for some $j > i$ the computation of B_j fails to terminate, so that its value is undefined; this computation is not carried out in the evaluation of $B_1 \,\&\, \cdots \,\&\, B_n$. Similarly, $B_1 \vee \cdots \vee B_n$ evaluates to **true** as soon as some B_i is encountered which evaluates to **true**. We take the conventions from C in these lecture notes and allow expressions like (3.8).

Next we turn to properties of the condition **err**. In particular, we have to specify which formulas are true and which are false under the environment *Error*. Similarly as for formulas involving undefined values, several possible conventions are appropriate; we choose the following. A predicate $P(A_1, \ldots, A_n)$ has the truth value **false** whenever one of the arguments A_i has the value **ERR**.

Boolean combinations of **err** and other formulas can be simplified only in trivial cases like $\mathbf{false} \,\&\, \mathbf{err} \equiv \mathbf{false}$. A conjunction $\mathbf{err} \,\&\, \psi$ occurs relatively rarely. Indeed, **err** alone provides a complete description of a situation in the course of the execution of a program. Therefore it is pointless to include additional information about the values of program variables, but ψ may be an assertion about free mathematical variables. By contrast, a disjunction $\mathbf{err} \vee \psi$ still makes sense if ψ is an assertion about the program variables, especially as a postcondition. Indeed, the specification $\{\phi\}\ S\ \{\mathbf{err} \vee \psi\}$ asserts that if ϕ holds prior to the execution of S, then the execution either leads to a run-time error, or ψ holds upon termination.

The meaning of some notation should be adjusted for the current section. Namely, $A\downarrow$ rather than $A\uparrow$ holds if A has value **ERR**, and a run-time error counts no longer as nontermination, so the meaning of $\{\phi\}\ S\ \{\psi\}$ and $\{\phi\}\ S'\ \{\psi\}$ may be formulated as in the introduction. Among the proof rules only (R2) and (R2′) have to be modified, but first we note the following specification, which holds generally:

$$\{\mathbf{err}\} \quad S\downarrow \quad \{\mathbf{err}\}. \tag{3.9}$$

It should be interpreted as follows. Typically the *precondition* **err** results from the execution of another program segment S'. Hence regard S as part of $S'; S$, let ϕ be the precondition for S', and assume that S' gives rise to a run-time error, so that $\{\phi\}\ S'\ \{\mathbf{err}\}$ holds. Now if the execution of S' already results in a run-time error, then so does the execution of $S'; S$ a fortiori. This implication is the meaning of the claim 'If **err** holds before the execution of S, then **err** also holds afterwards.' Indeed, (3.9) allows to deduce $\{\phi\}\ S'; S\ \{\mathbf{err}\}$ from $\{\phi\}\ S'\ \{\mathbf{err}\}$, using (R4).

There is yet another way to view (3.9). A computer always stops as soon as a run-time error occurs. Hence if the condition **err** holds prior to the execution of S, then the execution really does not involve any action, so, naturally, conditon **err** is preserved. Both interpretations also explain why total, not just partial correctness may be asserted in (3.9).

Next we have to formalize how **err** results from the computation of an expression of value **ERR**, which happens typically for assignment statements. In order that a predicate $P(A_1, \ldots, A_n)$ holds after an assignment $X := A$, $P(A_1, \ldots, A_n)\langle A/X\rangle$ must hold before, and the value of A must not be **ERR**. Note that $P(A_1, \ldots, A_n)\langle A/X\rangle$ is of the form $P(A'_1, \ldots, A'_n)$, so it implies $\neg\mathbf{err}$ according to our conventions. On the other hand, **err** holds after the assignment iff before the assignment either **err** holds, or the value of A is **ERR**. Thus we obtain the following modified assignment rule:

(R2a) If ϕ' is obtained from ϕ by replacing every atomic subformula $P(A_1, \ldots, A_n)$ by $P(A_1, \ldots, A_n)\langle A/X\rangle \& \neg A = \mathbf{ERR}$, and every occurrence of **err** by $\mathbf{err} \vee A = \mathbf{ERR}$, then

$$\{\phi'\} \quad X := A \quad \{\phi\}.$$

Other proof rules remain unaffected.

Example 3.3: The following program P computes the solution of a system of two linear equations

$$AX + BY = C$$
$$DX + EY = F$$

if it exists and is unique:

$U := AE - BD;$
$X := (CE - BF)/U;$
$Y := (AF - CD)/U$

A run-time error occurs of course in the case where $U = 0$. It may be imagined that this program was jotted down quickly to solve some systems of equations which in most cases do have a unique solution; an occasional run-time error was regarded as acceptable, and the check for $U = 0$ was omitted to save time. As usual we denote the original values of $A, \ldots, F$ by $a, \ldots, f$. Let χ be the formula which asserts that the system has a unique solution. The notation $(\exists! x)(P(x))$ is commonly used; it means: there exists a unique x such that $P(x)$ holds. Then χ has the form

$$(\exists! x, y)(ax + by = c \mathbin{\&} dx + ey = f).$$

The specification satisfied by P is

$$\{\phi\} \quad P \quad \{\psi\}, \tag{3.10}$$

where pre- and postcondition are given as follows:

$$\begin{aligned}
\phi&: \quad A = a \mathbin{\&} B = b \mathbin{\&} \cdots \mathbin{\&} F = f,\\
\psi&: \quad (\chi \Longrightarrow aX + bY = c \mathbin{\&} dX + eY = f) \mathbin{\&} (\neg\chi \Longrightarrow \mathbf{err}).
\end{aligned}$$

To prove (3.10), backward substitution is applied in the modified form according to (R2a). In the first and second step the resulting formula can be simplified by noting that an expression of the form H/U has value **ERR** if and only if $U = 0$. In the last step the substitution corresponding to $U := AE - BD$ can be carried out in the original form (rule (R2)), since $AE - BD$ never equals **ERR**. Hence the following formulas are obtained in order:

$$\begin{aligned}
&(\chi \Longrightarrow aX + b(AF - CD)/U = c \mathbin{\&} dX + e(AF - CD)/U = f \mathbin{\&} U \neq 0)\\
&\qquad \mathbin{\&} (\neg\chi \Longrightarrow \mathbf{err} \vee U = 0),
\end{aligned}$$

$$\begin{aligned}
&(\chi \Longrightarrow a(CE - BF)/U + b(AF - CD)/U = c\\
&\qquad \mathbin{\&} d(CE - BF)/U + e(AF - CD)/U = f \mathbin{\&} U \neq 0) \mathbin{\&} (\neg\chi \Longrightarrow \mathbf{err} \vee U = 0),
\end{aligned}$$

$$\begin{aligned}
&(\chi \Longrightarrow a(CE - BF)/(AE - BD) + b(AF - CD)/(AE - BD) = c\\
&\qquad \mathbin{\&} d(CE - BF)/(AE - BD) + e(AF - CD)/(AE - BD) = f \mathbin{\&} AE - BD \neq 0)\\
&\qquad \mathbin{\&} (\neg\chi \Longrightarrow \mathbf{err} \vee (AE - BD) = 0).
\end{aligned}$$

Denote the last of these formulas by ξ. We have to show $\phi \Longrightarrow \xi$, so assume ϕ. Then ξ follows by the well-known facts from linear algebra. Indeed, if a unique solution exists, that is, χ holds, then $ae - bd \neq 0$, and the solution is given by $x = (ce - bf)/(ae - bd)$ and $y = (af - cd)/(ae - bd)$. If, on the other hand, no unique solution exists, then $ae - bd = 0$.

In these lecture notes, run-time errors, like operations on real computers, are considered only in the one section where they are the main topic. So from now on our former conventions apply again, as do the assignment rules (R2), (R2′). However, in critical cases we will usually provide proof rules which are valid under either convention. For instance, in section 4.2 a proof rule for read-statements will be given, and the precondition will include a clause which asserts that the end of the file has not yet been reached. Thus no postcondition is given for 'reading past the end of the file.'

3.3. Verification techniques for complexity analysis.

In this section we show that the methods for program verification can also be used to show other facts about programs than correctness—typically assertions about complexity. Indeed, such an assertion can be interpreted as a specification for a slightly modified program. For instance, let a program P and a function f be given, and assume we wish to prove that for input x the main loop **while** B **do** S is executed $f(x)$ times. Then we introduce a new variable T which represents the number of executions of S carried out so far. Each execution involves an implicit increment of T by 1. Let P' be the program obtained from P by making these implicit assignments to T explicit, that is, P' is obtained from T by including $T := T + 1$ in the body of the loop S. Then our assertion about the complexity of P is given by the specification

$$\{X = x \mathbin{\&} T = 0\} \quad P' \quad \{T = f(x)\}. \tag{3.11}$$

Here we assume that there is no precondition except that x is the input value, to which X is initialized. Of course, this is not necessarily the case: a nontrivial precondition for P may be included in the precondition in (3.11). The same holds for the postcondition; in this case (3.11) also includes a statement about the correctness of P. Note that any specification which does not involve the new variable T and which is satisfied by P is also satisfied by P' and vice versa.

We consider a trivial example first, just to illustrate the point.

Example 3.4: Let P be the program

$$\textbf{while } X > 0 \textbf{ do } X := X - 1;\ Y := Y + 1 \textbf{ end}$$

It satisfies the specification

$$\{X = x > 0 \mathbin{\&} Y = y\} \quad P \quad \{Y = x + y\}. \tag{3.12}$$

Trivially, the loop is executed x times for $x \geq 0$. Thus if P' is the program

$$\textbf{while } X > 0 \textbf{ do } X := X - 1;\ Y := Y + 1;\ T := T + 1 \textbf{ end}$$

then P' satisfies

$$\{X = x > 0 \mathbin{\&} Y = y \mathbin{\&} T = 0\} \quad P' \quad \{Y = x + y \mathbin{\&} T = x\}. \tag{3.13}$$

A proof of (3.13) establishes both the complexity and the correctness of P. In order to obtain the right loop invariant ρ, note that the values of X, Y and T are $x - z$, $y + z$ and z when the loop has been executed z times. Thus we get

$$\rho\colon \quad X = x - T \mathbin{\&} Y = y + T.$$

The rest is left as a straightforward exercise.

Before turning to a more interesting example we briefly discuss a few variants of our theme. In most cases the time complexity cannot be determined precisely, but only a reasonable upper bound can be given. This means that a modified form of (3.11) has to be shown, where the postcondition $T = f(x)$ is replaced by $T \leq f(x)$. Instead of the executions of some loop we may wish to count operations of some type. For instance, if we wish to prove an assertion about the number of multiplications, then P' is obtained by putting the assignment $T := T+1$ after each multiplication. If P contains assignments where the right side is a complicated expression involving k multiplications, then T has to be increased by k afterwards.

Space complexity can also be handled by similar methods, but there are a few points to take care of. In a sense an array A cannot contribute to the space complexity in a nontrivial way, since its size has to be declared in advance. However, it occurs frequently that only the first l components of A contain relevant information at any stage of the computation, where the current value of l is given by a variable L. Then the maximum value of L indicates the amount of space really needed, and the declared array size must be at least that large. In analogy to the variable T above we have to introduce a variable S which indicates the amount of space used so far, that is, the maximum value of L up to that point. Then an increment of L should be followed by an implicit assignment $S := \max(S, L)$, which has to be explicitly included in the modified program P'.

So with respect to arrays, the interesting problem is not how much space is really occupied (once the declaration is given), but what the declarared array size must be. Nevertheless, the amount of space really required may also be nontrivial, even after the declarations are given. This is due to linked lists and recursive procedures; the latter are implemented by stacks.

Our second example is much trickier and also more interesting than the first.

Example 3.5: Consider once more the Euclidean algorithm from example 1.2. We assume that the function mod yields smallest nonnegative remainders, and that the original values of X and Y are nonnegative. Then the same is true of all intermediate results, and the final assignment $X := |X|$ is no longer necessary. Also, we restrict ourselves to inputs (x, y) with $x > y$. Indeed, if $x < y$, then the first execution of the loop will swap the values of X and Y, so we simply have one more division step than for input (y, x). Also, if $x = y$, then the first remainder is 0 already, and we are done.

We wish to find the best possible estimate for the number of division steps in the *worst case*. So the problem can be stated as follows:

(C) Given m, what is the maximum number n of division steps for inputs (x, y) with $m \geq x > y$?

It turns out to be easier to handle first the *dual problem*, namely:

(D) Given n, find an input (x, y) with $x > y \geq 0$ for which n division steps are needed, with x as small as possible.

We denote a solution for (D) by (x_n, y_n). At the moment we do not yet know that y_n is uniquely determined, though x_n trivially is. Clearly, we have

$$\begin{aligned} x_0 &= 1, & y_0 &= 0, \\ x_1 &= 2, & y_1 &= 1, \\ x_2 &= 3, & y_2 &= 2, \end{aligned}$$

etc. Now let us try to determine (x_3, y_3) from x_2 and y_2. When the Euclidean algorithm is applied to (x_3, y_3), then after one division step X and Y have values $x' = y_3$ and $y' = x_3 \bmod y_3$ for which the algorithm requires two more divison steps. It is plausible to assume that $x' = x_2$ and $y' = y_2$. If so, then $y_3 = 3$, and x_3 must be such that $x_3 \bmod 3 = 2$. The smallest possible value is clearly $3 + 2 = 5$.

In the same way we can find x_{n+1} and y_{n+1} from x_n and y_n for any n. We conjecture that the algorithm applied to (x_{n+1}, y_{n+1}) sets X and Y to x_n and y_n after one division step, which means that $y_{n+1} = x_n$ and $x_{n+1} \bmod y_{n+1} = y_n$, yielding $x_{n+1} = x_n + y_n$ as the smallest possible value. Now we find that in the sequence $\langle x_0, x_1, \ldots\rangle$ each element is the sum of the two preceding ones (except for x_0 and x_1, of course). A sequence defined according to this recurrence relation is the well-known *Fibonacci sequence* $\langle F_0, F_1, \ldots\rangle$, defined by

$$F_0 = 0, \quad F_1 = 1,$$
$$F_{n+2} = F_n + F_{n+1}.$$

A comparison shows $x_0 = F_2$, $x_1 = F_3$ etc. Also $y_n = x_{n-1}$ for $n \geq 1$ (as seen above), and $y_0 = 0$. Thus our conjecture takes the following form:

($*$) The solution for problem (D) is given by $x_n = F_{n+2}$, $y_n = F_{n+1}$, except $y_0 = 0$.

Assuming ($*$), the original problem (C) can also be solved easily. Given $m \geq 1$, let $i = g(m)$ be such that $F_i \leq m < F_{i+1}$. Thus g is roughly the inverse of the function $\lambda i.F_i$. The following table gives the values of $g(m)$ for some small m. The value $g(0)$ will also be needed, and it turns out that it should be defined to be 1.

i	$g(x)$
0	1
1	2
2	3
3	4
4	4
5	5
6	5

Now $i - 2$ division steps are needed for input (F_i, F_{i-1}), so the maximum number n of division steps satisfies $n \geq i - 2$. On the other hand, there exists no input (x, y) with $m \geq x > y$ for which $i - 1$ division steps are needed: the smallest such input is (F_{i+1}, F_i), but $F_{i+1} > m$. Thus we get:

($**$) The solution for problem (C) is given by $n = g(m) - 2$.

It remains to prove ($**$), using the verification methods.

Let P be the following program:

while $Y \neq 0$ **do**
  begin
    $Z := X \bmod Y$;
    $X := Y$;
    $Y := Z$;
    $T := T + 1$
  end

It plays the rôle of P' in the discussion above. In comparison with the original program from example 1.2 the final assignment $X := |X|$ has been omitted, but loop counting has been included.

The specification to show is

$$\{X = x \,\&\, Y = y \,\&\, T = 0\} \quad P \quad \{T \leq g(m) - 2\} \tag{3.14}$$

under the condition

$$m \geq x > y \geq 0. \tag{3.15}$$

So at this point we prove only the upper bound for the complexity, not the correctness of the algorithm (which was done in chapter 1). Denote pre- and postcondition of (3.14) by ϕ and ψ, the body of the loop by S. Of course, the formula (3.15) could also be made a clause of ϕ, but it saves some writing to take it separately. Note that (3.14) is a mathematical formula with free mathematical variables x, y and m, so it may or may not hold for any values of these variables. What we have to show is that (3.15) implies (3.14).

Now comes the tricky part: We have to find the right loop invariant ρ. If the input is (F_i, F_{i-1}), which represents the worst case, then after z executions of the loop the values of X and Y are F_{i-z} and F_{i-z-1}, so $g(X)$ has decreased by z. In accordance with $(**)$ or (3.14), it seems plausible to assume that this is generally the case, since after z executions of S, the number of division steps left is z less than it was initially. Thus, in analogy to example 3.4, it seems natural to take $g(X) \leq g(x) - T$ as the main clause of the invariant. However, since does not work! Indeed, the value of X after one division step is y, so if $g(x) = g(y)$, then the first division step does not decrease $g(X)$. This is the case for $x = 7$, $y = 5$, for instance. Nor does it work to take $g(Y)$ instead of $g(X)$.

What we have to look for is an expression E which combines $g(X)$ and $g(Y)$ and which does decrease with every division step. Some attempts fail, for instance: $g(X) + g(Y)$ is too large, since it does not suffice to show $T \leq g(x) + g(y)$ upon termination. The value of $\max(g(X), g(Y))$ does not decrease with every division step. Indeed, if $X = 7$ and $Y = 5$, then after one step we have $X = 5$, $Y = 2$, but $\max(g(7), g(5))$ and $\max(g(5), g(2))$ both equal 5. Nor does $\min(g(X), g(Y))$ necessarily decrease; in this case a counterexample occurs for $X = 13$ and $Y = 7$. After some trial and error we finally find

$$E\colon \quad \min(g(x), g(y) + 1).$$

So as the main clause of the loop invariant we may take a formula which asserts that after z executions of S ($T = z$), the value of E has decreased by z at least. Actually, it suffices to assert that this value is at most $g(m) - z$, since this yields the right final value of T. In addition, ρ has to include the clause $X > Y \geq 0$. Thus

$$\rho\colon \quad X > Y \geq 0 \,\&\, \min(g(X), g(Y) + 1) \leq g(m) - T.$$

An application of the **while**-rule can only yield

$$\{\rho\} \quad P \quad \{\rho \,\&\, Y \leq 0\}, \tag{3.16}$$

so in order to prove (3.14), we also have to show

$$\phi \Longrightarrow \rho, \quad \text{and} \tag{3.17}$$

$$\rho \,\&\, Y \leq 0 \Longrightarrow \psi. \tag{3.18}$$

Here (3.17) is trivial, using (3.15). (3.18) is also easy: just note that $\rho \,\&\, Y \leq 0$ implies $X > Y = 0$, hence $\min(g(X), g(Y)+1) = 2$, hence $T \leq g(m) - 2$. In order to deduce (3.16) by (R8) we have to show:

$$\{\rho \,\&\, Y \neq 0\} \quad S \quad \{\rho\}. \tag{3.19}$$

Backward substitution from ρ yields the following formulas in order:

$$\begin{gathered}
X > Y \geq 0 \,\&\, \min(g(X), g(Y)+1) \leq g(m) - T - 1, \\
X > Z \geq 0 \,\&\, \min(g(X), g(Z)+1) \leq g(m) - T - 1, \\
Y > Z \geq 0 \,\&\, \min(g(Y), g(Z)+1) \leq g(m) - T - 1, \\
Y > X \bmod Y \geq 0 \,\&\, \min(g(Y), g(X \bmod Y)+1) \leq g(m) - T - 1.
\end{gathered}$$

Denote the last of these formulas by ξ, so it remains to show $\rho \& Y \neq 0 \Longrightarrow \xi$. Assuming $\rho \,\&\, Y \neq 0$, the first clause $Y > X \bmod Y \geq 0$ is clear. For the second clause it suffices to show

$$\min(g(Y), g(X \bmod Y)+1) < \min(g(X), g(Y)+1), \tag{3.20}$$

which is the key point of the whole proof. Write U for $X \bmod Y$, so U is of the form $X - qY$ with $q \geq 1$. Also, let $g(X) = i$, $g(Y) = j$. We have to consider two cases.

Case (a): $i > j$. Then $\min(g(X), g(Y)+1) = j+1$, and $\min(g(Y), g(X \bmod Y) + 1) \leq j < j+1$, so (3.20) follows.

Case (b): $i = j$. In this case $\min(g(X), g(Y)+1) = i = g(Y)$, so we have to show $g(X \bmod Y) + 1 = g(U) + 1 < i$. At this point the definition of g is finally used. Since $i = g(X) = g(Y)$ we have $F_i \leq Y < X < F_{i+1}$, hence $U \leq X - Y < F_{i+1} - F_i = F_{i-1}$, hence $g(U) < i - 1$, and the claim $g(U) + 1 < i$ follows. The proof is complete.

4. Data structures

This chapter and the next deal with the most essential features of programming languages not considered in chapter 1: structured data types and procedures/functions. Among the structured data types records and sets cause virtually no new difficulties for program verification, and they are considered first.

4.1. Records and sets.

In the theory of program verification records play no essential rôle. In PASCAL a record type is declared in the form

$$\textbf{type}\ \mathrm{RTP} = \textbf{record}\ F_1{:}\,T_1;\ \ldots\ F_n{:}\,T_n\ \textbf{end},$$

where RTP and $F_1, \ldots, F_n, T_1, \ldots, T_n$ are identifiers; T_i specifies the data type of the field F_i. If R is a variable of type RTP, then R has components $R.F_i$ of type T_i ($i = 1, \ldots, n$). In order to prove a specification involving R, one simply treats $R.F_1, \ldots, R.F_n$ as separate variables.

A new proof rule is needed in order to deal with **with**-statements. Such a statement has the form

$$\textbf{with}\ R\ \textbf{do}\ S;$$

the meaning is that within S every occurrence of a field identifier F_i means $R.F_i$. This leads trivially to the following proof rule:

$$\text{(R15)}\qquad \frac{\{\phi\}\quad S\langle R.F_1/F_1, \ldots, R.F_n/F_n\rangle\quad \{\psi\}}{\{\phi\}\quad \textbf{with}\ R\ \textbf{do}\ S\quad \{\psi\}}.$$

Simple as it is, this rule illustrates a relevant principle: A program segment P which involves some peculiar construct can be verified by replacing it by an equivalent program segment P' where this construct does not appear. In our example the statement **with** R **do** S is replaced by $S\langle R.F_1/F_1, \ldots, R.F_n/F_n\rangle$.

Next we consider sets. In the rest of this section the letter S will be used for variables of set type, not for program segments. For instance, if S is declared as a **set of integer**, then the value of S may be any *finite* set of integers. From an abstract theoretical point of view we might also consider arbitrary sets S, but such S has in general no finite representation, thus violating certain principles of computability. Still, since we base these lecture notes on mathematical rather than real computers, no other restriction than finiteness is imposed on sets. In real programming languages the values of a set variable are usually restricted much more severely, namely to subsets of some data type which is much smaller than the computer type **integer**.

No new principles are involved in the verification of programs where sets occur in expressions or at the left side of assignment statements. Sets may be handled like integers or Booleans etc., and in particular the assignment rule does not change. We give an example which is somewhat more involved than earlier ones, and which provides a first application of rule (R14). No previous example illustrated **for**-loops with nontrvial increment. Only the main points of the verification will be given.

Example 4.1: The following program P computes the set S of all prime numbers $\leq M$, where $M \geq 2$. The algorithm used is the sieve of Eratosthenes.

$S := \{2\}$;
for $I := 3$ **to** M **step** 2 **do**
  $S := S \cup \{I\}$;
for $D := 3$ **to** $\lfloor \sqrt{M} \rfloor$ **step** 2 **do**
  if $D \in S$ **then**
    for $I := D^2$ **to** M **step** $2D$ **do**
      $S := S - \{I\}$

A few explanations are in order. Usually the sieve algorithm is presented in the following form: Start with the set $\{2, \ldots, M\}$, eliminate all multiples of 2 (except 2 itself), then all nontrivial multiples of 3, then of 5 etc. The number whose multiples are being eliminated is represented by the variable D. After the multiples of some d have been removed from S, find the smallest $d' > d$ which still belongs to S; this d' is the next value of D. In the program P the multiples of 2 are not included in S in the first place; this allows to increment D by 2 rather than 1.

The largest D to consider is $l = \lfloor \sqrt{M} \rfloor$. Indeed, if $D > l$, then no multiples I of D are left to eliminate, for the following reason. If I is a nontrivial multiple of $D > l$, then I is also a multiple of $D' = I/D < M/\sqrt{M} < l + 1$, hence $D' \leq l$, so I was already eliminated when multiples of D' were taken.

Now consider some fixed D. Multiples of D are removed from S by the loop

$$\textbf{for } I := D^2 \textbf{ to } M \textbf{ step } 2D \textbf{ do } S := S - \{I\}.$$

Here the expressions D^2 and $2D$ should be explaned. The smallest multiple of D to be eliminated is indeed D^2, since any smaller multiple DD' with $1 < D' < D$ has already been eliminated earlier. And the increment for I is $2D$ rather than D, since only odd multiples of D have to be considered; even numbers (except for 2, which is $< D$) were left out of S from the beginning.

The following notation is used: $x|y$ asserts that x divides y, $x\nmid y$ is the negation of $x|y$, and $\mathrm{Pr}(x)$ asserts that x is prime. The specification to show is

$$\{\phi\} \quad P \quad \{\psi\}, \tag{4.1}$$

where pre- and postcondition are as follows:

$$\begin{aligned} \phi&: \quad M = m \geq 2, \\ \psi&: \quad S = \{\, x \leq m \mid \mathrm{Pr}(x) \,\}. \end{aligned}$$

The precondition $m \geq 2$ is necessary; otherwise the final value of S would be $\{2\}$, although it should be $\emptyset$. The condition ϕ remains valid throughout.

Denote the the first, second and third **for**-statement by P_1, P_2 and P_3; thus P_3 is contained within P_2. P_1 simply puts odd numbers into S and contains nothing subtle. The loop invariant is

$$\rho_1: \quad \phi \mathbin{\&} S = \{2\} \cup \{\, x \mid x \text{ is odd} \mathbin{\&} 3 \leq x < I \,\}.$$

It is straightforward to show

$$\{\phi\} \quad S := \{2\} \quad \{\rho_1(3)\}, \quad \text{and} \tag{4.2}$$
$$\{\rho_1(I) \,\&\, \mathrm{betw}(3, m, 2, I)\} \quad S := S \cup \{I\} \quad \{\rho_1(I+2)\}. \tag{4.3}$$

From (4.3) we deduce by (R14):

$$\{\rho_1(3)\} \quad P_1 \quad \{\rho_1(\mathrm{lv}(3, m, 2))\}; \tag{4.4}$$

the clause $M = m$ is contained in $\rho_1(3)$. Thus $\rho_1(\mathrm{lv}(3, m, 2))$ holds when P_1 is finished. This condition can be written as

$$M = m \geq 2 \,\&\, S = \{2\} \cup \{\, x \mid x \text{ is odd} \,\&\, 3 \leq x \leq m \,\}, \tag{4.5}$$

using the fact that $\mathrm{lv}(3, m, 2)$ is the smallest odd number $> m$.

What is the invariant ρ_2 for P_2? Before this loop is executed for $D = d$, multiples have been eliminated of 2, 3, 5, ..., up to the largest prime $< d$. Of course, multiples i of composite numbers $d' < d$ have been eliminated as well, since each such d' has a prime factor $< d$ which also divides i. Thus we get

$$\rho_2(D){:} \quad \phi \,\&\, S = \{\, x \mid 2 \leq x \leq m \,\&\, (\forall d')(2 \leq d' < D \,\&\, d' < x \Longrightarrow d' \nmid x) \,\}.$$

We have to show

$$\{\rho_2(D) \,\&\, \mathrm{betw}(3, e_2, 2, D)\} \quad \textbf{if } D \in S \textbf{ then } P_3 \quad \{\rho_2(D+2)\}, \tag{4.6}$$

in order to deduce

$$\{\lfloor \sqrt{M} \rfloor = e_2 \,\&\, \rho_2(3)\} \quad P_2 \quad \{\rho_2(\mathrm{lv}(3, e_2, 2))\}. \tag{4.7}$$

One remark about this application of (R14). Recall that in the hypothesis and conclusion of (R14), e_1–e_3 are free mathematical variables, and it is understood that these specifications hold for all values of the e_i. However, the rule also holds with constants substituted for some of these variables. In the current instance it suffices to consider $e_1 = 3$ and $e_3 = 2$, but the free variable e_2 cannot yet be eliminated, since the corresponding program expression E_2 contains the program variable M. All the same, we will see that e_2 disappears shortly.

Indeed, in the precondition of (4.7), M may be replaced by m, since $M = m$ is contained in $\rho_2(3)$. Then (4.7) may be applied with $\lfloor \sqrt{m} \rfloor$ in place of e_2, yielding

$$\{\rho_2(3)\} \quad P_2 \quad \{\rho_2(\mathrm{lv}(3, \lfloor \sqrt{m} \rfloor, 2))\}. \tag{4.8}$$

It is easy to see that $\rho_2(3)$ follows from (4.5), which is equivalent to the postcondition in (4.4). Hence from (4.2), (4.4) and (4.8) we obtain

$$\{\phi\} \quad P \quad \{\rho_2(\mathrm{lv}(3, \lfloor \sqrt{m} \rfloor, 2))\}. \tag{4.9}$$

Let ψ' be the postcondition in (4.9). It asserts that S consists of those x with $2 \leq x \leq m$ which have no divisor d' with $2 \leq d' < \mathrm{lv}(3, \lfloor \sqrt{m} \rfloor, 2)$ and $d' < x$. But the latter condition is equivalent to $\Pr(x)$. Indeed, if x, $2 \le x \le m$ is composite, then x has a factorization dd' with $1 < d' \le d < x$, which implies $d' \le \sqrt{m} < \mathrm{lv}(3, \lfloor \sqrt{m} \rfloor, 2)$. This means that ψ' implies ψ, so the desired specification (4.1) follows from (4.9).

It remains to prove (4.6), which was needed for (4.7). P_2 is a conditional, so we need

$$\{\rho_2(D)\ \&\ \mathrm{betw}(3, e_2, 2, D)\ \&\ D \in S\} \quad P_3 \quad \{\rho_2(D+2)\}, \quad \text{and} \tag{4.10}$$

$$\rho_2(D)\ \&\ \mathrm{betw}(3, e_2, 2, D)\ \&\ D \notin S \Longrightarrow \rho_2(D+2). \tag{4.11}$$

(4.11) can be seen as follows. In view of the definition of ρ_2 we have to show

$$\begin{aligned} M = m \ge 2\ &\&\ \mathrm{betw}(3, e_2, 2, D) \\ &\&\ D \notin S = \{\, x \mid 2 \le x \le m\ \&\ (\forall d')(2 \le d' < D\ \&\ d' < x \Longrightarrow d' \nmid x) \,\} \\ \Longrightarrow S &= \{\, x \mid 2 \le x \le m\ \&\ (\forall d')(2 \le d' < D + 2\ \&\ d' < x \Longrightarrow d' \nmid x) \,\}. \end{aligned} \tag{4.12}$$

Assuming the left side of (4.12), it suffices to show that no $x \in S$ is divisible by D or $D+1$. The condition $\mathrm{betw}(3, e_2, 2, D)$ implies that D is odd and $D \ge 3$, hence S contains no even number > 2, and x certainly is not divisible by $D+1$. Now $D \notin S$, hence D is divisible by some d' with $2 \le d' < D$. If $x \in S$ were divisible by D, then x would also be divisible by $d' < x$, contradicting the representation of S at the left side of (4.12). Thus (4.11) follows.

We formulate the loop invariant ρ_3 for P_3 in English. $\rho_3(I)$ asserts that ϕ holds, and S contains precisely those x, $2 \le x \le m$ such that x is not divisible by any d' with $2 \le d' < D$ and $d' < x$, and x is also not divisible by D if $D < X < I$. The proof of (4.10) is then straightforward; the remaining details are left as an exercise.

There is a new control structure involving sets which does require a new proof rule, namely loops of the form

$$\textbf{for } I \in S \textbf{ do } P, \tag{4.13}$$

with S a set. This type of statement is allowed in the computer algebra system MAPLE, for instance. The meaning is of course that the instruction P has to be carried out for all $I \in s$, where s is the initial value of S. Similarly as in section 1.4 we stipulate that the set of values for which the loop has to be executed remains unaffected if P contains assignments to S. The order in which I runs through the elements of s is not specified, hence an algorithm using loops of this form is *nondeterministic* (unless $|s| \le 1$). This is how (4.13) should be viewed in theory, although the computation is of course deterministic for any real implementation. Indeed, it would be very unsatisfactory to provide a specification for (4.13) whose validity depends on the order in which the elements of s are considered.

Nondeterminism is a new feature not considered before. A more typical example of a nondeterministic step in a computation is the choice of some random element. The definition of the semantics from section 2.1 has to be modified somewhat for nondeterministic programs. If P_0 is nondeterministic, then several valid executions of P_0 may be possible for any given initial environment. Then a specification $\{\phi\}\ P_0\ \{\psi\}$ has the following meaning:

For every valid execution $\mathcal{C}$ of P_0, if ϕ holds prior to $\mathcal{C}$, then ψ holds after $\mathcal{C}$.

Returning to loops of the form (4.13), a specification of the form $\{\phi\}$ **for** $I \in S$ **do** P $\{\psi\}$ means that if ϕ holds before the execution, then ψ holds afterwards, irrespective of the order in which I runs through S. An appropriate proof rule is obtained similarly to (R12)–(R14). The loop invariant is of the form $\rho(t)$, where t represents the set of all elements of S for which the loop has been carried out prior to the execution for some particular value of I. Thus $\rho(\emptyset)$ holds initially and $\rho(s)$ holds upon termination. If $\rho(t)$ holds before the execution for some $I \in s$, then $\rho(t \cup \{I\})$ holds afterwards (provided that ρ has been chosen appropriately). The resulting proof rule is

$$\text{(R16)} \qquad \frac{\{\rho(t) \;\&\; t \subset s \;\&\; I \in s - t\} \quad P \quad \{\rho(t \cup \{I\})\}}{\{S = s \;\&\; \rho(\emptyset)\} \quad \textbf{for } I \in S \textbf{ do } P \quad \{\rho(s)\}}.$$

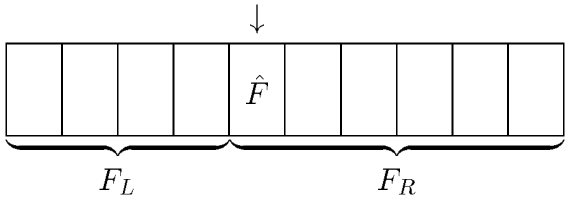
A straightforward application of (R16) is the proof of the specification

$$\{U = 0\} \quad \textbf{for } I \in S \textbf{ do } U := U + I \quad \{U = \sum_{I \in S} I\};$$

we leave this again as an exercise. A more interesting application will be given in section 4.4.

4.2. Files.

As long as no variables of file type occur, an environment completely describes the situation during a computation, except for the position of the instruction counter. This is no longer the case when a file F occurs; in this case some additional information besides the contents of F is needed. Assume first that the file is open for reading. Then it has to be known which element of F is being scanned by the reading head H. The relevant information is contained in the three variables F_L, F_R and $\hat{F}$; the latter is called the *file buffer*. We consider them as program variables, although they do not appear in the program itself, since every file operation may be regarded as an operation on F_L, F_R and $\hat{F}$. F_L is the portion of F to the left and F_R the portion to the right of H. The element currently scanned is also included in F_R; it equals $\hat{F}$ usually, but not necessarily, since explicit assignments to $\hat{F}$ are also possible. Thus we get the following picture:



When reasoning about computations involving files, we may always pretend that the first element of F_R is the next element to be read. In reality a computer works slightly differently, since it always reads one position ahead (in comparison with the read-statements executed). The extra element read is stored in the variable $\hat{F}$. The values (contents) of F_L, F_R and of F itself are sequences of elements of the same type as $\hat{F}$. We use the following

Notation: Let F, G be sequences of elements of type T, x, x_1, ... individual elements of type T.

(i) $\langle x_1, \ldots, x_n\rangle$ is the file (sequence) consisting of elements x_1, ..., x_n. In particular, $\langle\rangle$ denotes the empty file.

(ii) FG is the concatenation of F and G; xF and Fx mean the same as $\langle x\rangle F$ and $F\langle x\rangle$ respectively.

(iii) $\text{first}(F)$ is the first element of F and $\text{rest}(F)$ the file F with its first element removed. Thus $\text{first}(xF) = x$ and $\text{rest}(xF) = F$. Both $\text{first}(F)$ and $\text{rest}(F)$ are undefined for $F = \langle\rangle$.

(iv) Now assume that F is a file in a program, so F_L and F_R are also defined in any environment. Then $\text{eof}(F)$ means $F_R = \langle\rangle$, that is, the reading head H has already reached the end of F. (The function eof, which takes an argument of type **file of** T and returns a value of type Boolean, is available in programming languages like PASCAL.)

The condition $\neg\text{eof}(F)$ is necessary in order that reading from F is possible and does not lead to a run-time error. Indeed, if $\text{eof}(F)$ holds, then the computer would read past the end of F. On the other hand, if F is open for writing rather than reading, then the writing head always scans the end of F, where the next element may be added, so $\text{eof}(F)$ holds necessarily. For simplicity we assume that the conditions $\neg\text{eof}(F)$ and $\text{eof}(F)$ are also *sufficient* for read- or write-commands respectively to be executable. In reality the situation is somewhat different: a *file status* is defined for each file F, which indicates whether F is closed, open for reading or open for writing.

Next we describe the file-commands and their effects. $\text{reset}(F)$ opens the file for reading and places the reading head at the beginning of F; the first element becomes the value of $\hat{F}$. So the effect of $\text{reset}(F)$ corresponds to the following three simultaneous assignments:

$$F_L := \langle\rangle, \quad F_R := F_L F_R, \quad \hat{F} := \text{first}(F_L F_R).$$

If the third assignment were written in the form $\hat{F} := \text{first}(F_L)$, then it would not be entirely correct, since it would fail in the case $F_L = \langle\rangle$. If $F = \langle\rangle$, then $\text{first}(F_L F_R)$ is undefined (or **ERR**); in this case $\text{reset}(F)$ does not lead to a run-time error, but $\hat{F}$ is undefined afterwards.

Now (R2) can be extended in the obvious way to deal with several simultaneous assignments: in order to obtain the right precondition one only has to carry out the corresponding substitutions simultaneously. Thus we obtain the proof rule

$$\text{(R17)} \qquad \{\phi\langle\langle\rangle/F_L, F_L F_R/F_R, \text{first}(F_L F_R)/\hat{F}\rangle\} \quad \text{reset}(F)\downarrow \quad \{\phi\}.$$

This rule is analogous to (1.26): we may assert total correctness, since, as noted, $\text{reset}(F)$ terminates without run-time error even if $F = \langle\rangle$. In this case $\hat{F}$ is undefined in the postcondition, $\text{first}(F_L F_R)$ is undefined in the precondition, and the specification remains valid according to our conventions about formulas involving undefined values (section 1.3).

Usually we do not consider the validity of specifications were the value of some variable may be 'undefined' ($\uparrow$). This requires an extension of the definition of *environment* from chapter 2. Actually, file buffers are the only variables which we allow to be undefined; this may also result from other operations (besides reset), as will be seen shortly.

The operation rewrite(F) empties the file and opens it for writing. Again $\hat{F}$ is undefined afterwards, but in contrast to reset, we do not stipulate that in the precondition $\uparrow$ should be substituted for $\hat{F}$. Instead we require that $\hat{F}$ does not occur in the postcondition ϕ. The reason is that $\hat{F}$ is *always* undefined afterwards, so there is no point in making an assertion about $\hat{F}$ in ϕ. Since F_L andF_R are set to $\langle\rangle$, we get

$$\text{(R18)} \qquad \{\phi\langle\langle\rangle/F_L, \langle\rangle/F_R\rangle\} \quad \text{rewrite}(F)\!\downarrow \quad \{\phi\} \,\&\, \text{eof}(F).$$

When this rule (like those for put and write below) is applied, then ϕ must not contain F; we have to write $F_L F_R$ instead. The condition eof(F) clearly holds afterwards, so we include it in the postcondition, but actually it would suffice to give the rule in the form

$$\{\psi\langle\langle\rangle/F_L, \langle\rangle/F_R\rangle\} \quad \text{rewrite}(F)\!\downarrow \quad \{\psi\}. \tag{4.14}$$

Then (R18) is obtained by using (4.14) with $\phi \,\&\, \text{eof}(F)$ in the rôle of ψ: eof(F) means $F_R = \langle\rangle$, which upon backward substitution becomes $\langle\rangle = \langle\rangle$, a clause does not affect the precondition.

The commands *get* and *read* serve to retrieve data from F. The latter is of course much more commonly used, but get(F) is in a sense more elementary: it advances H by one position and sets $\hat{F}$ to the first element of the new F_R. A run-time error results if F_R was empty before the execution; this is ruled out by including the clause $\neg$eof(F) in the preconditon. No run-time error results if F_R is empty afterwards, although $\hat{F}$ will be undefined in this case. In this respect get resembles reset. It is clear how get(F) changes F_L, F_R and $\hat{F}$ in general; the resulting proof rule is

$$\text{(R19)} \qquad \{\neg\text{eof}(F) \,\&\, \phi\langle F_L\text{first}(F_R)/F_L, \text{rest}(F_R)/F_R, \text{first}(\text{rest}(F_R))/\hat{F}\rangle\} \quad \text{get}(F)\!\downarrow \quad \{\phi\}.$$

The instruction read(F, X) has the same effect as get(F), and in addition it assigns the previous value of $\hat{F}$ to X. This value is usually the first element of F_R before execution, so this is why $\hat{F}$ is preceived as 'the next element to be read from F.' The resulting proof rule is clear:

$$\text{(R20)} \qquad \{\neg\text{eof}(F) \,\&\, \phi\langle F_L\text{first}(F_R)/F_L, \text{rest}(F_R)/F_R, \text{first}(\text{rest}(F_R))/\hat{F}, \hat{F}/X\rangle\} \quad \text{read}(F,X)\!\downarrow \quad \{\phi\}.$$

For storing data on F there are again two commands, *put* and *write*, of which the latter is used much more often. This time the condition eof(F) has to be included in the precondition in order to rule out a run-time error; this condition remains valid afterwards. The instruction put(F) attaches the current value of $\hat{F}$, and this value has to be defined. $\hat{F}$ will be undefined afterwards, so again we require that $\hat{F}$ does not occur in ϕ.

$$\text{(R21)} \qquad \{\text{eof}(F) \,\&\, \hat{F}\!\downarrow \,\&\, \phi\langle F_L\hat{F}/F_L\rangle\} \quad \text{put}(F)\!\downarrow \quad \{\phi \,\&\, \text{eof}(F)\}.$$

Finally, write(F, X) attaches the value of X rather than $\hat{F}$, so we get the rule

$$\text{(R22)} \qquad \{\text{eof}(F) \,\&\, \phi\langle F_L X/F_L\rangle\} \quad \text{write}(F,X)\!\downarrow \quad \{\phi \,\&\, \text{eof}(F)\}.$$

As for (R21) it is required that ϕ does not contain $\hat{F}$.

Example 4.2: The following program S copies file F into G:

reset(F); rewrite(G);
while not eof(F) **do**
  begin read($F, \hat{G}$); put(G) **end**

We prove a specification which asserts that $F = G$ upon termination, and that F is left unchanged. Proceeding as usual, we denote the initial value (contents) of F by f, and we have to write $F_L F_R$ for F and $G_L G_R$ for G in the various conditions. So the specification to prove is

$$\{\phi\} \quad S \quad \{\psi\} \tag{4.15}$$

with

$$\begin{aligned} \phi&: \quad F_L F_R = f, \\ \psi&: \quad F_L F_R = G_L G_R = f. \end{aligned}$$

Now for the loop invariant. Denote the portion of S before the loop by S_1, the body of the loop by S_2. The following conditions hold before and after every execution of S_2: F still equals f, F_L is the portion of F copied so far, $G_R = \langle\rangle$ since G is open for writing, and $\hat{F} = \mathrm{first}(F_R)$, a condition which holds after each of the file operations on F. Thus we get

$$\rho: \quad F_L F_R = f \mathbin{\&} F_L = G_L \mathbin{\&} \mathrm{eof}(G) \mathbin{\&} \hat{F} = \mathrm{first}(F_R).$$

First we have to show that ρ holds before the first execution of S_2, that is,

$$\{\phi\} \quad S_1 \quad \{\rho\}. \tag{4.16}$$

Working backwards from ρ and using rules (R18) and (R17), we get:

$$\begin{gathered} F_L F_R = f \mathbin{\&} F_L = \langle\rangle \mathbin{\&} \hat{F} = \mathrm{first}(F_R), \\ \langle\rangle F_L F_R = f \mathbin{\&} \langle\rangle = \langle\rangle \mathbin{\&} \mathrm{first}(F_L F_R) = \mathrm{first}(F_L F_R), \end{gathered}$$

which follows from ϕ. Note that $\mathrm{first}(F_L F_R)$ may be undefined, but the equality $\mathrm{first}(F_L F_R) = \mathrm{first}(F_L F_R)$ holds even in this case according to the conventions from section 1.3. Thus (4.16) is established.

Next we show that ρ is indeed a loop invariant:

$$\{\rho \mathbin{\&} \neg\mathrm{eof}(F)\} \quad S_2 \quad \{\rho\}. \tag{4.17}$$

Again we work backward from ρ, this time using (R21) and (R20):

$$\begin{aligned} &\mathrm{eof}(G) \mathbin{\&} \hat{G}{\downarrow} \mathbin{\&} F_L F_R = f \mathbin{\&} F_L = G_L \hat{G} \mathbin{\&} \hat{F} = \mathrm{first}(F_R) \\ &\neg\mathrm{eof}(F) \mathbin{\&} \mathrm{eof}(G) \mathbin{\&} \hat{F}{\downarrow} \mathbin{\&} F_L \mathrm{first}(F_R) \mathrm{rest}(F_R) = f \\ &\qquad\qquad \mathbin{\&} F_L \mathrm{first}(F_R) = G_L \hat{F} \mathbin{\&} \mathrm{first}(\mathrm{rest}(F_R)) = \mathrm{first}(\mathrm{rest}(F_R)) \end{aligned}$$

Denote the last of these formulas by ξ; we have to show

$$\rho \mathbin{\&} \neg\mathrm{eof}(F) \Longrightarrow \xi. \tag{4.18}$$

Here ρ contains the clause $\mathrm{eof}(G)$, as well as $\hat{F} = \mathrm{first}(F_R)$, which together with $\neg\mathrm{eof}(F)$ implies $\hat{F}\downarrow$, since $\neg\mathrm{eof}(F)$ is equivalent to $F_R \neq \langle\rangle$. Also, $\neg\mathrm{eof}(F)$ implies $F_L\mathrm{first}(F_R)\mathrm{rest}(F_R) = F_L F_R$, which equals f according to ρ. Thus (4.18) and hence (4.17) follows, and the **while**-rule yields

$$\{\rho\} \quad \textbf{while} \ \ldots \quad \{\rho \,\&\, \mathrm{eof}(F)\}. \tag{4.19}$$

To complete the proof of (4.15), it remains to show

$$\rho \,\&\, \mathrm{eof}(F) \Longrightarrow \psi. \tag{4.20}$$

The left side of this implication contains the clauses $\mathrm{eof}(F)$ and $\mathrm{eof}(G)$, that is, $F_R = G_R = \langle\rangle$, so we get indeed $f = F_L F_R = F_L = G_L = G_L G_R$, which is ψ.

A new approach is needed to show the correctness of *interactive* programs. In this case it does not suffice to relate the contents of input- and output files before and after execution; one also has to show that input and output are intermixed correctly. For this purpose yet another file has to be introduced, namely the sequence of input- and output symbols as it appears at the terminal. We do not pursue this issue any further.

4.3. Arrays.

For an array A let s_A the corresponding set of subscripts; it is fixed as soon as the declaration is given, so it has to be treated like a constant. For instance, if A was declared by

var A: **array**[2..17] **of integer**

in PASCAL, then s_A is $\{2, \ldots, 17\}$. In practice s_A is of course always an interval of $\mathbb{Z}$ or of some other scalar data type if A onedimensional. If A is multidimensioal, then we may either take s_A as a cartesian product of such intervals, or we may regard A as a onedimensional array whose components are in turn arrays. From a theoretical point of view there is no need to put any restriction on s_A; in particular, we allow s_A to be all of ω or $\mathbb{Z}$ (unbounded array).

In contrast to sets and files, the syntax of common programming languages suggests to treat array components like individual variables. Indeed, if E is an expression, then $A[E]$ is called a *subscripted variable*, and $A[E]$ may occur where variables are allowed, for instance, at the left side of assignment statements or as actual variable parameters in procedure calls. (Variables which are not subscripted will from now on be called *simple variables.*) Taking array components as single variables is the approach we follow first, although, in contrast to records, it will be seen shortly that this leads to some difficulties. The alternative, which will be considered later, is to regard only A as a whole as a variable.

Most proof rules remain unaffected by the consideration of arrays; only the assignment rule (R2) (or (R2′)) causes some trouble. Indeed, in certain cases it is not clear how the substitution required by this rule should be performed. Given ψ, the appropriate precondition ϕ satisfying $\{\phi\}\ A[5] := A[6]{+}1\ \{\psi\}$ should be $\psi\langle A[6]{+}1/A[5]\rangle$, but it is not obvious how $\psi\langle A[6] + 1/A[5]\rangle$ is to be defined if ψ is a formula like $(\forall i \in s_A)(A[i] > 5)$.

On the other hand, there are cases where a naïve way of carrying out the substitution suggests itself, which, however, is wrong! If X is a simple variable, then trivially we have

$$\{\textbf{true}\} \quad X := 1 \quad \{X = 1\}. \tag{4.21}$$

Of course, backward substitution yields the precondition $1 = 1$, which is equivalent to **true**. Does (4.21) remain valid if X is replaced by a subscripted variable? Not necessarily! The specification

$$\{\mathbf{true}\} \quad A[A[2]] := 1 \quad \{A[A[2]] = 1\}$$

fails if prior to the assignment we have $A[1] = A[2] = 2$. In this case the assignment sets $A[2]$ to 1, and thereafter we have $A[A[2]] = A[1] = 2$. The problem is clear: $A[2]$ has changed, so $A[A[2]]$ refers to different addresses before and after the assignment.

As a remedy we may either construct an entirely new assignment rule for subscripted variables, or just redefine the operaton of substitution. The latter turns out to be sufficient, so we now define $E'\langle E/X\rangle$ for any expressions E' and E and any variable X. In doing so the assignment rule should be kept in mind.

(a) We have

$$F(E_1, \ldots, E_n)\langle E/X\rangle = F(E_1\langle E/X\rangle, \ldots, E_n\langle E/X\rangle),$$

and similarly if E' is of the form **if** B **then** E_1 **else** E_2. In this way the problem reduces to the case where E' is just a simple or indexed variable.

(b) If E' is a *simple* variable, then

$$E'\langle E/X\rangle = \begin{cases} E & \text{for } E' = X, \\ X & \text{otherwise.} \end{cases}$$

This is of course the obvious way of performing the substitution. We are left with the case where E' is a subscripted variable $A[\tilde{E}]$.

(c) If X is a simple variable, then

$$A[\tilde{E}]\langle E/X\rangle = A[\tilde{E}\langle E/X\rangle],$$

that is, the substitution is applied to the subscript. The same works if X is a subscripted variable from a different array than A:

$$A[\tilde{E}]\langle E/A'[E_0]\rangle = A[\tilde{E}\langle E/A'[E_0]\rangle].$$

(d) We now turn to the last and most interesting case, where X is also a component of A: $X = A[\hat{E}]$. So we have to define $A[\tilde{E}]\langle E/A[\hat{E}]\rangle$ such that the assignment rule is preserved:

$$\{\phi(A[\tilde{E}]\langle E/A[\hat{E}]\rangle)\} \quad A[\hat{E}] := E \quad \{\phi(A[\tilde{E}])\}. \tag{4.22}$$

In the assignment rule in the original form (R2), the precondition asserts the following: the values of the variables are such that the postcondition ϕ holds with X replaced by its new value, which is given by (the old value of) E. Analogously, in (4.22) $A[\tilde{E}]\langle E/A[\hat{E}]\rangle$ has to be the new value of $A[\tilde{E}]$, expressed in terms of old values. We indicate the new value of some expression by the subscript new, whereas without this subscript the old value is meant. Then

$$A_{\text{new}}[\tilde{E}_{\text{new}}] = \begin{cases} E & \text{if } \hat{E} = \tilde{E}_{\text{new}}, \\ A[\tilde{E}_{\text{new}}] & \text{otherwise.} \end{cases}$$

Moreover, $\tilde{E}_{\text{new}} = \tilde{E}\langle E/A[\hat{E}]\rangle$. Combining, we obtain the desired representation of $A[\tilde{E}]$ in terms of old values, which is the result of the substitution. We obtain the following formula, where the definition by cases is expressed in terms of **if-then-else**:

$$A[\tilde{E}]\langle E/A[\hat{E}]\rangle = \textbf{if } \hat{E} = \tilde{E}\langle E/A[\hat{E}]\rangle \textbf{ then } E \textbf{ else } A[\tilde{E}\langle E/A[\hat{E}]\rangle].$$

So the definition of $E'\langle E/X\rangle$ for any expression E' is complete. When substitution is applied to a mathematical formula, then bound variables sometimes have to be renamed. One reason is well-known from mathematical logic: it must be avoided that a free variable of the expression substituted gets bound after the substitution. A new problem arises when a subscripted variable has to be replaced: in this case it must also be guaranteed that no bound variable of the formula before substitution coincides with a free variable of the subscript in the variable to be replaced.

With this extension of the substitution operation, rule (R2) remains valid in the original form. If a subscript at the left or right side of the assignment is out of range, then a run-time error occurs, and the rule does not claim anything according to our conventions before remark 1.3. (Recall that these conventions were modified only temporarily in section 3.2.) By contrast, (R2$'$) has to be modified. To be sure, the case were E contains a subscript out of range is covered by the clause $E{\downarrow}$ in the precondition. Indeed, $A[I]$ is undefined for $I \notin s_A$, and if E contains $A[I]{\uparrow}$, then E will also be undefined, with exceptions like (3.8). However, if X is a subscripted variable $A[E']$, then the additional requirements $E'{\downarrow}$ and $E' \in s_A$ are needed. So the modified version of (R2$'$) is

$$\text{(R2}a\text{)} \qquad \{E{\downarrow} \,\&\, E' \in s_A \,\&\, \psi\langle E/X\rangle\} \quad A[E'] := E\downarrow \quad \{\psi\}.$$

Note that $E' \in s_A$ implies $E'{\downarrow}$ according to our conventions.

Example 4.3: (i) Consider the formula

$$\psi\colon \quad (\exists x \in s_A)(x + 1 \in s_A \,\&\, A[x] = A[x+1] \,\&\, (\forall y)(y^2 \neq A[x])).$$

We wish to compute the formula $\phi \equiv \psi\langle y/A[x]\rangle$ such that

$$\{\phi\} \quad A[x] := y \quad \{\psi\} \tag{4.23}$$

holds. In the second clause the bound variable y has to be renamed, in order that the occurrence of y introduced by substitution remains free. Likewise, in the first clause the bound variable x has to be renamed to avoid a conflict with the subscripted variable. Indeed, it would be utter nonsense to carry out the naïve substitution in the form $(\exists x \in s_A)(x + 1 \in s_A \,\&\, y = A[x+1])$. After the bound variables have been replaced by z and u, the substitution yields

$$\begin{aligned}\phi\colon \quad (\exists z \in s_A)(&z + 1 \in s_A \\ &\& \,(\textbf{if } x = z \textbf{ then } y \textbf{ else } A[z]) = (\textbf{if } x = z+1 \textbf{ then } y \textbf{ else } A[z+1]) \\ &\& \,(\forall u)(u^2 \neq (\textbf{if } x = z \textbf{ then } y \textbf{ else } A[z]))).\end{aligned}$$

Write ϕ as $(\exists z \in s_A)(\phi')$. This formula can be simplified as follows. An element z satisfying ϕ' may be either $x - 1$ or x, or another element of s_A. If $z = x - 1$, then the expression **if** $x = z$ **then** y **else** $A[z]$ has the value $A[x-1]$, and **if** $x = z+1$ **then** y **else** $A[z]$ has the value y, so we must have $A[x-1] = y$. Similarly, if $z = x$, then $A[x+1]$ must equal y. In both cases y must not be the square of a number u. On the other hand, if $z \notin \{x-1, x\}$, then the expressions **if** $x = z$ **then** y **else** $A[z]$ and **if** $x = z+1$ **then** y **else** $A[z]$ equal $A[z]$ and $A[z+1]$ respectively, so the condition is that $A[z] = A[z+1]$ holds, along with other obvious clauses. Thus we get

$$\begin{aligned}\phi \equiv &[(\exists z \in \{x-1, x+1\})(z \in s_A \mathbin{\&} A[z] = y) \mathbin{\&} (\forall u)(u^2 \neq y)]\\ &\quad \vee (\exists z \in s_A - \{x, x+1\})(z+1 \in s_A \mathbin{\&} A[z] = A[z+1] \mathbin{\&} (\forall u)(u^2 \neq A[z])).\end{aligned}$$

The formula ϕ makes sense even for $x \notin s_A$. In this case the assignment leads to a run-time error, but (4.23), which asserts only partial correctness, remains true. In order to obtain the right precondition for total correctness, the clause $x \in s_A$ has to be included. Alternatively, this may be stated as a condition outside the specification, since no program variables are involved: 'If $x \in s_A$, then $\{\phi\}\ A[x] := y\!\downarrow\ \{\psi\}$ holds.'

(ii) Next we compute the correct precondition ϕ satisfying

$$\{\phi\} \quad A[A[2]] := 1 \quad \{A[A[2]] = 1\}. \tag{4.24}$$

It was observed above that **true** is not the correct ϕ. The rules of substitution yield

$$\begin{aligned}\phi &\equiv \big(A[A[2]]\langle 1/A[A[2]]\rangle\big) = 1\\ &\equiv \big(\textbf{if } A[2] = A[2]\langle 1/A[A[2]]\rangle \textbf{ then } 1 \textbf{ else } A[A[2]\langle 1/A[A[2]]\rangle]\big) = 1\\ &\equiv \big(\textbf{if } A[2] = (\textbf{if } A[2] = 2 \textbf{ then } 1 \textbf{ else } A[2]) \textbf{ then } 1\\ &\qquad\qquad \textbf{else } A[\textbf{if } A[2] = 2 \textbf{ then } 1 \textbf{ else} A[2]]\big) = 1.\end{aligned}$$

This formula can be simplified, but we deal with it no further. It was noted above that the postcondition $A[A[2]]$ fails after the assignment $A[A[2]] := 1$ if $A[1] = A[2] = 2$ holds previously. So in this case ϕ must fail as well, and, indeed, the evaluation of ϕ yields

$$\begin{gathered}\big(\textbf{if } 2 = (\textbf{if } 2 = 2 \textbf{ then } 1 \textbf{ else } 2) \textbf{ then } 1 \textbf{ else } A[\textbf{if } 2 = 2 \textbf{ then } 1 \textbf{ else } 2]\big) = 1\\ (\textbf{if } 2 = 1 \textbf{ then } 1 \textbf{ else } A[1]) = 1\\ 2 = 1\\ \textbf{false}\end{gathered}$$

Nested subscripts, as they appear in the last example, typically cause a substantial increase of the size of the formula obtained upon substitution. In practice they occur relatively rarely, however.

We now turn to the second method for dealing with an array A in program verification, which consists in treating A as a single variable. This has the advantage that no new proof rules and no extension of the substitution operator are necessary. However, the complexity of the formulas involved is now hidden in the mathematical properties of the function St to be introduced shortly; overall there is no simplification.

Assume that $A[E]$ occurs at the right side of an assignment statement, perhaps within a larger expression. Then it is no longer regarded as a subscripted variable, but as an expression obtained by applying a certain operator to A and E. What if $A[E]$

appears at the left side? Among the variables considered, A is of course the one which gets changed, so the assignment $A[E] := E'$ has to be translated into an assignment to A. The new value of A is obtained from the old values of A, E and E' in the form $\mathrm{St}(A, E, E')$. We define St as a mathematical function: $\mathrm{St}(a, i, e)$ is the result of replacing the component of the array a with index i by the value e. It is undefined if $i \notin s_a$. The crucial mathematical property of St is

$$\mathrm{St}(a, i, e)[j] = \textbf{if } i = j \textbf{ then } e \textbf{ else } a[j]. \tag{4.25}$$

Example 4.4: This example is somewhat less artificial than the previous ones; we illustrate both methods for handling arrays. Let $s_A = \{1, \ldots, n\}$, where n is a constant ≥ 1, and denote the array with components $a_1, \ldots, a_n$ by $(a_1, \ldots, a_n)$. The following program S shifts the elements of A cyclically, that is, a_1 is moved from the first position to the end:

$X := A[1]$;
for $I := 1$ **to** $n - 1$ **do**
$A[I] := A[I + 1]$;
A[n]:=X

Precondition and postcondition are given by

$$\phi \equiv A = (a_1, \ldots, a_n),$$
$$\psi \equiv A = (a_2, \ldots, a_n, a_1).$$

This representation of formulas is very easy to comprehend, but for substitutions and other manipulations a more precise representation using quantifiers is preferable and will be used from now on. We take the convention that a quantifier ranges over s_A whenever the quantified variable is used as a subscript for A. Thus we get

$$\phi \equiv (\forall i)(A[i] = a_i),$$
$$\psi \equiv (\forall i < n)(A[i] = a_{i+1}) \,\&\, A[n] = a_1.$$

As usual, we wish to show:

$$\{\phi\} \quad S \quad \{\psi\}. \tag{4.26}$$

Before the loop is executed for $I = i$, we have $A[j] = a_{j+1}$ for $j < i$ and $A[j] = a_j$ for $j \geq i$, so the loop invariant is

$$\rho\text{:} \quad X = a_1 \,\&\, (\forall j < I)(A[j] = a_{j+1}) \,\&\, (\forall j \geq I)(A[j] = a_j).$$

The first thing to show is that $\rho(1)$ holds before the loop is entered. Since s_A contains no $j < 1$, $\rho(1)$ simplifies to $X = a_1 \,\&\, (\forall j)(A[j] = a_j)$. The assignment rule yields

$$\{A[1] = a_1 \,\&\, (\forall j)(A[j] = a_j)\} \quad X := A[1] \quad \{\rho(1)\}.$$

Trivially, the precondition is equivalent to ϕ, so we get

$$\{\phi\} \quad X := A[1] \quad \{\rho(1)\}. \tag{4.27}$$

In order to apply the **for**-rule, we have to show:

$$\{\rho(I)\ \&\ 1 \leq I \leq n\} \quad A[I] := A[I+1] \quad \{\rho(I+1)\}. \tag{4.28}$$

Here $\rho(I+1)$ is the formula

$$X = a_1\ \&\ (\forall j < I+1)(A[j] = a_{j+1})\ \&\ (\forall j \geq I+1)(A[j] = a_j);$$

backward substitution yields

$$X = a_1\ \&\ (\forall j < I+1)\big((\textbf{if } I = j \textbf{ then } A[I+1] \textbf{ else } A[j]) = a_{j+1}\big)$$
$$\&(\forall j \geq I+1)\big((\textbf{if } I = j \textbf{ then } A[I+1] \textbf{ else } A[j]) = a_j\big).$$

This formula is equivalent to

$$X = a_1\ \&\ (\forall j < I)(A[j] = a_{j+1})\ \&\ A[I+1] = a_{I+1}\ \&\ (\forall j \geq I+1)(A[j] = a_j),$$

which in turn follows from $\rho(I)$. Thus (4.28) follows, and we can deduce

$$\{\rho(1)\} \quad S_1 \quad \{\rho(n)\}, \tag{4.29}$$

where S_1 is the **for**-statement. Here $\rho(n)$ simplifies to

$$X = a_1\ \&\ (\forall j < n)(A[j] = a_{j+1})\ \&\ A[n] = a_n.$$

It remains to show:

$$\{\rho(n)\} \quad A[n] := X \quad \{\psi\}. \tag{4.30}$$

Backward substitution from ψ yields formally

$$(\forall i < n)\big((\textbf{if } i = n \textbf{ then } X \textbf{ else } A[i]) = a_{i+1}\big)\ \&\ (\textbf{if } n = n \textbf{ then } X \textbf{ else } A[n]) = a_1,$$

which simplifies to

$$(\forall i < n)(A[i] = a_{i+1})\ \&\ X = a_1.$$

The latter condition is contained in $\rho(n)$, so (4.30) follows. From (4.27), (4.29) and (4.30), the desired specification (4.26) is deduced. It is easy to see that total correctness can be established similarly, with little extra writing.

What changes are involved in the proof if A is handled as a single variable? The loop invariant as well as specifications (4.27)–(4.30) remain as before. The only thing that changes is the form of the application of the assignment rule for (4.28) and (4.30); we provide details only for (4.28). The assignment $A[I] := A[I+1]$ translates into $A := \mathrm{St}(A, I, A[I+1])$, so backward substitution now yields

$$X = a_1\ \&\ (\forall j < I+1)(\mathrm{St}(A, I, A[I])[j] = a_{j+1})\ \&\ (\forall j \geq I+1)(\mathrm{St}(A, I, A[I])[j] = a_j).$$

Using (4.25), it is straightforward to show that this formula follows again from $\rho(I)$, similarly as in the preceding proof. Thus we can observe that the argument is essentially the same under both approaches; the differences are of a purely formal nature.

4.4. Example: connected components of a graph.

With the inclusion of structured data types, many more algorithms of real significance are possible. We present one of them, as indicated by the title of this section.

Consider a graph $G = (V, E)$, where V is the set of vertices, E the set of edges. Without loss of generality we may assume $V = \{1, \ldots, N\}$. E is given by an array of sets A, where $A[i]$ is the set of nodes which are adjacent to i, that is, $(i, j) \in E$. Since the size of arrays has to be determined in advance, we take s_A to be $\{1, \ldots, m\}$ for some constant $m \geq N$. Clearly, G is uniquely determined by N and A, but some additional conditions have to be satisfied in order that a given number N and array A really specify a graph. These conditions will be formalized shortly.

The goal is to compute an array of sets $C[1..L]$ such that $C[1], \ldots, C[L]$ are the connected components of G. The algorithm proceeds as follows:

(a) Set Z to V. Choose an element $I \in Z$ and set L to 1; component $C[1]$ will now be created. Initialize $C[L]$ and U to $\{I\}$; U is an auxiliary set whose significance will soon be clear.

(b) For all $J \in U$ include the elements of $A[J]$ in U. Then remove the old elements from U, that is, those which were already in $C[L]$, and include the new ones in $C[L]$. Thus U always contains those nodes which were newly added to $C[L]$. Repeat this step until $U = \emptyset$.

(c) Remove the elements of the new component $C[L]$ from Z, so Z contains the nodes not yet included in any component. If $Z \neq \emptyset$, then increase L by 1, choose $I \in Z$ and go back to (b).

This leads to the following program S. When $I \in Z$ has to be selected, this could be done randomly, but since we have not dealt with random choices, we pick the smallest element. We assume that an interval of the integers can be assigned to a set in one step. This is allowed in PASCAL, for instance, although the syntax is different.

$L := 0$; $Z := \{1, \ldots, N\}$;
while $Z \neq \emptyset$ do
    begin
        $L := L + 1$; $I := \min(Z)$; $C[L] := \{I\}$; $U := \{I\}$;
        repeat
            for $J \in U$ **do**
                $U := U \cup A[J]$;
            $U := U - C[L]$;
            $C[L] := C[L] \cup U$
        until $U = \emptyset$;
        $Z := Z - C[L]$
    end

As usual the values of N and A are denoted by n and a, and we also use v, e and g for the set of vertices, set of edges and graph as determined by n and a. Thus $v = \{1, \ldots, n\}$. The precondition is simply

$$\phi: \quad N = n \,\&\, A = a;$$

it remains valid during the whole computation.

Some further assumptions are also needed, but they are given as assertions involving only the *mathematical* variables n, a etc.; therefore they may be taken as a hypothesis underlying the whole proof, and they need not be included in the pre- and postcondition. Namely: (a) n is in the allowed range; we admit the case $n = 0$, when g is empty and 0 is the number of connected components. In this case the **while**-loop will not be executed. (b) e does indeed specify the set of edges for a graph which has v as its set of vertices. This means that $j \in a[i]$ is a symmetric relation between i and j, and the elements of $a[i]$ belong to v. These assumptions are summarized by the following formula:

$$0 \leq n \leq m \mathbin{\&} (\forall i \in s_A)(\forall j)(j \in a[i] \Longrightarrow j \in v \mathbin{\&} i \in a[j]). \tag{4.31}$$

In order to write down the postcondition ψ, we use the abbreviation $\mathrm{CC}(x)$ for the assertion 'x is a connected component of g.' (Of course, we could also write G rather than g; G is not a program variable, but may be taken as an abbreviation for the 'graph determined by N and E.') ψ contains the following additional assertions besides ϕ: the sets $C[i]$ are connected components of g and disjoint, and their union is v. Thus

$$\begin{aligned} \psi\text{:}\quad & \phi \mathbin{\&} (\forall i)(1 \leq i \leq L \Longrightarrow \mathrm{CC}(C[i])) \\ & \qquad \mathbin{\&} (\forall i,j)(1 \leq i < j \leq L \Longrightarrow C[i] \neq C[j]) \mathbin{\&} \bigcup_{i=1}^{L} C[i] = v. \end{aligned}$$

Here $C[i] \neq C[j]$ implies $C[i] \cap C[j] = \emptyset$, since two different connected components of a graph are disjoint. We will show:

$$\{\phi\} \quad S \quad \{\psi\}. \tag{4.32}$$

The following notation for segments of S will be used:

- S_1: sequence of two assignments before **while**-loop
- S_2: body of the **while**-statement
- S_3: sequence of four assignments at beginning of S_2, before **repeat**-**until**
- S_4: body of **repeat**-**until**-loop
- S_5: **for**-statement at beginning of S_4
- S_6: sequence of two assignments in S_4, after S_5

The invariant for the **while**-loop asserts almost the same as ψ. The only difference is that the union of the sets $C[i]$ is not v but $v - Z$, where $Z \subseteq v$:

$$\begin{aligned} \rho\text{:}\quad & \phi \mathbin{\&} (\forall i)(1 \leq i \leq L \Longrightarrow \mathrm{CC}(C[i])) \\ & \qquad \mathbin{\&} (\forall i,j)(1 \leq i < j \leq L \Longrightarrow C[i] \neq C[j]) \mathbin{\&} Z \subseteq v \mathbin{\&} \bigcup_{i=1}^{L} C[i] = v - Z. \end{aligned}$$

The first step in the verification is of course to establish

$$\{\phi\} \quad S_1 \quad \{\rho\}. \tag{4.33}$$

The two assignments in S_1 are independent of each other, so the corresponding substitutions can be carried out simultaneously. Working backwards from ρ we get formally

$$\begin{aligned}\phi \,\&\, (\forall i)(1 \leq i \leq 0 \Longrightarrow \mathrm{CC}(C[i])) \,\&\, (\forall i,j)(1 \leq i < j \leq 0 \Longrightarrow C[i] \neq C[j])\\ \&\, \{1,\ldots,N\} \subseteq v \,\&\, \bigcup_{i=1}^{0} C[i] = v - \{1,\ldots,N\}.\end{aligned}$$

Trivially, this formula is just ϕ, since a universal formula with the quantifier ranging over the empty set is true, ϕ contains the clause $N = n$, and we are given $v = \{1,\ldots,n\}$. Thus (4.33) is shown.

The main task is the show that ρ is indeed a loop invariant:

$$\{\rho \,\&\, Z \neq \emptyset\} \quad S_2 \quad \{\rho\}. \tag{4.34}$$

Then we can conclude

$$\{\rho\} \quad \textbf{while } Z \neq \emptyset \textbf{ do } S_2 \quad \{\rho \,\&\, Z = \emptyset\}. \tag{4.35}$$

From (4.33) and (4.35) we deduce the required assertion (4.32), since $\rho \& Z = \emptyset$ obviously implies ψ.

For the proof of (4.34) we have to find invariants associated with the **repeat**-**until**-loop. Which condition ρ_2 holds after every execution of S_4? For one thing, ρ holds with $L-1$ in place of L, for since the beginning of the current execution of S_2, L has been increased by 1, whereas $C[1], \ldots, C[L-1]$ and Z were left unchanged. We can write $\rho(L-1)$ for this condition, so $\rho(L)$ is ρ. Secondly, I is an element of Z, which implies by $\rho(L-1)$ that $I \in v - \bigcup_{i=1}^{L-1} C[i]$. All other elements included in $C[L]$ so far belong to the same connected component as I. Actually, ρ_2 need not include a statement about I; it suffices to assert that $C[L]$ is nonempty, contained in v, disjoint from all $C[i]$ with $i < L$, and that all elements of $C[L]$ belong to the same connected component. For the latter assertion another abbreviation is introduced: $\mathrm{CC1}(x)$ means that all elements of x belong to the same connected component of g, that is, any two elements of x can be connected by a path.

Yet another clause of ρ_2 states that $U \subseteq C[L]$. Finally, one more crucial assertion must be included. Consider a moment τ in the computation when some execution of S_4 has just been finished. It was mentioned before S was given that U contains the *new* elements of $C[L]$, those which were included in the most recent execution of S_4. Subsequently the only vertices to be included in $C[L]$ are those which are adjacent to elements of U; why does this suffice? The answer is as follows. If vertex k is adjacent to some $j \in C[L] - U$, then j was included in $C[L]$ in an *earlier* execution of S_4 (or by the assignment $C[L] := \{I\}$), and k was included in the following execution of S_4, which still precedes point τ. Hence $k \in C[L]$ at point τ, and there is no need to include k (or other vertices adjacent to j) later. So the last clause of ρ_2 asserts that every vertex adjacent to some $j \in C[L] - U$ already belongs to $C[L]$. Thus we get

$$\begin{aligned}\rho_2\colon \quad & \rho(L-1) \,\&\, \mathrm{CC1}(C[L]) \,\&\, \emptyset \neq C[L] \subseteq v \,\&\, (\forall i < L)(C[i] \cap C[L] = \emptyset)\\ & \&\, U \subseteq C[L] \,\&\, (\forall j \in C[L] - U)(\forall k \in A[j])(k \in C[L]).\end{aligned}$$

The proof rule for **repeat**-**until** requires another invariant ρ_1 which holds *before* every execution of S_4. At this point the additional condition $U \neq \emptyset$ holds, so we put

$$\rho_1: \quad \rho_2 \,\&\, U \neq \emptyset.$$

Actually, the verification would also work if ρ_1 were chosen to be identical with ρ_2. In general, a **repeat**-**until**-loop for which a single loop invariant suffices can be replaced by a **while**-loop which happens to be executed at least once; in this case the exit condition B has to be negated, of course.

For (4.34) we have to show

$$\{\rho \,\&\, Z \neq \emptyset\} \quad S_3 \quad \{\rho_1\}, \tag{4.36}$$

$$\{\rho_1\} \quad \textbf{repeat } S_4 \textbf{ until } U = \emptyset \quad \{\rho_2 \,\&\, U = \emptyset\}, \quad \text{and} \tag{4.37}$$

$$\{\rho_2 \,\&\, U = \emptyset\} \quad Z := Z - C[L] \quad \{\rho\}. \tag{4.38}$$

Specification (4.37) is of course given in the form which may be deduced by an application of (R11).

For the proof of (4.36), the postcondition ρ_1 has to be written out in full, except that ϕ may be left as an abbreviation. Indeed, no values are assigned to the program variables in ϕ, so ϕ is not affected by the substitutions. A look at the four assignments in S_3 shows that the last two are not independent of the first two, since they use the values of L and I, which were changed before. However, the first assignment is independent of the second, and the third is independent of the fourth. This means that backward substitution may be carried out in two steps. The first applies simultaneous substitution in correspondence with the third and fourth assignment and yields formally

$$\begin{aligned}
\phi \,\&\, & (\forall i)(1 \leq i \leq L-1 \Longrightarrow \mathrm{CC}(\textbf{if } L = i \textbf{ then } \{I\} \textbf{ else } C[i])) \\
& \&\, (\forall i,j)(1 \leq i < j \leq L-1 \\
& \qquad \Longrightarrow (\textbf{if } L = i \textbf{ then } \{I\} \textbf{ else } C[i]) \neq (\textbf{if } L = j \textbf{ then } \{I\} \textbf{ else } C[j])) \\
& \&\, Z \subseteq v \,\&\, \bigcup_{i=1}^{L-1} (\textbf{if } L = i \textbf{ then } \{I\} \textbf{ else } C[i]) = v - Z \\
& \&\, \mathrm{CC1}(\textbf{if } L = L \textbf{ then } \{I\} \textbf{ else } C[L]) \,\&\, \emptyset \neq (\textbf{if } L = L \textbf{ then } \{I\} \textbf{ else } C[L]) \subseteq v \\
& \&\, (\forall i < L)((\textbf{if } L = i \textbf{ then } \{I\} \textbf{ else } C[i]) \cap (\textbf{if } L = L \textbf{ then } \{I\} \textbf{ else } C[L]) = \emptyset) \\
& \&\, \{I\} \subseteq (\textbf{if } L = L \textbf{ then } \{I\} \textbf{ else } C[L]) \\
& \&\, (\forall j \in (\textbf{if } L = L \textbf{ then } \{I\} \textbf{ else } C[L]) - \{I\})(\forall k \in A[j]) \\
& \qquad\qquad\qquad\qquad (k \in (\textbf{if } L = L \textbf{ then } \{I\} \textbf{ else } C[L])) \\
& \&\, \{I\} \neq \emptyset.
\end{aligned}$$

In this formula each **if**-**then**-**else**-expression may first be replaced by $\{I\}$, $C[i]$ or $C[j]$, then trivially valid clauses are omitted. We obtain:

$$\begin{aligned}
\phi \,\&\, & (\forall i)(1 \leq i \leq L-1 \Longrightarrow \mathrm{CC}(C[i])) \,\&\, (\forall i,j)(1 \leq i < j \leq L-1 \Longrightarrow C[i] \neq C[j]) \\
& \&\, Z \subseteq v \,\&\, \bigcup_{i=1}^{L-1} C[i] = v - Z \,\&\, \mathrm{CC1}(\{I\}) \,\&\, I \in v \,\&\, (\forall i < L)(C[i] \cap \{I\} = \emptyset).
\end{aligned}$$

In the second step substitution corresponding to the first two assignments is performed:

$$\begin{aligned}
&\phi \,\&\, (\forall i)(1 \leq i \leq L \Longrightarrow \mathrm{CC}(C[i])) \,\&\, (\forall i,j)(1 \leq i < j \leq L \Longrightarrow C[i] \neq C[j]) \\
&\quad \& \, Z \subseteq v \,\&\, \bigcup_{i=1}^{L} C[i] = v - Z \,\&\, \mathrm{CC1}(\{\min(Z)\}) \,\&\, \min(Z) \in v \\
&\quad \& \, (\forall i < L+1)(C[i] \cap \{\min(Z)\} = \emptyset).
\end{aligned}$$

Denote this formula by ξ. All clauses except the last three are contained in $\rho \,\&\, Z \neq \emptyset$, and the last three follow trivially from $\emptyset \neq Z \subseteq v$ and $\bigcup_{i=1}^{L} C[i] = v - Z$. Note that $\min(Z)$ would be undefined if $Z = \emptyset$. Hence we have $\rho \,\&\, Z \neq \emptyset \Longrightarrow \xi$, completing the proof of (4.36).

Next we show (4.38), which is obviously simpler than (4.37). Backward substitution from ρ yields the following formula, which we denote by χ:

$$\begin{aligned}
&\phi \,\&\, (\forall i)(1 \leq i \leq L \Longrightarrow \mathrm{CC}(C[i])) \,\&\, (\forall i,j)(1 \leq i < j \leq L \Longrightarrow C[i] \neq C[j]) \\
&\quad \& \, Z - C[L] \subseteq v \,\&\, \bigcup_{i=1}^{L} C[i] = v - (Z - C[L]).
\end{aligned}$$

We have to show

$$\rho_2 \,\&\, U = \emptyset \Longrightarrow \chi, \tag{4.39}$$

so assume $\rho_2 \,\&\, U = \emptyset$. Then $Z - C[L] \subseteq v$, since $Z \subseteq v$ is a clause of $\rho(L-1)$. Also,

$$\bigcup_{i=1}^{L} C[i] = \left(\bigcup_{i=1}^{L-1} C[i]\right) \cup C[L] = (v - Z) \cup C[L] = v - (Z - C[L]),$$

using $C[L] \subseteq v$. The condition $C[i] \neq C[j]$ holds for $1 \leq i < j < L$ by $\rho(L-1)$, and for $1 \leq i < j = L$ it follows from $C[L] \neq \emptyset$ and $(\forall i < L)(C[i] \cap C[L] = \emptyset)$. Likewise, $\mathrm{CC}(C[i])$ holds for $1 \leq i < L$ by $\rho(L-1)$. Thus it remains to show $\mathrm{CC}(C[L])$. Now ρ_2 already asserts that all elements of $C[L]$ belong to the same connected component, so we have to prove that this component contains no further elements. Equivalently, every node k adjacient to some $j \in C[L]$ also belongs to $C[L]$. But this claim follows from the last clause of ρ_2, since we are given that $U = \emptyset$. Thus (4.39) and hence (4.38) follow.

In order to deduce (4.37) by (R11), we only have to show

$$\{\rho_1\} \quad S_4 \quad \{\rho_2\}, \tag{4.40}$$

since $\rho_2 \,\&\, U \neq \emptyset \Longrightarrow \rho_1$ is trivial. We have to find the last loop invariant, the one for S_5. (R16) has to be applied with u in place of s, where u represents the value of U before the execution of S_5. Then the loop invariant for S_5 is denoted by $\rho_3(t)$, where t ranges over subsets of u. What should $\rho_3(t)$ be? This formula reflects the situation after S_5 has been executed for all $J \in t$. In comparison with the situation at the beginning of the current execution of the **repeat**-**until**-loop, which is described by ρ_1, only U has changed. The old value of U is given by u, so ρ_1 still holds with u in place of U. Moreover, the current set U consists of the elements of u (old elements) and the elements of $A[j]$ with $j \in t$. Thus we get:

$$\rho_3(t){:}\quad \rho(L-1) \mathbin{\&} \mathrm{CC1}(C[L]) \mathbin{\&} \emptyset \neq C[L] \subseteq v \mathbin{\&} (\forall i < L)(C[i] \cap C[L] = \emptyset) \\ \mathbin{\&} \emptyset \neq u \subseteq C[L] \mathbin{\&} (\forall j \in C[L]-u)(\forall k \in A[j])(k \in C[L]) \mathbin{\&} U = u \cup \bigcup_{j \in t} A[j].$$

In order to apply (R16), we have to show:

$$\{\rho_3(t) \mathbin{\&} t \subset u \mathbin{\&} J \in u - t\} \quad U := U \cup A[J] \quad \{\rho_3(t \cup \{J\})\}. \tag{4.41}$$

U and t both occur only in the last clause of $\rho_3(t)$, even after the subformula $\rho(L-1)$ has been written out. Hence the last clause is the only one where $\rho_3(t)$ and $\rho_3(t \cup \{J\})$ differ, and the only one which is affected by the backward substitution according to the assignment in (4.41). In the formula $\rho_3(t \cup \{J\})\langle U \cup A[J]/U\rangle$, the last clause is

$$U \cup A[J] = u \cup \bigcup_{j \in t \cup \{J\}} A[j],$$

which follows from the last clause of $\rho_3(t)$ and $J \in s_A$. We claim that $J \in s_A$ follows from the precondition in (4.41). Indeed, assuming $\rho_3(t) \mathbin{\&} t \subset u \mathbin{\&} J \in u-t$, we have $J \in u \subseteq C[L] \subseteq v \subseteq s_A$. The remaining clauses of $\rho_3(t \cup \{J\})\langle U \cup A[J]/U\rangle$ are identical with the corresponding clauses of $\rho_3(t)$. Thus $\rho_3(t) \mathbin{\&} t \subsetneq u \mathbin{\&} J \in u-t$ implies $\rho_3(t \cup \{J\})\langle U \cup A[J]/U\rangle$, so (4.41) follows.

From (4.41) we obtain by (R16):

$$\{U = u \mathbin{\&} \rho_3(\emptyset)\} \quad S_5 \quad \{\rho_3(u)\}. \tag{4.42}$$

Here we wish to modify the postcondition in such a way that u does no longer play a rôle; this allows to substitute U for u in the precondition, as described at the end of chapter 1. Finding the right formula to replace $\rho_3(u)$ cannot be done mechanically, though; it is based on an understanding of the algorithm. The crucial condition effected by S_5 is that U subsequently contains exactly those nodes $k \notin C[L]$ which are adjacient to some $j \in C[L]$. This is asserted by

$$\pi'{:}\quad \Big(\bigcup_{j \in C[L]} A[j]\Big) - C[L] \subseteq U \subseteq \Big(\bigcup_{j \in C[L]} A[j]\Big) \cup C[L].$$

Now we get the following formula to replace $\rho_3(u)$ in (4.42), which resembles each of the formulas ρ_1–ρ_3, except π' is the new assertion about U:

$$\pi{:}\quad \rho(L-1) \mathbin{\&} \mathrm{CC1}(C[L]) \mathbin{\&} \emptyset \neq C[L] \subseteq v \mathbin{\&} (\forall i < L)(C[i] \cap C[L] = \emptyset) \mathbin{\&} \pi'.$$

We claim that π follows from $\rho_3(u)$. Thus assume $\rho_3(u)$; this formula already includes each clause of π except for the last. The last clause π' is equivalent to the assertion that every $k \notin C[L]$ belongs to U if and only if k belongs to $A[j]$ for some $j \in C[L]$. Assume first that $k \in U$. Then by $\rho_3(u)$, either $k \in u \subseteq C[L]$, or $k \in A[j]$ for some $j \in u \subseteq C[L]$, so we are done. Conversely, assume $k \in A[j]$ with $j \in C[L]$. If

$j \in u$, then $k \in U$ by $\rho_3(u)$. But if $j \in C[L] - u$, then the next to last clause of $\rho_3(u)$ implies $k \in C[L]$. Thus our claim follows, and by (R3) we get

$$\{U = u \mathbin{\&} \rho_3(\emptyset)\} \quad S_5 \quad \{\pi\}. \tag{4.43}$$

As noted above we may now substitute U for u in the precondition, so the clause $U = u$ disappears. The resulting formula is

$$\begin{aligned}&\rho(L-1) \mathbin{\&} \mathrm{CC1}(C[L]) \mathbin{\&} \emptyset \neq C[L] \subseteq v \mathbin{\&} (\forall i < L)(C[i] \cap C[L] = \emptyset)\\ &\qquad \mathbin{\&} \emptyset \neq U \subseteq C[L] \mathbin{\&} (\forall j \in C[L] - U)(\forall k \in A[j])(k \in C[L]) \mathbin{\&} U = U \cup \bigcup_{j \in \emptyset} A[j],\end{aligned}$$

which clearly follows from ρ_1. Hence (R3) may be applied again, this time for the precondition in (4.43), yielding

$$\{\rho_1\} \quad S_5 \quad \{\pi\}. \tag{4.44}$$

In order to complete the proof of (4.40), we need to show the following specification:

$$\{\pi\} \quad S_6 \quad \{\rho_2\}. \tag{4.45}$$

Here ρ_2 has to be written out in full before backward substitution is applied. This time the result of applying substitution to $C[\ldots]$ is simplified immediately, we do no longer write clumsy **if-then-else**-expressions. The first substitution step yields the following formula:

$$\begin{aligned}\lambda_1\colon\quad &\phi \mathbin{\&} (\forall i)(1 \leq i \leq L-1 \Longrightarrow \mathrm{CC}(C[i]))\\ &\mathbin{\&} (\forall i,j)(1 \leq i < j \leq L-1 \Longrightarrow C[i] \neq C[j]) \mathbin{\&} Z \subseteq v \mathbin{\&} \bigcup_{i=1}^{L-1} C[i] = v - Z\\ &\mathbin{\&} \mathrm{CC1}(C[L] \cup U) \mathbin{\&} \emptyset \neq C[L] \cup U \subseteq v \mathbin{\&} (\forall i < L)(C[i] \cap (C[L] \cup U) = \emptyset)\\ &\mathbin{\&} U \subseteq C[L] \cup U \mathbin{\&} (\forall j \in (C[L] \cup U) - U)(\forall k \in A[j])(k \in C[L] \cup U).\end{aligned}$$

λ_1 may be simplified by omitting the next to last clause (which is trivially true) and replacing $(C[L] \cup U) - U$ by $C[L] - U$. The second substitution step, where U is replaced by $U - C[L]$, has no other effect but to change $C[L] - U$ to $C[L]$, since $C[L] \cup (U - C[L]) = C[L] \cup U$ etc. The result is

$$\begin{aligned}\lambda\colon\quad &\phi \mathbin{\&} (\forall i)(1 \leq i \leq L-1 \Longrightarrow \mathrm{CC}(C[i]))\\ &\mathbin{\&} (\forall i,j)(1 \leq i < j \leq L-1 \Longrightarrow C[i] \neq C[j]) \mathbin{\&} Z \subseteq v \mathbin{\&} \bigcup_{i=1}^{L-1} C[i] = v - Z\\ &\mathbin{\&} \mathrm{CC1}(C[L] \cup U) \mathbin{\&} \emptyset \neq C[L] \cup U \subseteq v \mathbin{\&} (\forall i < L)(C[i] \cap (C[L] \cup U) = \emptyset)\\ &\mathbin{\&} (\forall j \in C[L])(\forall k \in A[j])(k \in C[L] \cup U).\end{aligned}$$

In order to establish (4.45), it remains to show the implication

$$\pi \Longrightarrow \lambda. \tag{4.46}$$

The first five clauses of λ are contained in π, namely, in $\rho(L-1)$. The clause $\mathrm{CC1}(C[L] \cup U)$ follows from $\mathrm{CC1}(C[L])$ and π', since π' asserts that every $k \in U - C[L]$ is adjacient to some $j \in C[L]$. Next, $\emptyset \neq C[L] \cup U \subseteq v$ follows from $\emptyset \neq C[L] \subseteq v$ and $U \subseteq v$, and $U \subseteq v$ follows from π' and from the formula (4.31), which is used as a hypothesis for the whole verification. $(\forall i < L)(C[i] \cap (C[L] \cup U) = \emptyset)$ follows from $(\forall i < L)(C[i] \cap C[L] = \emptyset)$ together with $(\forall i < L)(\mathrm{CC}(C[i]))$ and $\mathrm{CC1}(C[L] \cup U)$; namely, the sets $C[i]$ $(i < L)$ are connected components, and $C[L] \cup U$ is contained in a connected component which must be different from $C[i]$. Finally, the last clause of λ is again a consequence of π'.

This argument completes the proof for each of the assertions (4.46), (4.45), (4.40), (4.37), (4.34), (4.35) and, finally, (4.32).

Proving *total* correctness involves only straightforward modifications. For instance, every execution of S_2 decreases the cardinality of Z, so (R8′) is used with $|Z|$ in the rôle of E and $W = \omega$, as usual. Instead of (4.34) one has to show

$$\{\rho \,\&\, Z \neq \emptyset \,\&\, |Z| = z\} \quad S_2 \quad \{\rho \,\&\, |Z| < z\}.$$

The first hypothesis in (R8′) is trivially true, since $|Z|$ cannot be negative. In the **repeat**-**until**-loop one may take $\left|v - C[L]\right|$ as an expression whose value decreases with every execution of S_4.

5. Procedures and functions

5.1. Procedures.

We consider procedure declarations of the form

$$\textbf{procedure } \mathrm{Proc}(\textbf{var } X_1, \ldots, X_n; Y_1, \ldots, Y_m);\ S, \tag{5.1}$$

where $X_1, \ldots, X_n$ are the *formal variable parameters*, $Y_1, \ldots, Y_m$ the *formal value parameters* and S the *body* of the procedure Proc. The syntax is similar as in PASCAL, except that we omit the type declarations, as usual. With respect to the type of some parameter, the same conventions apply as for program variables in general: it is specified in each example, and it is **integer** by default.

A call of Proc has the form

$$\mathrm{Proc}(A_1, \ldots, A_n, E_1, \ldots, E_m), \tag{5.2}$$

where $A_1, \ldots, A_n$ are the *actual variable parameters*, $E_1, \ldots E_m$ the *actual value parameters*. The A_i must be variables, whereas E_i may be arbitrary expressions. Of course, the type of A_i or E_i must coincide with that of X_i or Y_i respectively. The call (5.2) is executed as follows: the local parameters Y_i are used like local variables which are initialized to (the original values of) E_i. Then S is executed with A_i substituted for X_i. Since Y_i serves as a local variable, assignments to Y_i do not affect any variable outside the procedure. In particular, there is no relationship between Y_i and a global variable which happens to be represented by the same identifier.

We briefly discuss the double use of identifiers, namely, the same X may represent a formal parameter and a global variable. In this case the global variable and the parameter still have to be regarded as different variables, and it must be clear in each context which of the two is denoted by X. Such multiple use of idenitifiers occurs quite often when programming is done in real life. As long as no recursive procedures and functions occur, it can (and usually will) be avoided simply by renaming formal parameters. This is no longer possible when Proc is recursive, that is, calls itself.

We may also declare *local* variables $Z_1, \ldots, Z_k$. In this case the procedure body S in (5.1) has to be preceded by

$$\textbf{local } Z_1, \ldots, Z_k$$

Then Z_i denotes the local variable within S, which is again unrelated to a global variable represented by the same identifier.

The use of local variables Z is again absolutely necessary only for recursive procedures and functions. For if Proc is recursive, then there may be several activations of Proc at the same time, and Z refers to a different address in each case. But if Proc is nonrecursive and we intend to use Z only locally, then we may simply avoid using the same identifier also for a global variable and in specifications about segments of the main program (or of the procedure which calls Proc). Local variable declarations will be omitted when procedure declarations (5.1) are discussed in general, but not in specific examples.

When programming is done in real life, then the use of local variables is of considerable practical importance: it allows to regard Proc as a *black box*, and when code for the main program is written, we need not worry about accidentally using an identifier which also occurs inside S.

Our goal is to provide proof rules for procedures. Such rules allow to deduce a specification about a procedure call (5.2) from a specification about the procedure body S. The simpliest interpretation (which is not always correct) is that (5.2) does with the actual parameters what S does with the formal ones. This leads to the following rule, which holds under certain conditions to be worked out subsequently, where the notation from (5.1) is assumed:

$$\text{(R23)} \qquad \frac{\{\phi\} \quad S \quad \{\psi\}}{\begin{array}{r}\{\phi\langle A_1/X_1,\ldots,A_n/X_n,E_1/Y_1,\ldots,E_m/Y_m\rangle\} \quad \mathrm{Proc}(A_1,\ldots,E_m) \\ \{\psi\langle A_1/X_1,\ldots,A_n/X_n,E_1/Y_1,\ldots,E_m/Y_m\rangle\}\end{array}}.$$

Actually, it is immediately clear that (R23) cannot be valid without any restrictions, since this rule does not even distinguish between variable- and value parameters! This consideration leads to the first condition for the application of (R23). If the procedure body S contains assignments to formal value parameters, then these assignments have only local effect. As a consequence, the following condition is needed for (R23) to hold:

(C1) S contains no assignment to a formal value parameter Y_i which also occurs in the postcondition ϕ.

Example 5.1: Consider the declaration

> **procedure** Proc(**var** $X; Y$);
> **begin** $Y := Y + 1$; $X := Y$ **end**

Let S be the body of this procedure. Clearly, we have

$$\{\mathbf{true}\} \quad S \quad \{X = Y\},$$

but the specification

$$\{\mathbf{true}\} \quad \mathrm{Proc}(A, B) \quad \{A = B\},$$

which might be deduced by (R23), does not hold. Indeed, if b is the value of B before this call, then the value of B remains b, whereas A is assigned the value $b + 1$.

We continue to use the notation from (5.1), with the X_i and Y_i as the formal variable- and value parameters, S as the body, and a procedure call in the form (5.2). The remaining restrictions for (R23) all have to do with multiple appearences of identifiers. Namely, the rule may fail if any of the following conditions is violated:

(C2) No two actual variable parameters A_i are identical.

(C3) No A_i occurs within an actual value parameter E_j.

(C4) No global variable used in S is identical with an actual variable parameter.

(C5) No global variable which appears in the pre- or postcondition ϕ or ψ in (R23) is identical with an actual variable parameter.

Example 5.2: (i) Consider the declaration

> **procedure** Proc(**var**$X; Y$);
> **begin** $X := Y + 1$; $C := X + 1$ **end**

Here C is a global variable. Again let S be the body of Proc. Then $\{\mathbf{true}\}\ S\ \{X > Y\}$ holds, but $\{\mathbf{true}\}$ Proc(A, A) $\{A > A\}$ does not hold. So in this case (R23) fails because (C3) does not hold.

(ii) The same conclusion as in (i) would also apply if Y were also declared as a variable parameter. (For this purpose the semicolon between X and Y in the procedure heading only has to be replaced by a comma.) In this case (C2) fails.

(iii) S also satisfies the specification $\{\mathbf{true}\}\ S\ \{C > X\}$, but the procedure call does not satisfy $\{\mathbf{true}\}$ Proc(C, B) $\{C > C\}$, where C is substituted for X. This time (C4) is violated.

(iv) Now let D be another global variable. Then $\{D = 0\}\ S\ \{D = 0\}$ holds, but $\{D = 0\}$ Proc(D, B) $\{D = 0\}$ does not hold. Here it is condition (C5) which fails to be satisfied.

If a procedure call violates one of the conditions (C2)–(C4), then obviously nothing can be done about it, and (R23) cannot be used. However, if (C2)–(C4) are satisfied, then (R23) can be applied by choosing ϕ and ψ appropriately, so as to meet (C1) and (C5) as well. Then the proof of a correct specification about the procedure call can usually be accomplished, possibly by combining the application of (R23) with that of other proof rules.

Example 5.3: The following procedure is somewhat less academic than the ones considered before, since it computes U^V for $V \geq 0$:

> **procedure** Exp$(U, V;$ **var** $X)$;
> **begin**
> $X := 1$;
>   **while** $V > 0$ **do**
>   **begin** $V := V - 1$; $X := X * U$ **end**
> **end**

Denote the body of Exp again by S.

(i) A specification which expresses that Exp does indeed compute U^V is

$$\{A = a \,\&\, B = b \geq 0\} \quad \mathrm{Exp}(A, B, Z) \quad \{Z = a^b\}. \tag{5.3}$$

According to (R23) we have to show

$$\{U = a \,\&\, V = b \geq 0\} \quad S \quad \{X = a^b\},$$

which is straightforward by the methods from chapter 1. Conditions (C1)–(C5) are clearly satisfied.

(ii) Now suppose we wish to show

$$\{B \geq 0\} \quad \mathrm{Exp}(A, B, Z) \quad \{(Z = A^B)\}, \tag{5.4}$$

a specification which is also correct, although probably less useful than (5.3). At first glance it appears that in order to deduce (5.4) by (R23), we have to show $\{V \geq 0\}\ S\ \{X = U^V\}$, but this specification is not correct, since S sets the value of V to zero. Nevertheless, a proof of (5.4) by (R23) can be accomplished, namely, (5.4) can be deduced from the following correct specification about S:

$$\{A = U \mathbin{\&} B = V \geq 0\} \quad S \quad \{X = A^B\}.$$

After substituting A and B for U and V, the resulting clauses $A = A$ and $B = B$ can of course be omitted. So the trick is to use actual value parameters even in the hypothesis for (R23), that is, in the formulas ϕ and ψ before substitution. In a similar fashion we may show an analogue of (5.3) which also asserts that A and B are left unchanged, that is, the postcondition contains the additional clauses $A = a$ and $B = b$.

Perhaps the main shortcoming of (R23) is restriction (C3), for in practice it is by no means unusual that a variable- and a value parameter coincide. Therefore we now present a modified proof rule which avoids this deficiency. The execution of the call $\mathrm{Proc}(A_1, \ldots, A_n, E_1, \ldots, E_m)$ was already described above, after (5.2). Let e_i be the value of E_i prior to this procedure call. Then the procedure body S is executed with Y_i initialized to e_i and with A_i in place of X_i. Assume that we can show that ψ holds after execution of S under the condition that before the execution ϕ holds and Y_i has value e_i for $i = 1, \ldots, m$. Then a corresponding specification is satisfied by the procedure call with A_i substituted for X_i, provided that e_i is the value of E_i before execution. The proof rule resulting from this consideration is

$$\text{(R24)} \qquad \frac{\{\phi \mathbin{\&} Y_1 = e_1 \mathbin{\&} \cdots \mathbin{\&} Y_m = e_m\} \quad S \quad \{\psi\}}{\begin{array}{c}\{\phi\langle A_1/X_1, \ldots, A_n/X_n\rangle \mathbin{\&} E_1 = e_1 \mathbin{\&} \cdots \mathbin{\&} E_m = e_m\} \\ \mathrm{Proc}(A_1, \ldots, A_n, E_1, \ldots, E_m) \quad \{\psi\langle A_1/X_1, \ldots, A_n/X_n\rangle\}\end{array}}$$

What conditions are needed for this rule to be valid? For one thing, the following condition must hold, which is even stronger than (C1):

(C1′) The formulas ϕ and ψ do not contain formal value parameters.

The reason is that a formal value parameter in ϕ or ψ would still appear in the conclusion in (R24), since it is not eliminated by the substitution. This would not make sense, of course. However, the consequences of this restriction are essentially the same as those of (C1) for (R23). True, ϕ must not contain a formal value parameter Y_i, but ϕ may use e_i instead. Similarly, if Y_i has not been assigned a value, then ψ may use e_i instead of Y_i, since Y_i still has value e_i after the execution of S.

The latter assertion does no longer hold if Y_i has been assigned a value, so we really obtain a restriction analogous to (C1). This restriction is unavoidable. Indeed, if ψ contains a clause like $Y_i > 0$ and Y_i has been assigned a new value by S, then no corresponding clause can be deduced in the conclusion, since the final value of Y_i (after execution of S) does not correspond to the value of any expression after the execution of the procedure call.

In addition to (C1′), restrictions (C2), (C4) and (C5) also remain valid for (R24). This is again illustrated by parts (ii)–(iv) of example 5.2. In parts (iii) and (iv) we may make Y a variable parameter, as it was done in part (ii). Then no value parameters are left, and an application of (R24) does no longer differ from an application of (R23).

On the other hand, as noted above, restriction (C3) does no longer apply. Indeed, the informal justification for (R24) remains unaffected if (C3) were violated, and it is easy to see that the situation from example 5.2 (i) can no longer occur.

Example 5.4: Consider the declaration

procedure Gcd(X, Y; **var** U);
local Z;
begin
  while $Y \neq 0$ **do**
    begin
      $Z := X \bmod Y$;
      $X := Y$;
      $Y := Z$;
    end;
  $U := |X|$
end

Denote the procedure body once again by S; it is the same as the program from example 1.2, except that the result is ultimately assigned not to X, but to the variable parameter U. Analogously to (1.17) we can show:

$$\{X = x \;\&\; Y = y\} \quad S \quad \{U = \gcd(x, y)\}. \tag{5.5}$$

Using (R23), we obtain readily:

$$\{A = x \;\&\; B = y\} \quad \mathrm{Gcd}(A, B, C) \quad \{C = \gcd(x, y)\}.$$

However, if we wish to prove

$$\{A = x \;\&\; B = y\} \quad \mathrm{Gcd}(A, B, B) \quad \{B = \gcd(x, y)\}, \tag{5.6}$$

then only (R24) can be used, since condition (C3) is violated. In this application of (R24), ϕ is **true** (empty conjuction of clauses), whereas ψ is the formula $U = \gcd(x, y)$.

We have seen that there are ways to cope with conditions (C1), (C3) and (C5), and that these conditions usually do not prevent correct specifications about procedure calls from being proven. But what if (C2) or (C4) fails? In this case there is no satisfactory remedy, and no rule like (R23) or (R24) can be used. Indeed, a procedure call with two identical actual variable parameters may have an effect which is totally unrelated to the effect of the execution of the procedure body S, which involves two distinct formal variable parameters.

Example 5.5: Let S_1, S_2 be program segments satisfying arbitrary specifications of the form

$$\{\textbf{true}\} \quad S_1 \quad \{\psi_1\}, \qquad \{\textbf{true}\} \quad S_2 \quad \{\psi_2\}.$$

Consider the declaration

procedure Proc(**var** X, Y);
begin
  $X := Y + 1$;
  if $X = Y$ **then** S_1 **else** S_2
end

For simplicity we assume that S_1, S_2, ψ_1 and ψ_2 do not contain X and Y, that is, the formulas ψ_i are assertions about global variables, not about parameters in the following procedure. The parameters matter only inasmuch as they determine whether S_1 or S_2 is executed. The body of this procedure is denoted by S, as usual. Clearly, we have

$$\{\mathbf{true}\} \quad S \quad \{\psi_2\},$$

since the equality $X = Y$ cannot hold after the assignment $X := Y + 1$, so an execution of S is tantamount to an execution of S_2. By contrast, we have

$$\{\mathbf{true}\} \quad \mathrm{Proc}(A, A) \quad \{\psi_1\},$$

since the execution of this procedure call amounts to an execution of S_1. There need not be any relation whatsoever between S_1 and S_2, and, accordingly, between ψ_1 and ψ_2.

We may modify this example by making Y a global variable rather than a parameter, and considering the procedure call $\mathrm{Proc}(Y)$. Again the effect of the procedure call is unrelated to that of S, but this time (C4) is violated.

But, surely, it must be possible to prove a specification about a procedure call where (C2) or (C4) fails? Yes indeed. For this purpose we have to replace the call by a program segment S' which does not use the procedure, whereupon the usual verification methods are applied to S'. Given the declaration (5.1), the translation S' of the call (5.2) is

$Y_1 := E_1;\ \ldots;\ Y_m := E_m;$
$S\langle A_1/X_1, \ldots, A_n/X_n\rangle$

This approach—rewriting the program in order to avoid some construct which is hard to handle—was already mentioned at the beginning of chapter 4, after the statement of (R15). It can always be applied as a last resort. However, it is much less satisfactory than an application of (R23) or (R24), since it violates the black box-principle. Consider a team of programmers, and assume that A writes a procedure proc, B another procedure pr2 which calls proc and perhaps other procedures. Then, ideally, B should be able to do his work, and also to verify pr2, without looking into the code written by A. A look at the parameters and at a specification for the body of proc should be sufficient, but in some cases they are not.

5.2. Functions.

The syntax we choose for function declarations is as follows:

$$\textbf{function}\ \mathrm{Func}(X_1, \ldots, X_n);\ \textbf{begin}\ S;\ \textbf{return}(E)\ \textbf{end} \tag{5.7}$$

As for procedures, the (value) parameters X_i behave like local variables. When $\mathrm{Func}(E_1, \ldots, E_n)$ is computed, then S is executed with X_i initialized to the value of E_i, and $\mathrm{Func}(A_1, \ldots, A_n)$ is the value of E upon termination, as indicated by the **return**-statement. An example was the declaration of Fact at the beginning of section 1.3. That declaration was recursive, however, and recursion will be dealt with only in the next section.

A call of Func is not an instruction by itself, but occurs as an expression or within an expression, as in the assignment $U := Y - \mathrm{Func}(U)$. Hence verification rules have a different objective than for procedures: we wish to derive mathematical properties of the function defined by the declaration. We restrict ourselves to the ideal case, where the following condition holds:

(†) The call $\mathrm{Func}(E_1, \ldots, E_n)$ has no other effect than to return a value which depends only on the values of the actual parameters E_i.

If (†) is met, then Func may be treated like any any operation, such as +, sin etc.

In real programming languages (†) is not necessarily satisfied. Namely, the body of the function declaration S may contain assignments to global variables or variable parameters, whereupon a call of Func changes the values of these global variables or of the corresponding actual variable parameters. These are called *side effects*. For instance, we may wish to determine the number of calls of Func, and for this purpose S may include the assignment $C := C + 1$ for a global counter C. In almost all cases, however, side effects are undesirable and a symptom of bad programming practice, and we will rule them out altogether.

Needless to say, side effects complicate the verification of programs considerably. For instance, how should we deduce a specification for an assignment $U := \hat{E}$, where $\hat{E}$ contains a call of Func? The assignment rule does not take side effects into account. The only way to proceed is to translate this assignment statement into a program segment P which avoids the call of Func, compare the last two paragraphs of the preceding section. P consists of the following:

(i) Initializations of the formal value parameters of Func according to the corresponding actual parameters.
(ii) The function body S with formal variable parameters replaced by actual ones.
(iii) The assignment $Z := E$, where Z is an auxiliary variable to store the value of $\mathrm{Func}(E_1, \ldots, E_n)$.
(iv) The assignment $U := \tilde{E}$, where $\tilde{E}$ is obtained from $\hat{E}$ by replacing $\mathrm{Func}(E_1, \ldots, E_n)$ by Z.

If E contains several function calls with side effects, then the translation is even more complicated, but it is clear how it works.

Condition (†) also fails if S contains assignments where global variables occur at the right side. True, this does not cause side effects, but the value of $\mathrm{Func}(E_1, \ldots, E_n)$ does no longer depend only on the values of the E_i.

In order to guarantee (†), we take the following conventions. No variable parameters are allowed in function declarations, and no global variable may occur in S. This means also that a variable Y in S does not occur in specifications for segments of the main program. As a consequence, the declaration of Func is completely independent of the main program.

There is yet another restriction, which is already contained in our description of the syntax of function declarations. Namely, we allow the **return**-statement only to occur at the *end* of the function body. This, again, is necessary to make reasonable verification rules possible. Why? A **return**-statement which occurs somewhere in the middle of S would not only associate the value of the function with that of an expression, but it also signals that the execution of S should be terminated, which is tantamount to a jump to the end of S. However, jumps, that is, **goto**-statements, are hard to cope with in program verification, and they are not dealt with in these lecture notes.

Next we turn to proof rules associated with functions, so consider a declaration of the form (5.7). We wish to deduce an assertion about Func from a specification $\{\phi\}\ S\ \{\psi\}$. How can this be done, and what form should ϕ andψ have? Certainly, ϕ should be regarded as a condition on the parameters, and ϕ should contain no other program variables. It would be pointless in view of condition (†) to include global variables in ϕ. The inclusion of local variables is also pointless, since no convention about their initialization was taken. Namely, the effect of the execution of S and the resulting value of $\mathrm{Func}(X_1, \ldots, X_n)$ should not depend on the initial values of local variables.

The postcondition ψ, on the other hand, should relate the values of the parameters (arguments) with that of Func, and the latter is given by E. The value of argument X_i may be represented by X_i itself, provided that S contains no assignment to it. Otherwise this argument has to be represented by a mathematical variable x_i, with the clause $X_i = x_i$ included in ϕ. We could provide two distinct proof rules correponding to the two cases, but it is simpler to use x_i in any case for the original value of X_i.

Local variables including formal value parameters may of course be contained in E, but no other occurrence in ψ makes sense. Accordingly we write $\psi(E)$, which means that ψ is really of the form $\psi'\langle E/z\rangle$, where z is an auxiliary mathematical variable, and no program variables occur in ψ'.

Finally, the property of Func to be deduced is the following: if the values x_i of the parameters X_i satisfy condition ϕ, and if the execution of S terminates, then $f(X_1, \ldots, X_n)$, which is the value of E after execution of S, satisfies ψ. The condition that the execution of S terminates is equivalent to $f(X_1, \ldots, X_n)\downarrow$. Accordingly, proof rules can be given as follows:

$$\text{(R25)} \qquad \frac{\{\phi \;\&\; X_1 = x_1 \;\&\; \cdots \;\&\; X_n = x_n\} \quad S \quad \{\psi(E)\}}{\phi \;\&\; \mathrm{Func}(x_1, \ldots, x_n)\downarrow \Longrightarrow \psi(\mathrm{Func}(x_1, \ldots, x_n))},$$

$$\text{(R25}'\text{)} \qquad \frac{\{\phi \;\&\; X_1 = x_1 \;\&\; \cdots \;\&\; X_n = x_n\} \quad S\downarrow \quad \{\psi(E)\}}{\phi \Longrightarrow \psi(\mathrm{Func}(x_1, \ldots, x_n))}.$$

Example 5.6: Once again we consider exponentiation, this time using a function rather than a procedure (which is in fact more appropriate). The declaration, in analogy to example 5.3, is

```
function Exp(U, V);
local X;
begin
    X := 1;
    while V > 0 do
        begin V := V − 1; X := X * U end;
    return(X)
end
```

Again we denote the body of this function by S. What we wish to show is

$$v \geq 0 \Longrightarrow \mathrm{Exp}(u, v) = u^v. \tag{5.8}$$

For this purpose we have to show:

$$\{U = u \,\&\, V = v \geq 0\} \quad S{\downarrow} \quad \{X = u^v\}, \tag{5.9}$$

whereupon (R25′) yields (5.8). (5.9) is the same as the specification needed in example 5.3 (i), except that different mathematical variables are used.

5.3. Recursion.

So far only nonrecursive procedures and functions were considered in our examples, though we did not state that this is a condition for a correct application of (R23)–(R25′). As a matter of fact, those proof rules remain formally correct for recursive procedures and functions, but they are no longer of any use. Indeed, consider again a declaration of the form (5.1), but assume this time that it is recursive, which means that the procedure body S in turn contains calls of Proc. The proof rules allow to deduce a specification for a procedure call from a specification for S, but in order to proof the latter, one has to show specifications for the individual instructions of S, and, in particular, for the recursive calls of Proc contained in S. In this way the task of carrying out the verification would lead to an infinite cycle.

For this reason a refined proof rule is required. In the following discussion let Z, Z' etc. denote a parameter list for Proc, so Proc(Z) is a typical procedure call. Informally, we make the following

Claim: In order to establish the correctness of a procedure call, it suffices to prove it *under the assumption that each recursive call induced by it is correct.*

This claim can be justified as follows. The set of all calls of Proc which are induced directly or indirectly by Proc(Z) form a tree in a natural way. Namely, Proc(Z'') is a son of Proc(Z') iff Proc(Z') directly induces Proc(Z''), that is, Proc(Z'') is one of the instructions carried out in the execution of Proc(Z'). The root of this tree is Proc(Z) itself. For any call Proc(Z') let $L(Z')$ be the depth of the subtree with root Proc(Z').

Now if the hypothesis in the claim is satisfied, then the correctness of each call Proc(Z') follows by induction on $L(Z')$. If $L(Z') = 0$, then Proc(Z') does not induce any further procedure call, the requirement that each induced procedure call is correct holds vacuously, and it follows that Proc(Z') yields the correct result. Now assume that the assertion holds for every call Proc(Z'') with $L(Z'') < n$. If $L(Z') = n$, then every call Proc(Z'') induced by Proc(Z') satisfies $L(Z'') < n$, so Proc(Z'') is correct by induction hypothesis, hence the hypothesis in the claim implies that Proc(Z') is correct as well. This completes the induction step, and we can conclude that Proc(Z), in particular, is correct, too.

Throughout the preceding discussion, ‘correctness’ mean partial correctness. Indeed, it is a well-known fact that recursive procedures are one of those constructs which may cause nontermination of a program. How *total* correctness can be shown in specific cases will be briefly discussed later.

All these considerations apply to functions as well. As an example we consider a recursive declaration of the function Fib(N) which returns the N-th Fibonacci number for $N \geq 0$. As is well-known, this is a very inefficient way to compute Fib(N) for large N, since the number of recursive calls increases exponentially. Therefore the example serves only as an illstration.

Example 5.7: Let Fib be declared by

> **function** Fib(N);
> **local** X;
> **begin**
>   **if** $N \leq 1$
>   **then** $X := N$
>   **else** $X :=$ Fib$(N-2)$ + Fib$(N-1)$;
>   **return**(X)
> **end**

The tree of recursive calls induced by Fib(4) looks as follows:

$$
\begin{array}{ccccccc}
 & & & \mathrm{Fib}(4) & & & \\
 & \mathrm{Fib}(2) & & & & \mathrm{Fib}(3) & \\
\mathrm{Fib}(0) & & \mathrm{Fib}(1) & & \mathrm{Fib}(1) & & \mathrm{Fib}(2) \\
 & & & & & \mathrm{Fib}(0) & \quad \mathrm{Fib}(1)
\end{array}
$$

In the notation above we have $L(2) = 1$, $L(3) = 2$, $L(4) = 3$, and, generally, $L(n) = n-1$ for $n \geq 1$. In order to establish correctness, one has to show that Fib(0) and Fib(1) yield the right result—in this case no recursive calls are induced—and that for $N \geq 2$, Fib(N) yields the right result provided that Fib($N-2$) and Fib($N-1$) do.

So the idea for the verification of recursive procedures and functions has been sketched; how can it be formalized? For this purpose we have to extend the kind of specification considered so far, and to introduce the notion of *conditional correctness.* Let S be a program segment, and $S_1, \ldots, S_k$ occurrences of smaller program segments within S. For instance, S could be of the form

if B **then** S_1 **else begin** S_2; S_3 **end**.

The notation

$$[\{\phi_1\} \quad S_1 \quad \{\psi_1\}] \,\&\cdots\&\, [\{\phi_k\} \quad S_k \quad \{\psi_k\}] \rightarrow [\{\phi\} \quad S \quad \{\psi\}] \tag{5.10}$$

means: If ϕ holds before the execution of S, then ψ holds upon termination, *provided that every execution of some S_i carried out as part of the execution of S meets the specification $\{\phi_i\}$ S_i $\{\psi_i\}$*. We call this a *conditional specification.*

Using the notation from definition 2.4, and recalling in particular part (ii) of that definition, this assertion can be made somewhat more precise as follows: Let E be an environment satisfying $\phi[E]$, and assume that the execution of S is started from environment E, $M(S,E)\downarrow$, and that for every environment occurring in this execution just before the execution of some S_i we have $\phi_i[E'] \Longrightarrow \psi_i[M(S_i,E')]$, then $\psi[M(S,E)]$ follows. (Note that in this context $M(S_i,E')\downarrow$ follows from $M(S,E)\downarrow$; if the execution of S terminates, then every execution of some S_i which is part of it must terminate as well.)

A complete formalization of the meaning of (5.10), and also of a similar assertion involving total correctness, is possible, using the methods from chapter 2, but we will not go through the details. Note that (5.10) is a stronger assertion than the logical implication

$$[\{\phi_1\} \quad S_1 \quad \{\psi_1\}] \,\&\cdots\&\, [\{\phi_k\} \quad S_k \quad \{\psi_k\}] \Longrightarrow [\{\phi\} \quad S \quad \{\psi\}] \tag{5.11}$$

Indeed, (5.11) is true if any of the specifications $\{\phi_i\}\ S_i\ \{\psi_i\}$ fails, that is, if there exists any environment E' such that $\phi_i[E]$ holds but $\psi[M(S_i,E')]$ fails, even if this E' cannot occur before S_i in the course of a computation of S. Note also that the S_i in (5.10) are not just sequences of instructions, but specific occurrences of such sequences within S. Therefore several of the S_i in (5.10) may be syntactically identical, although they may have to satisfy different specifications $\{\phi_i\}\ S_i\ \{\psi_i\}$ at different places within S.

Example 5.8: Consider the following program S:

$$X := X + Y;\ Y := 2;\ X := X + Y$$

Let S_1 and S_2 be the first and second occurrence of the assignment $X := X + Y$. Trivially, we have

$$[\{X \geq 0\} \quad S_1 \quad \{X \geq 2\}] \,\&\, [\{X \geq 2\} \quad S_2 \quad \{X \geq 4\}] \rightarrow [\{X \geq 0\} \quad S \quad \{X \geq 4\}]. \tag{5.12}$$

Although the specification $\{X \geq 2\}\ S_2\ \{X \geq 4\}$ by itself is not correct, it is satisfied by S_2 whenever it is executed as part of S. If the same were true for S_1, then the unconditional specification $\{X \geq 0\}\ S\ \{X \geq 4\}$ could be deduced, but this is not necessarily the case. However, the conditional specification (5.12) holds in any case.

Actually, (5.10) is not yet the most general kind of conditional specification. To the left of $\rightarrow$ there may appear, in addition to specifications about segments S_i, assertions about the result of function calls, say

$$\phi_i(X_1,\ldots,X_n) \Longrightarrow \psi_i(X_1,\ldots,X_n,\mathrm{Func}(X_1,\ldots,X_n)); \tag{5.13}$$

then one of the conditions for the correctness of S is that each function call $\mathrm{Func}(X_1,\ldots,X_n)$ occurring as part of the execution of S meets (5.13). This kind of condition is needed for the verification of recursive function declarations. Once again, $\mathrm{Func}(X_1,\ldots,X_n)$ refers to a specific *occurrence* of a function call within S; different function calls, even with the same parameter list, may have to meet different conditions of the form (5.13).

Finally, among the hypothesis to the left of $\rightarrow$, there may also be assertions of the form

$$\tau_i\colon \phi_i, \tag{5.14}$$

where ϕ_i is simply a formula relating the values of the variables, and τ is a location in S, just before or after some instruction. (5.14) is the kind of assertion often contained in comments about algorithms: "At this point the relation ϕ_i holds between the values of the variables." But whereas such a comment is meant to be a definitively true claim, (5.14) serves only as a hypothesis within a conditional specification for S.

If S is a call of a procedure Proc, then the program segments S_i at the left side of (5.10) occur not within S itself (which is just one instruction), but within the body S' of the procedure declaration. In this case every execution of S consists of an execution of S' with appropriate substitutions for the parameters. The condition for the validity $\{\phi\}\ S\ \{\psi\}$ is that each execution of some S_i, as part of the execution of S, meets $\{\phi_i\}\ S_i\ \{\psi_i\}$, with formal variable parameters replaced by actual ones.

There are also conditional specifications about functions, where the assertion to the right of $\rightarrow$ in (5.10) is of the form (5.13):

$$\phi(X_1,\ldots,X_n) \Longrightarrow \psi(X_1,\ldots,X_n,\mathrm{Func}(X_1,\ldots,X_n)).$$

Once again the program segments S_i at the left side are contained within the body of the function declarations.

Now, is it necessary to develop the theory of conditional correctness from scratch, as it was done for unconditional correctness? Fortunately not. The fact is that each of the proof rules presented can be modified to provide a conditional specification, and that is all we need. For instance, the rule for conditional statements (R7) can be rewritten in the form

$$[\{\phi\,\&\,B\}\quad S\quad \{\psi\}]\,\&\,[\tau\colon \phi\,\&\,\neg B \Longrightarrow \psi] \rightarrow [\{\phi\}\quad \textbf{if } B \textbf{ then } S\quad \{\psi\}],$$

where τ is the point just before the conditional. In applications one always proves that the formula $\phi\,\&\,\neg B \Longrightarrow \psi$ is generally valid, which is a stronger assertion than $\tau\colon \phi\,\&\,\neg B \Longrightarrow \psi$. For this reason hypothesis of the form (5.14) are not really needed for the verification of recursive procedures and functions.

We do not present a complete system of axioms which allows to formalize all logical inferences involving conditional specifications. The main point is as follows. Let σ be a specification involving a program S, and let $\sigma_1,\ \ldots,\ \sigma_k$ be specifications or assertions like (5.13) or (5.14) involving segments of S, as in (5.10). Then $\sigma_1\,\&\cdots\&\,\sigma_k \rightarrow \sigma$ can be inferred whenever σ can be deduced from $\sigma_1,\ \ldots,\ \sigma_k$ by repeatedly applying the proof rules in the usual way, obtaining specifications for a program segment S' from specifications for its constituents.

Finally, we are in a position to provide a proof rule for recursive procedures. Consider once again a declaration of the form

$$\textbf{procedure } \mathrm{Proc}(\textbf{var } X_1,\ldots,X_n;Y_1,\ldots,Y_m);\ S$$

(which was (5.1)), and assume that S contains recursive calls S_i of the form $\mathrm{Proc}(A_{i,1},\ldots,A_{i,n},E_{i,1},\ldots,E_{i,m})$. What we wish to show is a specification of the form

$$\sigma\colon\quad \{\phi\}\quad S\quad \{\psi\}, \tag{5.15}$$

which allows to deduce specifications for procedure calls (including S_i) by rules (R23)–(R25′). Conversely, in order to obtain (5.15), one needs to use specifications

$$\sigma_i\colon\quad \{\phi_i\}\quad S_i\quad \{\psi_i\}\qquad (i=1,\ldots,k). \tag{5.16}$$

Then we get (with notation as above):

$$\text{(R26)} \qquad \frac{\sigma_1 \,\&\, \cdots \,\&\, \sigma_k \to \sigma, \sigma \to \sigma_1, \ldots, \sigma \to \sigma_k}{\{\phi\} \quad S \quad \{\psi\}}$$

Precisely this rule applies to function declarations as well, except that σ_i is of the form (3.13) in this case.

When (R26) is applied, then it is important to recall that local variables and formal value parameters of Proc have their own version for each activation of Proc, and to distinguish between different versions, even though they are represented by the same parameter. (See the discussion of section 5.1.) Let X be such a local variable or formal parameter, and use X_i to denote the version of X associated with an activation A_i. Consider one of the hypothesis

$$\sigma \to \sigma_i \tag{5.17}$$

in (R26). Here σ_i is a specification for an activation A_1 of the form S_i which was induced by another activation A_0. Any occurrence of X in ϕ_i or ψ_i represents X_0, since σ_i is a statement about the effect of the procedure call within A_0. The actual parameters in S_i, too, are associated with A_0.

However, the procedure body S, for which σ provides a specification, is associated with A_1, and so are occurrences of X within S. An occurrence of X within ϕ or ψ will usually also represent X_1, and the transition from σ to σ_1 amounts to an appropriate parameter substitution according to one of the rules (R23)–(R25′). Nevertheless, it may also happen that ϕ and ψ contain occurrences of the global X_0. Specifically, it is clear that X_0 cannot be changed by the execution of S (since any occurrence of X in S means X_1), and the specification σ may contain a clause which asserts this fact. In this case a notational distinction between X_0 and X_1 is called for; obviously, σ remains valid if X_0 is replaced by any variable not occurring in S. All this will be illustrated in example 5.10.

In the other hypothesis in (R26), namely

$$\sigma_1 \,\&\, \cdots \,\&\, \sigma_k \to \sigma, \tag{5.18}$$

the relationship between σ and σ_i is different. This time σ_i has to be regarded as a specification for an activation of Proc from within S. If S refers to the instantiation of the procedure body induced by activation A_1 with appropriate parameter substitutions (as it was the case in the discusion of (5,17) above), then the procedure call S_i to which σ_i refers in (5.18) is a new activation A_2, and an occurrence of X in ϕ_i or ψ_i refers to X_1 (just as it referred to X_0 in (5.17)).

We present two examples, first a simple one for functions, then a more complicated one involving a procedure. The function we consider was already declared earlier, now we carry out the verification.

Example 5.9: Let Fib be declared as in example 5.7, and let S be the body of this declaration (without the **return**-statement). For the time being we write $\mathrm{fib}(n)$ for the n-th Fibonacci number; we may use $\mathrm{Fib}(n)$ for this purpose only after the correctness of the function declaration has been established. We wish to show:

$$n \geq 0 \,\&\, \mathrm{Fib}(n)\!\downarrow \;\Longrightarrow\; \mathrm{Fib}(n) = \mathrm{fib}(n). \tag{5.19}$$

Of course the condition $\mathrm{Fib}(n){\downarrow}$ is satisfied for every $n \geq 0$, but we have not yet presented a method for proving it. We will apply (R26), and the assertions σ_i are the corresponding assertions about the recursive calls within S:

$$\begin{aligned}
\sigma_1\text{:}\quad & N-2 \geq 0 \,\&\, \mathrm{Fib}(N-2){\downarrow} \Longrightarrow \mathrm{Fib}(N-2) = \mathrm{fib}(N-2),\\
\sigma_2\text{:}\quad & N-1 \geq 0 \,\&\, \mathrm{Fib}(N-1){\downarrow} \Longrightarrow \mathrm{Fib}(N-1) = \mathrm{fib}(N-1).
\end{aligned}$$

Assertion σ from our application of (R26) is the specification about S which allows to deduce (5.19), as well as σ_1 and σ_2 according to (R25), namely:

$$\sigma\text{:}\quad \{N = n \geq 0\} \quad S \quad \{X = \mathrm{fib}(n)\}.$$

In order to apply (R26), we have to show:

$$\sigma_1 \,\&\, \sigma_2 \to \sigma, \tag{5.20}$$

$$\sigma \to \sigma_1 \quad \text{and} \quad \sigma \to \sigma_2. \tag{5.21}$$

(5.21) follows immediately by (R25), or, more precisely, by the corresponding rule providing conditional specifications, as illustrated for (R7) above.

For (5.20) we need:

$$\{N = n \geq 0 \,\&\, N \leq 1\} \quad X := N \quad \{X = \mathrm{fib}(n)\}, \tag{5.22}$$

$$\sigma_1 \,\&\, \sigma_2 \to [\{N = n > 1\} \quad X := \mathrm{Fib}(N-2) + \mathrm{Fib}(N-1) \quad \{X = \mathrm{fib}(n)\}]. \tag{5.23}$$

For (5.22), backward substitution yields of course $N = \mathrm{fib}(n)$, so we have to show $N = n \geq 0 \,\&\, N \leq 1 \Longrightarrow N = \mathrm{fib}(n)$, which is clear, since n does indeed equal $\mathrm{fib}(n)$ for $n = 0, 1$. For (5.23), we compute the weakest precondition according to theorem 2.1 (iii), which yields $\mathrm{Fib}(N-2) + \mathrm{Fib}(N-1){\uparrow} \vee \mathrm{Fib}(N-2) + \mathrm{Fib}(N-1) = \mathrm{fib}(n)$; in this case the subformula $\mathrm{Fib}(N-2) + \mathrm{Fib}(N-1){\uparrow}$ has to be included. (In examples considered so far we simply applied backward substitution, which means that the subformula $A{\uparrow}$ does not appear; in most cases $A{\uparrow}$ was trivial.) So we have to show:

$$N = n > 1 \Longrightarrow \mathrm{Fib}(N-2) + \mathrm{Fib}(N-1){\uparrow} \vee \mathrm{Fib}(N-2) + \mathrm{Fib}(N-1) = \mathrm{fib}(n). \tag{5.24}$$

This obviously follows from σ_1 and σ_2, since $\mathrm{fib}(n) = \mathrm{fib}(n-2) + \mathrm{fib}(n-1)$ for $n > 1$. Note that σ_1 and σ_2 refer precisely to the calls of Fib which appear in (5.24). This proves (5.23) and hence (5.20). Then we can infer σ by (R26) and (5.19) by (R25).

Example 5.10: Now consider the well-known QUICKSORT-algorithm for sorting an array $A[1..n]$, where n is a constant. We use A as a global variable. Basically the algorithm works as follows:

(a) Select an element $\hat{A}$ from the array and rearrange the elements such that for some K, $\hat{A}$ occurs at position K, the remaining elements $\leq \hat{A}$ occur to the left of position K, and the remaining elements $\geq \hat{A}$ to the right.

(b) Recursively apply the algorithm to sort $A[1..K-1]$ and $A[K+1..n]$.

Since the algorithm is applied recursively to segments of A, the procedure QUICKSORT has to use two value parameters; QUICKSORT(L, R) sorts the segment $A[L..R]$. Step (a) of the algorithm is accomplished by an auxiliary procedure with the heading

Part($L, R;$ **var** K); it returns a value for K between L and R and rearranges the segment $A[L..R]$ such that $A[L], \ldots, A[K-1]$ are $\leq A[K]$, and $A[K+1], \ldots, A[R]$ are $\geq A[K]$. We will not give the code for Part, which is of no relevance for our current topic (recursion). Only an appropriate specification for calls of Part will be needed.

How does the sequence of recursive calls of QUICKSORT terminate? Obviously, if $A[L..R]$ contains at most one element, then QUICKSORT(L, R) does not have to induce any further recursive call, in fact, no action at all is needed in this case. This leads to the following declaration:

```
procedure QUICKSORT(L, R);
local K;
begin
    if L < R then
        begin
            Part(L, R, K);
            QUICKSORT(L, K − 1);
            QUICKSORT(K + 1, R)
        end
end
```

What we want to show is the following: If A equals a initially, then after the call QUICKSORT$(1, n)$, A is a permutation of a and sorted. In order to write this as a formal specification in a convenient way, we have to introduce some notation, namely abbreviations (identifiers) for a number of predicates. At the moment only the abbreviations Perm and Sorted are defined. Perm means that A is a permutation of a, where a represents the initial value of A. Sorted means that A is the result of sorting a, a condition which is given by the formula

$$\text{Perm} \,\&\, (\forall i, j)(1 \leq i \leq j \leq n \Longrightarrow A[i] \leq A[j]).$$

So the desired specification to show is

$$\{A = a\} \quad \text{QUICKSORT}(1, n) \quad \{\text{Sorted}\}. \tag{5.25}$$

Now the tricky step is to find appropriate specifications for calls of QUICKSORT and Part, with arbitrary values of the parameters. Indeed, it does not suffice to provide a specification for QUICKSORT(L, R) which asserts that the segment $A[L..R]$ gets sorted. Rather, we also have to express the fact that other segments of A remain unaffected. The normal way to proceed would be to use a mathematical variable a' which represents the array A before the procedure call. However, this turns out to be unnecessary in the current situation. What we need is a condition which reflects the general situation at some point during the execution of the algorithm—the analogue of a loop invariant. Then the effect of each procedure call can be expressed by stating, how this condition is modified.

For instance, assume that $n = 100$, and that the following steps have been performed so far:

(a) The call Part$(1, 100, K)$, induced by QUICKSORT$(1, 100)$, has returned $K = 60$, so the recursive calls QUICKSORT$(1, 59)$ and QUICKSORT$(61, 100)$ were still needed.

(b) The call Part$(1, 59, K)$, induced by QUICKSORT$(1, 59)$, has returned $K = 22$, necessitating the recursive calls QUICKSORT$(1, 21)$ and QUICKSORT$(23, 59)$.
(c) QUICKSORT$(1, 21)$ has been finished.

Then the resulting situation is as follows: the segment $A[1..22]$ is sorted, the elements $A[i]$ satisfy $A[22] \leq A[i] \leq A[60]$ for $23 \leq i \leq 59$, and $A[60] \leq A[i]$ for $i \geq 61$. (In addtion, the condition Perm holds, of course, but this will be assumed tacitly from now on.)

The situation just described is an instance of a general condition PS$(x_1, \ldots, x_m)$ with $x_1 < x_2 < \cdots < x_m$ which holds before and after each procedure call. It means: the segment $A[1..x_1]$ is sorted, and we have $A[x_j] \leq A[i] \leq A[x_{j+1}]$ for $x_j < i < x_{j+1}$ and $A[x_m] \leq A[i]$ for $x_m < i$. In this situation the elements $A[1]$, ..., $A[x_1]$, as well as $A[x_2]$, ..., $A[x_m]$ are already at the right place, and it remains to sort each of the segments $A[x_1 + 1..x_2 - 1]$, $A[x_2 + 1..x_3 - 1]$, ..., $A[x_m + 1..m]$. The condition Sorted is equivalent to PS(n). What if there exists no initial segment $A[1..x_1]$ in which each element is already at the right place? In this case we put $x_1 = 0$, and in the condition $A[x_1] \leq A[i] \leq A[x_2]$ for $x_1 < i < x_2$ the first inequality is taken to be vacously true. This is accomplished by the following convention: $A[0]$ means $-\infty$ in the following mathematical formula, even though $A[0]$ cannot be used in program code. Likewise, it also proves convenient to define $A[n+1]$ as ∞, which makes PS$(x_1, \ldots, x_m)$ $(X_n \leq n)$ equivalent to PS$(x_1, \ldots, x_m, n+1)$. Once again the purpose of this convention is to simplify certain formulas; in particular, it allows to write the specification for Part without a distintion by cases (or **if-then-else**-expressions). The formula representing PS$(x_1, \ldots, x_m)$ is

$$
\begin{aligned}
\mathrm{Perm} \,\&\, 0 \leq x_1 < x_2 < \cdots < x_m \leq n+1 \,\&\, (\forall i, j)(1 \leq i \leq j \leq x_1 \Longrightarrow A[i] \leq A[j])& \\
\&\, (\forall i, j)(1 \leq j < m \,\&\, x_j < i < x_{j+1} \Longrightarrow A[x_j] \leq A[i] \leq A[x_{j+1}])& \\
\&\, (\forall i)(x_m < i \Longrightarrow A[x_m] \leq A[i]).&
\end{aligned}
$$

Later there will be no need to write PS$(x_1, \ldots, x_m)$ out in this fashion, so in particular the convention about $A[0]$ and $A[n+1]$ plays no further rôle. The initial situation, where A is completely unsorted, is given by PS(0), and after the first call of Part$(1, n, K)$ we have PS$(0, k)$, where k is the value of K which was returned.

The procedure call QUICKSORT(L, R) and the induced call Part(L, R, K) are only applied for $L = x_1 + 1$ and $R = x_2 - 1$ in a situation where PS$(x_1, \ldots, x_m)$ holds—possibly with $x_2 = n + 1$. This is the reason why we allowed x_m to be $n + 1$, and why $A[n + 1]$ was defined to be ∞.

An informal description of the effect of Part(L, R, K) was already given above. It amounts to the following: If PS$(x_1, \ldots, x_m)$ holds prior to this call, and if the call Part(L, R, K) is applied with $L = x_1 + 1$ and $R = x_2 - 1$, where $L < R$, then PS$(x_1, K, x_2, \ldots, x_m)$ holds afterwards. In addition it is important that Part(L, R, K) does not change the variables L and R from the calling procedure. This fact is trivial, since L and R are formal value parameters, which means that assignments to them only effect local variables which are also represented by L and R but distinct from the global ones. So the right specification is

$$
\begin{aligned}
\{\mathrm{PS}(x_1, \ldots, x_m) \,\&\, L = x_1 + 1 < R = x_2 - 1\} \quad \mathrm{Part}(L, R, K)& \\
\{\mathrm{PS}(x_1, K, x_2, \ldots, x_m) \,\&\, L = x_1 + 1 \,\&\, R = x_2 - 1\},& \qquad (5.26)
\end{aligned}
$$

which, of course, can be derived only when the code for Part is given. We take (5.26) as an assumption for our verification.

What we have to show is an appropriate specification for an arbitrary call of QS, which, naturally, will be more general than (5.25). Again we have to include the assertion that the values of the variables L, R and K from the calling procedure (possibly another activation of QUICKSORT) are not altered. In particular, the recursive call QUICKSORT$(L, K-1)$ must not change these values, since otherwise the second call QUICKSORT$(K+1, R)$ would be carried out for wrong values of the parameters. Of course, these recursive calls use their own local versions of these variables, which are distinct from the global ones. We use eq as an abbreviation for $L = l \,\&\, R = r \,\&\, K = k$; so the specification to prove is

$$\begin{aligned}&\{\mathrm{PS}(x_1, \ldots, x_m) \,\&\, L = x_1 + 1 \,\&\, R = x_2 - 1 \,\&\, \mathrm{eq}\}\\ &\qquad\qquad\qquad \mathrm{QUICKSORT}(L, R) \quad \{\mathrm{PS}(x_2, \ldots, x_m) \,\&\, \mathrm{eq}\},\end{aligned} \tag{5.27}$$

As instantiations of (5.27) we obtain the specifications σ_1, σ_2 needed for the application of (R26):

$$\begin{aligned}\sigma_1\colon \quad &\{\mathrm{PS}(x_1, \ldots, x_m) \,\&\, L = x_1 + 1 \,\&\, K - 1 = x_2 - 1 \,\&\, \mathrm{eq}\}\\ &\qquad\qquad\qquad \mathrm{QUICKSORT}(L, K-1) \quad \{\mathrm{PS}(x_2, \ldots, x_m) \,\&\, \mathrm{eq}\},\\ \sigma_2\colon \quad &\{\mathrm{PS}(x_1, \ldots, x_m) \,\&\, K + 1 = x_1 + 1 \,\&\, R = x_2 - 1 \,\&\, \mathrm{eq}\}\\ &\qquad\qquad\qquad \mathrm{QUICKSORT}(K+1, R) \quad \{\mathrm{PS}(x_2, \ldots, x_m) \,\&\, \mathrm{eq}\}.\end{aligned}$$

The procedure body S also satisfies the same specification as QUICKSORT(L, R), but at this point it is crucial to distinguish between global and local versions of L, R and K. In (5.27), where L and R are actual parameters, and also in the pre- and postcondition of the following specification σ, there occur only global versions, that is, versions which are defined in the procedure from which the call QUICKSORT(L, R) is made. By contrast, the body S contains local versions from the call QUICKSORT(L, R) itself. We indicate global versions by the subscript g, and eq_g is an abbreviation for $L_g = l \,\&\, R_g = r \,\&\, K_g = k$. So the right specification for S is

$$\sigma\colon \quad \{\mathrm{PS}(x_1, \ldots, x_m) \,\&\, L_g = x_1 + 1 \,\&\, R_g = x_2 - 1 \,\&\, \mathrm{eq}_g\} \quad S \quad \{\mathrm{PS}(x_2, \ldots, x_m) \,\&\, \mathrm{eq}_g\}.$$

Analogously as in example 5.9 we have to show:

$$\sigma_1 \,\&\, \sigma_2 \rightarrow \sigma, \tag{5.28}$$

$$\sigma \rightarrow \sigma_1 \quad \text{and} \quad \sigma \rightarrow \sigma_2, \tag{5.29}$$

and (5.29) is obtained immediately by (R23). For (5.28) we have to show:

$$\begin{aligned}&\mathrm{PS}(x_1, \ldots, x_m) \,\&\, L = x_1 + 1 \,\&\, R = x_2 - 1 \,\&\, \mathrm{eq}_g \,\&\, L \geq R\\ &\qquad\qquad\qquad\qquad \Longrightarrow \mathrm{PS}(x_2, \ldots, x_m) \,\&\, \mathrm{eq}_g,\end{aligned} \tag{5.30}$$

$$\begin{aligned}&\sigma_1 \,\&\, \sigma_2 \rightarrow \{\mathrm{PS}(x_1, \ldots, x_m) \,\&\, L = x_1 + 1 \,\&\, R = x_2 - 1 \,\&\, \mathrm{eq}_g \,\&\, L < R\}\\ &\qquad\qquad\qquad\qquad S_1 \quad \{\mathrm{PS}(x_2, \ldots, x_m) \,\&\, \mathrm{eq}_g\},\end{aligned} \tag{5.31}$$

where S_1 is the **then**-part of the conditional, namely

$$\text{Part}(L, R, K);\ \text{QUICKSORT}(L, K-1);\ \text{QUICKSORT}(K+1, R).$$

(5.30) is established simply by noting that if $\text{PS}(x_1, \ldots, x_m)$ holds, and $x_1 + 1 \leq x_2 - 1$, then the segment $A[1..x_1]$ is sorted and $A[x_1+1..x_2-1]$ consists of at most one element which lies between $A[x_1]$ and $A[x_2]$; this implies that $A[1..x_2]$ is sorted.

For (5.31) we note first that S_1 cannot contain any assignment to L_g, R_g and K_g, although the call $\text{Part}(L, R, K)$ does involve an assignment to the local variable K. This means that whenever there exists a proof for a specification $\{\phi\}\ S_1\ \{\psi\}$ (using σ_1 and σ_2), then there also exists a proof for $\{\phi \,\&\, \text{eq}_g\}\ S_1\ \{\psi \,\&\, \text{eq}_g\}$; the clause eq_g remains unaffected by substitution corresponding to assignments. Therefore it suffices to prove the following conditional specification, where eq_g has been omitted from (5.31) and the precondition has been rewritten:

$$\sigma_1 \,\&\, \sigma_2 \rightarrow \{\text{PS}(x_1, \ldots, x_m) \,\&\, L = x_1 + 1 < R = x_2 - 1\} \quad S_1 \quad \{\text{PS}(x_2, \ldots, x_m)\}. \tag{5.32}$$

First (5.26) is applied, which yields the condition

$$\text{PS}(x_1, K, x_2, \ldots, x_m) \,\&\, L = x_1 + 1 \,\&\, R = x_2 - 1 \tag{5.33}$$

after the execution of $\text{Part}(L, R, K)$. Next we use σ_1 and σ_2 with appropriate substitutions for the free variables. Specifically, $(x_1, \ldots, x_m)$ is substituted by $(x_1, k, x_2, \ldots, x_m)$ in σ_1 and by $(k, x_2, \ldots, x_m)$ in σ_2, and in both cases l and r are replaced by $x_1 + 1$ and $x_2 - 1$ respectively. We get:

$$\begin{array}{l}\{\text{PS}(x_1, k, x_2, \ldots, x_m) \,\&\, L = x_1 + 1 \,\&\, K = k \,\&\, R = x_2 - 1\} \\ \text{QUICKSORT}(L, K-1) \\ \{\text{PS}(k, x_2, \ldots, x_m) \,\&\, L = x_1 + 1 \,\&\, K = k \,\&\, R = x_2 - 1\},\end{array} \tag{5.34}$$

$$\begin{array}{l}\{\text{PS}(k, x_2, \ldots, x_m) \,\&\, L = x_1 + 1 \,\&\, K = k \,\&\, R = x_2 - 1\} \\ \text{QUICKSORT}(K+1, R) \\ \{\text{PS}(x_2, \ldots, x_m) \,\&\, L = x_1 + 1 \,\&\, K = k \,\&\, R = x_2 - 1\}.\end{array} \tag{5.35}$$

By combining and omitting the last three clauses in the postcondition (which are no longer needed) we get:

$$\begin{array}{l}\{\text{PS}(x_1, k, x_2, \ldots, x_m) \,\&\, L = x_1 + 1 \,\&\, K = k \,\&\, R = x_2 - 1\} \\ \qquad \text{QUICKSORT}(L, K-1);\ \text{QUICKSORT}(K+1, R) \quad \{\text{PS}(x_2, \ldots, x_m)\}.\end{array} \tag{5.36}$$

Here k does no longer occur in the postcondition, so it may be replaced by K, whereupon the precondition coincides with (5.33). Hence we may combine (5.26) and (5.36) to obtain

$$\{\text{PS}(x_1, \ldots, x_m) \,\&\, L = x_1 + 1 < R = x_2 - 1\} \quad S_1 \quad \{\text{PS}(x_2, \ldots, x_m)\}. \tag{5.37}$$

The derivation of (5.37) used σ_1 and σ_2, so (5.32) is established, and the proof complete.

We conclude the section and the lecture notes with brief remarks on two more issues, namely *mutually recursive procedures and functions* and the proof of *total* correctness. Let $\mathrm{Proc}_1, \ldots, \mathrm{Proc}_l$ be mutually recursive procedures, S_i the body of the declaration of Proc_i. (So S_i now plays a different rôle than in (R26), and similarly for the specifications σ_i to be considered.) Each S_i may contain recursive calls of each of the procedures Proc_j; denote them by $S_{i,1}, \ldots, S_{i,k_i}$. As before we have to find appropriate specifications σ_i for S_i and $\sigma_{i,j}$ for $S_{i,j}$. The verification rule (R26) is modified as follows: the specifications σ_i are deduced simultaneously, and the hypothesis needed are $\sigma_{i,1} \mathbin{\&} \cdots \mathbin{\&} \sigma_{i,k_i} \to \sigma_i$ $(i = 1, \ldots, l)$, and $\sigma_i \to \sigma_{i,j}$ for all i and j for which $S_{i,j}$ is a call of Proc_i.

How can termination of recursive procedure calls be established? Consider example 5.7, and assume that $\mathrm{Fib}(N)$ is called, where N has value n. Then the values of the parameter in the recursive procedure calls are $n-1$ and $n-2$, which is *smaller* than n. So if there is a sequence of activations $A_0, A_1, \ldots$, each inducing the next, then the corresponding values of N decrease, and eventually there has to be an activation which induces no further one.

In general we have to find an appropriate expression E, depending on the values of the parameters and possibly on global variables, and a well-founded relation $\preceq$ with the following property: If A_0 is an activation of Proc inducing another activation A_1, and if e_0 and e_1 are the corresponding values of E, then $e_1 \prec e_0$. Thus let R_1 be a recursive call of Proc within the procedure body S, S_1 the portion of S preceding R_1, E' the expression E with parameters substituted according to the call R_1, and assume that $\{\phi\}\ S_1\ \{\psi\}$ is a specification for S_1 needed to establish partial correctness. Then for total correctness we have to show $\{\phi \mathbin{\&} E = e\}\ S_1\ \{\psi \mathbin{\&} E' < e\}$. In addition, partial correctness has to be replaced by total correctness in the remaining hypothesis in (R26) and in the conclusion.