The formal treatment of some small examples.

In this chapter I shall give the formal development of a series of small programs solving simple problems. This chapter should not be interpreted as my suggestion that these programs must or should be developed in such a way: such a suggestion would be somewhat ridiculous. I expect most of my readers to be familiar with most of the examples and, if not, they can probably write down a program, hardly aware of having to think about it.

The development, therefore, is given ~~for~~ for quite other reasons. One reason is to make ourselves more familiar with the formalism as far as it has been developed up ~~till~~ to now. A second reason is to convince ourselves that, in principle at least, the formalism is able to make explicit and quite rigourous what is often justified with a lot of hand-waving. A third reason is precisely that most of us are so~~m~~ familiar with them that we have forgotten how, a long time ago, we have convinced ourselves of their correctness: in this respect this chapter resembles the beginning lessons in plane geometry that are traditionally devoted to proving the obvious. Fourthly, we may occasionally get a little surprise and discover that a little familiar problem is not so familiar after all. Finally it may shed some light on the feasibility, the difficulties and the possibilities of automatic program composition or mechanical assistance in the programming process. This could be of importance even if we do not have the slightest interest in automatic program composition, for it may give us a better appreciation of the role that our inventive powers may or have to play.

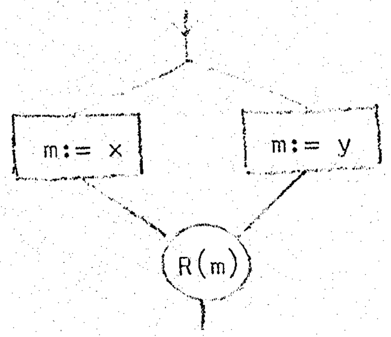
In my examples I shall state requirements of the form "for fixed x, y, ..."; this is an abreviation for "for any values x0 , y0, ... a post-condition of the form x = x0 <u>and</u> y = y0 <u>and</u> ... should give rise to a pre-condition implying x = x0 <u>and</u> y = y0 <u>and</u> ... ". We shall guarantee this by treating such quantities as "temporary constants", they will not occur to the left of an assigment statement.

<u>First example</u>.

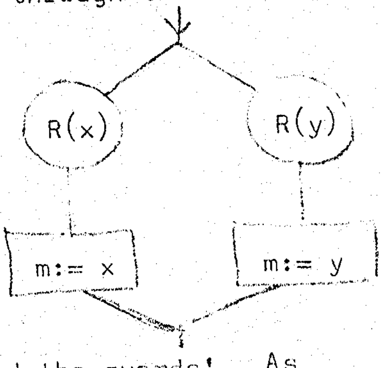
Establish for fixed x and y the relation R(m):

$$(m = x \ \underline{\text{or}}\ m = y)\ \underline{\text{and}}\ m \geq x\ \underline{\text{and}}\ m \geq y \quad .$$

For general values of x and y the relation m = x can only be established by the assignment m:= x ; as a consequence (m = x <u>or</u> m = y) can only be established by activating either m:= x or m:= y. In flow-chart form



The point is that at the entry the good choice must be made so as to guarantee that upon completion R(m) holds. For this purpose we "push the post-condition through the alternatives":



and we have derived the guards! As .

$$R(x) = ((x = x \text{ \underline{or} } x = y) \text{ \underline{and} } x \geq x \text{ \underline{and} } x \geq y) = (x \geq y)$$

and $$R(y) = ((y = x \text{ \underline{or} } y = y) \text{ \underline{and} } y \geq x \text{ \underline{and} } y \leq y) = (y \geq x)$$

we arrive at our solution:

```
if x ≥ y → m:= x [] y ≥ x → m:= y fi        .
```

Because $(x \geq y \text{ \underline{or} } y \geq x) = T$, the program will never abort (and in passing we have given an existence proof: for any values x and y there exists an m satisfying $R(m)$). Because $(x \geq y \text{ \underline{and} } y \geq x) \neq F$, our program is not necessarily deterministic: if initially $x = y$, it is undetermined which of the two assignments will be selected for execution; this non-determinacy is fully correct, because we have shown that the choice does not matter.

Note. If the function "max" had been an available primitive, we could have coded m:= max(x, y) because $R(\max(x, y)) = T$. (End of note.)

The program we have derived is not very impressive; on the other hand we observe that in the process of deriving the program from our post-condition next to nothing has been left to our invention.

Second example.

For a fixed value of n $(n > 0)$ a function $f(i)$ is given for $0 \leq i < n$. Establish the truth of R:

$$0 \leq k < n \text{ \underline{and} } (\underline{A}\ i: 0 \leq i < n: f(k) \geq f(i)) \quad .$$

✱ Because our program must work for any positive value of n it is hard to see how R can be established without a loop; we are therefore looking for a relation P that is easily established to start with and such that eventually $(P \text{ \underline{and} } \text{\underline{non} } BB) \Rightarrow R$. In search of P we are therefore looking for a relation weaker than R , in other words: we want a generalization of our final state. A standard way of generalizing a relation is the replacement of a constant by a variable --possibly with a restricted range-- and here my experience suggests that we replace the constant n by a new variable, j say, and take for P:

$$0 \le k < j \le n \ \underline{\text{and}}\ (\underline{A}\ i: 0 \le i < j : f(k) \ge f(i))$$

where the condition $j \le n$ has been added in order to do justice to the finite domain of the function f . Then, with such a generalization, we have trivially $(P\ \underline{\text{and}}\ n = j) \Rightarrow R$.

In order to verify whether this choice of P can be used, we must have an easy way of establishing it to start with. Well, because

$$(k = 0\ \underline{\text{and}}\ j = 1) \Rightarrow P$$

we venture the following structure for our program -comments being added between braces--

```
k, j:= 0, 1 {P has been established};
do j ≠ n → a step towards  j = n  under invariance of  P  od
{R has been established}   .
```

Again my experience suggests to choose as monotonicly decreasing function t of the current state $t = (n - j)$, which, indeed, is such that $P \Rightarrow (t \ge 0)$. In order to ensure this monotonic decrease of t I propose to subject j to an increase by 1 and we can develop

$$wp(\text{"}j{:=}\ j + 1\text{"}, P) =$$
$$0 \le k < j + 1 \le n\ \underline{\text{and}}\ (\underline{A}\ i: 0 \le i < j + 1: f(k) \ge f(i)) =$$
$$0 \le k < j + 1 \le n\ \underline{\text{and}}\ (\underline{A}\ i: 0 \le i < j: f(k) \ge f(i))\ \underline{\text{and}}\ f(k) \ge f(j) .$$

The first two terms are implied by $P\ \underline{\text{and}}\ j \ne n$ (for $(j \le n\ \underline{\text{and}}\ j \ne n) \Rightarrow$

$(j + 1 \leq n)$ and this is the reason why we decided to increase j only by 1.)
Therefore

$$(P \ \underline{\text{and}}\ j \neq n \ \underline{\text{and}}\ f(k) \geq f(j)) \Rightarrow wp(\text{"}j{:=}\ j + 1\text{"}, P)$$

and we can take the last condition as guard. The program

```
        k, j:= 0, 1;
        do j ≠ n → if f(k) ≥ f(j) → j:= j + 1 fi od
```

will indeed give the good answer when it terminates properly. Proper termination, however, is not guaranteed, because the alternative construct might lead to abortion ---and it will certainly do so, if $k = 0$ does not satisfy R . If $f(k) \geq f(j)$ does not hold, we can make it hold by the assignment $k{:=}\ j$ and therefore our next investigation is

$$\begin{aligned} &wp(\text{"}k, j{:=}\ j, j + 1\text{"}, P) = \\ &\quad 0 \leq j < j + 1 \leq n \ \underline{\text{and}}\ (\underline{A}\ i{:}\ 0 \leq i < j + 1{:}\ f(j) \geq f(i)) = \\ &\quad 0 \leq j < j + 1 \leq n \ \underline{\text{and}}\ (\underline{A}\ i{:}\ 0 \leq i < j{:}\ f(j) \geq f(i)) \end{aligned}$$

To our great relief we see that

$$(P \ \underline{\text{and}}\ j \neq n \ \underline{\text{and}}\ f(k) \leq f(j)) \Rightarrow wp(\text{"}k, j{:=}\ j, j + 1\text{"}, P)$$

and the following program will do the job without the danger of abortion:

```
        k, j:= 0, 1;
        do j ≠ n → if f(k) ≥ f(j) → j:= j + 1
                   ▯ f(k) ≤ f(j) → k, j:= j, j + 1 fi od      .
```

A few remarks are in order. The first one is that, as the guards of the alternative construct do not necessarily exclude each other, the program harbours the same kind of internal non-determinacy as the first example. Externally it may display this non-determinacy as well. The function f could be such that the final value of k is not unique: in that case our program can deliver any acceptable value!

The second remark is that having developed a correct program does not mean that we are through with the problem. Programming is as much a mathematical discipline as an engineering discipline, correctness is as much our concern as, say, efficiency. Under the assumption that the computation of a value of the function f for a given argument is a * relatively time-consuming operation, a good engineer should observe that in all probability this program will often ask for many re-computations of $f(k)$ for the same value of k . If this is the case the trading of some storage space against some computation time is indicated. The effort to make our program more time-efficient, however, should never be an excuse to make a mess of it. (This is obvious, but I state it explicitly because so much messiness is so often defended by an appeal to efficiency considerations, while upon closer inspection the defense is always unvalid: it must be, for a mess is never defensible.) The orderly technique for trading storage space versus computation time is the introduction of one or more redundant variables, the value of which can be used because some relation is kept invariant. In this example the observation of the possibly frequent re-computation of $f(k)$ for the same value of k suggests the introduction of a further variable, max say, and to extend the invariant relation with the further term

$$max = f(k) \quad .$$

This relation must be established upon initialization of k and be kept invariant --by explicit assignment to max -- upon modification of k . We arrive at the following program

```
k, j, max:= 0, 1, f(0);
do j ≠ n → if max ≥ f(j) → j:= j + 1
                   ▯ max ≤ f(j) → k, j, max:= j, j + 1, f(j) fi od      .
```

This program is probably much more efficient than our previous version. If it is, a good engineer does not stop here, because he will now observe that for the same value of j he might order a number of times the computation of $f(j)$. It is suggested to introduce a further variable, h say (short for "help") and to keep

$$h = f(j)$$

invariant. This, however, is a thing that we cannot do on the same global level as with our previous term: the value $j = n$ is not excluded and for that value $f(j)$ is not necessarily defined. The relation $h = f(j)$ is therefore re-established every time $j \neq n$ has just been checked; upon completion of the outer guarded command --"just before the od" so to speak-- we have $h = f(j - 1)$ but we don't bother and leave it at that.

```
k, j, max:= 0, 1, f(0);
do j ≠ n → h:= f(j);
        if max ≥ h → j:= j + 1
         [] max ≤ h → k, j, max:= j, j + 1, h fi od
```

A final remark is not so much concerned with our solution as with our considerations. We have had our mathematical concerns, we have had our engineering concerns and we have accomplished a certain amount of separation between them, now focussing our attention on this aspect and then on that aspect. While such a separation of concerns is absolutely essential when dealing with more complicated problems, I must stress that focussing one's attention on one aspect does not mean completely ignoring the others. In the more mathematical part of the design activity we should not head for a mathematically correct program that is so badly engineered that it is beyond salvation. Similarly, while "trading" we should not introduce errors through sloppiness, we should do it careful and systematic; also, although the mathematical analysis as such has been completed, we should still understand enough about the problem to judge whether our considered changes are significant improvements.

Note. Prior to my getting used to these formal developments I would always have used "$j < n$" as the guard for this repetitive construct, a habit I still have to unlearn, for in a case like this, the guard "$j \neq n$" is certainly to be preferred. The reason for the preference is twofold. The guard "$j \neq n$" allows us to conclude $j = n$ upon termination without an appeal to the invariant relation P and thus simplifies the argument about what the whole construct achieves for us compared with the guard "$j < n$". Much more important, however, is that the guard "$j \neq n$" makes termination dependent upon (part of) the invariant relation, viz. $j \leq n$ and is therefore to be preferred for reasons of robustness. If the addition $j := j + 1$ would erroneously increase j too much and would establish $j > n$, then the guard "$j < n$" would give no alarm, while the guard "$j \neq n$" would at least prevent proper termination. Even without taking machine malfunctioning into account, this argument seems valid. Let a sequence x_0, x_1, x_2, ... be given by a value for x_0 and for $i > 0$ by $x_i = f(x_{i-1})$, where f is some computable function and let us carefully and correctly keep the relation $X = x_i$ invariant. Suppose that we have in a program a monotonicly increasing variable n, such that for some values of n we are interested in x_n. Provided $n \geq i$, we can always establish $X = x_n$ by

$$\underline{\text{do}}\ i \neq n \rightarrow i, X := i + 1, f(X)\ \underline{\text{od}} \quad .$$

If --due perhaps to a later change in the program with the result that it is no longer guaranteed that n can only increase as the computation proceeds--

the relation $n \geq i$ does not necessarily hold, the above construct would (luckily!) fail to terminate, while the use of

$$\underline{\text{do}}\ j < n \rightarrow i, X:= i + 1, f(X)\ \underline{\text{od}}$$

would have failed to establish the relation $X = x_n$. The moral of the story is that, all other things being equal, we should choose our guards as weak as possible. (End of note.)

Third example.

For fixed a $(a \geq 0)$ and d $(d > 0)$ it is requested to establish

R: $$0 \leq r < d\ \underline{\text{and}}\ (a - r)|d \quad .$$

(Here the vertical bar "|" is to be read as: "is a multiple of".) In other words we are requested to compute the smallest non-negative remainder r that is left after division of a by d . In order that the problem be a problem, we have to restrict ourselves to addition and subtraction as the only aritmetic operations. Because the term $(a - r)|d$ is satisfied by $r = a$, an initialization that --on account of $a \geq 0$-- also satisfies $0 \leq r$, it is suggested to choose as invariant relation P:

$$0 \leq r\ \underline{\text{and}}\ (a - r)|d \quad .$$

* For the function t , the decrease of which should ensure termination, we choose r itself; because the massaging of r must be such that the relation $(a - r)|d$ is kept invariant, r may only be changed by a multiple of d , for instance d itself. Thus we find ourselves invited to evaluate

$$wp(\text{"}r:= r - d\text{"}, P)\ \underline{\text{and}}\ wdec(\text{"}r:= r - d\text{"}, r) =$$
$$0 \leq r - d\ \underline{\text{and}}\ (a - r + d)|d\ \text{and}\ d > 0 \quad .$$

Because the term $d > 0$ could have been added to the invariant

* relation $\mathcal{P}$, only the first term is then not implied; we find the corresponding guard "$r \geq d$" and the tentative program:

```
if a ≥ 0 and d > 0 →
     r:= a;
     do r ≥ d → r:= r - d od
fi     .
```

Upon completion the truth of P and non $r \geq d$ has been established, a relation that implies R and thus the problem has been solved.

Suppose now, that in addition it would have been required to assign to q such a value that finally we also have

$$a = d * q + r$$

--in other words it is requested to compute the quotient as well-- then we can try to add this term to our invariant relation. Because

$$(a = d * q + r) \Rightarrow (a = d *(q + 1)+(r - d))$$

we are led to the program:

```
if a ≥ 0 and d > 0 →
     q, r:= 0, a;
     do r ≥ d → q, r:= q + 1, r - d od
fi          .
```

The above programs are, of course, very time-consuming if the quotient is large. Can we speed it up? The obvious way to do that is to decrease r by larger multiples of d . Introducing for this purpose a new variable, dd say, the relation to be established and kept invariant is

$$dd|d \text{ and } dd \geq d \qquad .$$

We can speed up our first program by replacing "r:= r - d" by a possibly repeated decrease of r by dd , while dd , initially = d , is allowed to grow rather rapidly, e.g. by doubling it each time. So we are led to consider the following program

```
if a ≥ 0 and d > 0 →
    r:= a;
    do r ≥ d →
        dd:= d;
        do r ≥ dd → r:= r - dd; dd:= dd + dd od
    od
fi    .
```

The relation $0 \leq r$ and $(a - r)|d$ is clearly kept invariant and therefore this program establishes R if it terminates properly, but does it? Of course it does, because the inner loop --that terminates on account of $dd > 0$ -- is only activated with initial states satisfying $r \geq dd$ and therefore the decrease r:= r - dd is performed at least once for every repetition of the outer loop.

But the above reasoning --although convincing enough!-- is a very informal one and because this chapter is called "a formal treatment" we can try to formulate and prove the theorem to which we have appealed intuitively.

With the usual meanings of IF, DO and BB , let P be the relation that is kept invariant, i.e.

$$(P \text{ and } BB) \Rightarrow wp(IF, P) \quad \text{for all states} \qquad (1)$$

and let furthermore t be an integer function such that for any value of $t0$ and for all states

$$(P \underline{\text{and}}\ BB \underline{\text{and}}\ t \leq t0 + 1) \Rightarrow wp(IF,\ t \leq t0) \tag{2}$$

or --in an equivalent formulation--

$$(P \underline{\text{and}}\ BB) \Rightarrow wdec(IF,\ t) \quad \text{for all states} \tag{3}$$

then for any value of $t0$ and for all states

$$(P \underline{\text{and}}\ BB \underline{\text{and}}\ wp(DO,\ T) \underline{\text{and}}\ t \leq t0 + 1) \Rightarrow wp(DO,\ t \leq t0) \tag{4}$$

or --in an equivalent formulation--

$$(P \underline{\text{and}}\ BB \underline{\text{and}}\ wp(DO,\ T) \Rightarrow wdec(DO,\ t) \quad . \tag{5}$$

In words: if the relation P that is kept invariant guarantees that each selected guarded command causes an effective decrease of t, then the repetitive construct will cause an effective decrease of t if it terminates properly after at least one execution of a guarded command. The theorem is so obvious, that it would be a shame if it were difficult to prove, but luckily it is not. We shall show that from (1) and (2) follows that for any value $t0$ and all states

$$(P \underline{\text{and}}\ BB \underline{\text{and}}\ H_k(T) \underline{\text{and}}\ t \leq t0 + 1) \Rightarrow H_k(t \leq t0) \tag{6}$$

for all $k \geq 0$. It holds for $k = 0$ --because $(BB \underline{\text{and}}\ H_0(T)) = F$ -- and we have to derive from the assumption that (6) holds for $k = K$, that it holds for $k = K + 1$ as well.

$$\begin{aligned}
&(P \underline{\text{and}}\ BB \underline{\text{and}}\ H_{K+1}(T) \underline{\text{and}}\ t \leq t0 + 1) \\
&\quad \Rightarrow wp(IF,\ P) \underline{\text{and}}\ wp(IF,\ H_K(T)) \underline{\text{and}}\ wp(IF,\ t \leq t0) \\
&\quad = wp(IF,\ P \underline{\text{and}}\ H_K(T) \underline{\text{and}}\ t \leq t0) \\
&\quad \Rightarrow wp(IF, (P \underline{\text{and}}\ BB \underline{\text{and}}\ H_K(T) \underline{\text{and}}\ t \leq t0 + 1) \underline{\text{or}}\ (t \leq t0 \underline{\text{and}}\ \underline{\text{non}}\ BB)) \\
&\quad \Rightarrow wp(IF,\ H_K(t \leq t0) \underline{\text{or}}\ H_0(t \leq t0)) \quad .
\end{aligned}$$

$= wp(IF, H_K(t \leq t0))$

$\Rightarrow wp(IF, H_K(t \leq t0))$ <u>or</u> $H_0(t \leq t0)$

$= H_{K+1}(t \leq t0)$.

The first implication follows from (1), the definition of $H_{K+1}(T)$ and (2), the equality in the 3rf line is obvious, the implication in the 4th line is derived by taking the conjunction with (BB <u>or</u> <u>non</u> BB) and then weakening both terms, the implication in the 5th line follows from (6) for $k = K$ and the definition of $H_0(t \leq t0)$ and the rest is straightforward. Thus relation (6) has been proved for all $k \geq 0$ and from that result (4) and (5) follow immediately.

<u>Exercise</u>. Modify also our second program in such a way that it computes the quotient as well and give a formal correctness proof for your program.

Let us assume next, that there is a small number, 3 say, by which we are allowed to multiply and to divide and that these operations are sufficiently fast so that they are attractive to use. We shall denote the product by "m * 3" --or by "3 * m"-- and the quotient by "m / 3"; the latter expression will only be called for evalution provided initially $m \dot{|} 3$ holds. (We are working with integer numbers, aren't we?)

Again we try to establish the desired relation R by means of a repetitive construct, for which the invariant relation P is derived by replacing a constant by a variable. Replacing the constant d by the variable dd , whose values will be restricted to d *(a power of 3) we come to the invariant relation P:

$0 \leq r < dd$ <u>and</u> $(a - r)|dd$ <u>and</u> $(\underline{E}\ i: i \geq 0: dd = d * 3^i)$

We shall establish the relation and then try to reach, while keeping it invariant, a state satisfying $d = dd$.

In order to establish it, we need a further repetitive construct: first we establish

$$0 \leq r \ \underline{\text{and}} \ (a - r) | dd \ \underline{\text{and}} \ (\underline{E} \ i: i \geq 0: dd = d * 3^i)$$

and then let dd grow until it is large enough and $r < dd$ is satisfied as well. The following program would do:

```
if a ≥ 0 and d > 0 →
   r, dd:= a, d;
   do r ≥ dd → dd:= dd * 3 od;
   do dd ≠ d → dd:= dd / 3;
               do r ≥ dd → r:= r - dd od
   od
fi
```

Exercise. Modify also the above program in such a way that it computes the quotient as well and give a formal correctness proof for your program. This proof has to demonstrate that whenever $dd / 3$ is computed, originally $dd|3$ holds.

The above program exhibits a quite common characteristic. On the outer level we have two repetitive constructs in succession; when we have two or more repetitive constructs on the same level in succession, the guarded commands of the later ones tend to be more elaborate than those of the earlier ones. (This is known as "Dijkstra's Law", which does not always hold.) The reason for this tendency is clear: each repetitive construct adds his "<u>and</u> <u>non</u> BB" to the relation it keeps invariant and that additional relation has to be kept invariant by the next one as well. But for the inner loop, the second one is exactly the inverse of the first one; but it is precisely the function of the added statement

```
do r ≥ dd → r:= r - dd od
```

to restore the potentially destroyed relation $r < dd$, i.e. the achievement of the first loop.

<u>Fourth example</u>.

For fixed Q1 , Q2 , Q3 and Q4 it is requested to establish R where R is given as R1 <u>and</u> R2 with
R1: The sequence of values (q1, q2, q3, q4) is a permutation of the sequence of values (Q1, Q2, Q3, Q4)
R2: $q1 \le q2 \le q3 \le q4$.

Taking R1 as relation P to be kept invariant a possible solution is

```
q1, q2, q3, q4:= Q1, Q2, Q3, Q4;
do q1 > q2 → q1, q2:= q2, q1
 [] q2 > q3 → q2, q3:= q3, q2
 [] q3 > q4 → q3, q4:= q4, q3
od          .
```

The first assignment obviously establishes P and no guarded command destroys it. Upon termination we have <u>non</u> BB , and that is relation R2. The way in which people convince themselves that it does terminate depends largely on their background: a mathematician might observe that the number of inversions decreases, an operations researcher will interpret it as maximizing $q1 + 2*q2 + 3*q3 + 4*q4$ and I, as a physicist, just "see" the center of gravity moving in the one direction (to the right, to be quite precise). The program is remarkable in the sense that, whatever we would have chosen for the guards, never would there be the danger of destroying relation P : the guards are in this example a pure consequence of the requirement of termination.

Note. Observe that we could have added other alternatives such as

$$q1 > q3 \rightarrow q1, q3 := q3, q1$$

as well; they cannot be used to replace one of the given three. (End of note.)

It is a nice example of the kind of clarity that our non-determinacy has made possible to achieve; needless to say, however, that I do not recommend to sort a large number of values in an analogous manner.

Fifth example.

We are requested to design a program approximating a square root, more precisely: for fixed n $(n \geq 0)$ the program should establish

R: $$a^2 \leq n \ \underline{\text{and}}\ (a + 1)^2 > n \quad .$$

One way of weakening this relation is dropping one of the terms of the conjunction, e.g. the last one and focus upon

P: $$a^2 \leq n$$

a relation that is obviously satisfied by $a = 0$, so that the initialization need not bother us. We observe that if the second term is not satisfied this is due to the fact that a is too small and we could therefore consider the statement "a:= a + 1". Formally we find

$$wp(\text{"a:= a + 1"}, P) = ((a + 1)^2 \leq n) \quad .$$

Taking this condition as --the only!-- guard, we have $(P\ \underline{\text{and}}\ \underline{\text{non}}\ BB) = R$

and therefore we are invited to consider the program

```
if $n \geq 0$ →
    $a := 0$ {P has been established};
    do $(a + 1)^2 \leq n$ → $a := a + 1$ {P has not been destroyed} od
    {R has been established}
fi {R has been established}
```

all under the assumption that the program terminates, what it does thanks to the fact that the square of a non-negative number is a monotonicly increasing function: we can take for t the function $n - a^2$.

This program is not very surprising, it is not very efficient either: for large values of n it could be rather time-consuming. Another way of generalizing R is the introduction of another variable, b say --and again restricting its range-- that is to replace part of R, for instance

P: $$a^2 \leq n \text{ and } b^2 > n \text{ and } 0 \leq a < b \quad .$$

By the way this has been chosen it has the pleasant property that

$$(P \text{ and } (a + 1 = b)) \Rightarrow R \quad .$$

Thus we are led to consider a program of the form --from now onwards omitting the if $n \geq 0$ → ... fi--

```
$a, b := 0, n + 1$ {P has been established};
do $a + 1 \neq b$ → decrease $b - a$ under invariance of P od
{R has been established}    .
```

Let each time the guarded command is executed d be the amount by which the difference $b - a$ is decreased. Decreasing this difference can be done by either decreasing b or by increasing a or both. Without loss of generality we can restrict ourselves to such steps in which either a or b is changed, but not both: if a is too small and b is too large and in one step only b is decreased, then a can be increased in a next step. This consideration leads to a program of the following form

```
a, b:= 0, n + 1 {P has been established};
do a + 1 ≠ b →
        d:= ... {d has a suitable value and  P  is still valid};
        if ... → a:= a + d {P has not been destroyed}
         [] ... → b:= b - d {P has not been destroyed}
       fi {P has not been destroyed}
od {R has been established}
```

Now
$$wp(\text{"}a:= a + d\text{"}, P) = ((a + d)^2 \leq n \text{ \underline{and} } b^2 > n)$$

which, because p implies the second term leads to the first one as our first guard; the second guard is derived similarly and our next form is

```
a, b:= 0, n + 1;
do a + 1 ≠ b → d:= ...;
                if (a + d)^2 ≤ n → a:= a + d
                 [] (b - d)^2 > n → b:= b - d
                fi {P has not been destroyed}
od {R has been established}      .
```

We are still left with a suitable choice for d . Because we have chosen $b - a$ --well: $b - a - 1$ actually-- as our function t , effective decrease implies that d must satisfy $d > 0$. Furthermore the following alternative construct may not lead to abortion, i,e, at least one of the guards must be true. That is, the negation of the first: $(a + d)^2 > n$ must imply the other: $(b - d)^2 > n$; this is guaranteed if

$$a + d \leq b - d$$

or $$2 * d \leq b - a \quad .$$

Besides a lower bound we have also found an upper bound for d . We could choose $d = 1$, but the larger d , the faster the program and therefore we propose:

```
    a, b:= 0, n + 1;
    do a + 1 ≠ b → d:=(b - a)div 2;
                   if (a + d)^2 ≤ n → a:= a + d
                    ▯ (b - d)^2 > n → b:= b - d
                   fi
    od
```

where n div 2 is given by $n/2$ if $n|2$ and by $(n - 1)/2$ if $(n - 1)|2$.

The use of the operator div suggests that we should look what happens if we impose upon ourselves the restricition that whenever d is computed, $b - a$ should be even. Introducing $c = b - a$ and eliminating the b , we get the invariant relation

P: $a^2 \leq n$ and $(a + c)^2 > n$ and $(E\ i: i \geq 0: c = 2^i)$

and the program (in which the roles of c and d have coincided)

```
    a, c:= 0, 1; do c^2 ≤ n → c:= 2 * c od;
    do c ≠ 1 → c:= c / 2;
               if (a + c)^2 ≤ n → a:= a + c
                ▯ (a + c)^2 > n → skip
               fi
    od    .
```

Note. This program is very much like the last program for the third example, the computation of the remainder under the assumption that we could multiply and divide by 3 . The alternative construct in our above program could have been replaced by

$$\underline{\text{do}}\ (a + c)^2 \leq n \rightarrow a:= a + c\ \underline{\text{od}} \quad .$$

If the condition for the remainder $0 \leq r < d$ would have been rewritten as $r < d$ and $(r + d) \geq d$ the similarity would be even more striking. (End of note.)

Under admission of the danger of beating this little example to death, I would like to submit the last version to yet another transformation. We have written the program under the assumption that sqaring a number is among the repertoire of available operations, but suppose it is not and suppose that multiplying and dividing by (small) powers of 2 are the only (semi-)multiplicative operations at our disposal. Then our last program as it stands, is no good, i.e. it is no good if we assume that the values of the variables as directly manipulated by the machine are to be equated to the values of the variables a and c if this computation were performed "in abstracto". To put it in another way: we can consider a and c as abstract variables whose values are represented --according to a convention more complicated than just identity-- by the values of other variables that are in fact manipulated by the machine. Instead of directly manipulating a and c , we can let the machine manipulate p , q and r , such that

$$p = a * c$$

$$q = c^2$$

$$r = n - a^2 \quad .$$

It is a co-ordinate transformation and to each path through our (a,c)-space corresponds a path through our (p,q,r)-space. Not always the other way round, for the values of p , q and r are not independent: in terms of p , q and r we have redundancy and therefore the potential to trade some storage space against not only computation time but even against the need to square! (The transformation from a point in (a,c)-space to a point in (p,q,r)-space has quite clearly been constructed with that objective in mind.) We can now try to translate all boolean expressions and moves in (a,c)-space into the corresponding boolean expressions and moves in (p,q,r)-space. If this can be done in terms of the there permissible operations, we have been successful. The transformation suggested is indeed adequate and the following program is the result (the variable h has been introduced for a very local optimization):

```
p, q, r:= 0, 1, n; do q ≤ n → q:= q * 4 od;
do q ≠ 1 → q:= q / 4; h:= p + q; p:= p / 2 {h = 2 * p + q};
              if r ≥ h → p, r:= p + q, r - h
               [] r < h → skip
              fi
od; {p has the value desired for a}
```

This fifth example has been included because it relates --in an embellished form-- a true design history. When the youngest of our two dogs was only a few months old I walked with both of them one evening, preparing my lectures for the next morning, when I would have to address students with only a few weeks exposure to programming, and I wanted a simple problem such that I could "massage" the solutions. During that one-hour walk the first, third and fourth program were developed in that order, but for the fact that the correct introduction of h in the last program was something I could only manage with the aid of pencil and paper after I had returned home. The second program, the one manipulating a and b , which here has been presented as a stepping stone to our third solution, was only discovered a few weeks later --be it in a less elegant form than presented here. A second reason for its inclusion is the relation between the third and the fourth program: with respect to the latter one the other one represents our first example of so-called "representational abstraction".

Sixth example.

For fixed X $(X > 1)$ and Y $(Y \geq 0)$ the program should establish

R: $z = X^Y$

under the --obvious-- assumption that exponentiation is not among the available repertoire. This problem can be solved with the aid of an "abstract variable" , h say; we shall do it with a loop, for which the invariant relation is

P: $h * z = X^Y$

and our (equally "abstract") program could be

```
h, z:= $X^Y$, 1 {P has been established};
do h $\neq$ 1 $\rightarrow$ squeeze h under invariance of P od
{R has been established}
```

The last conclusion is justified because (P and h = 1) $\Rightarrow$ R . The above program will terminate under the assumption that a finite number of applications of the operation "squeeze" will have established h = 1. The problem, of course, is that we are not allowed to represent the value of h by that of a concrete variable directly manipulated by the machine:

if we were allowed to do that, we could have assigned the value of X^Y immediately to z, not bothering about introducing h at all. The trick is that we can introduce two --at this level concrete-- variables, x and y say, to represent the current value of h and our first assignment suggests as convention for this representation

$$h = x^y \quad .$$

The condition "$h \neq 1$" then translates into "$y \neq 0$" and our next task is to discover an implementable operation "squeeze". Because the product $h * z$ must remain invariant under squeezing, we should divide h by the same value by which z is multiplied. In view of the way in which h is represented, the current value of x is the most natural candidate. Without any further problems we arrive at the translation of our abstract program

```
x, y, z:= X, Y, 1{P has been established};
do y ≠ 0 → y, z:= y - 1, z * x {P has not been destroyed} od
{R has been established}     .
```

Looking at this program we realize that the number of times control goes through the loop equals the original value Y and we can ask ourselves whether we can speed things up. Well, the guarded command has now the task to bring y down to zero: without changing the value of h, we can investigate whether we can change the representation of that value, in the hope of decreasing the value of y. We are just going to try to exploit that the concrete representation of a value of h as given by x^y is by no means unique. If y is even we can halve y and square x, and this will not change h at all. Just before the squeezing operation we insert the transformation towards the most attractive representation of h and here is the next program:

```
x, y, z:= X, Y, 1;
do y ≠ 0 → do y|2 → x, y:= x * x, y / 2 od;
               y, z:= y - 1, z * x
od {R has been established}
```

There exists one value that can be halved indefinitely without becoming odd and that is the value 0, in other words: the outer guard ensures that the inner repetition terminates.

I have included this example for various reasons. The discovery that a mere insertion of what on the abstract level acts like an empty statement could change an algorithm invoking a number of operations proportional to Y into one invoking a number of operations only proportional to log(Y) startled me when I made it. This discovery was a direct consequence of my forcing myself to think in terms of a single abstract variable. The exponentiation program I knew was the following:

```
x, y, z:= X, Y, 1;
do y ≠ 0 → if non y|2 → y, z:= y - 1, z * x [] y|2 → skip fi;
               x, y:= x * x, y / 2
od .
```

This latter program is very well known, it is a program that many of us have discovered independently of each other. Because the last squaring of x when y has reached the value 0 is clearly superfluous, this program has often been cited as supporting the need for what were called "intermediate exits". In view of our second program I come to the conclusion that this support is weak.

Seventh example.

For a fixed value of n $(n \geq 0)$ a function $f(i)$ is given for $0 \leq i < N$. Assign to the boolean variable "allsix" the value such that eve$\overset{n}{\text{t}}$ually

R: $$allsix = (\underline{A}\ i: 0 \leq i < n: f(i) = 6)$$

holds. (This example shows some similarity to the Second Example of this chapter. Note, however, that in this example, $n = 0$ is allowed as well. In that case the range for i for the all-quantifier "$\underline{A}$" is empty and allsix = true should hold.) Analogous to what we did in the Second Example the invariant relation

P: $$(allsix = (\underline{A}\ i: 0 \leq i < j: f(i) = 6))\ \underline{and}\ 0 \leq j \leq n$$

suggests itself, because it is easily established for $j = 0$, while $(P\ \underline{and}\ j = n) \Rightarrow R$. The only thing to do is to investigate how to increase j under invariance of P . We therefore derive

$$wp("j:= j + 1", P) =$$
$$(allsix = (\underline{A}\ i: 0 \leq i < j + 1: f(i) = 6))\ \underline{and}\ 0 \leq j + 1 \leq n \quad .$$

The last term is implied by $P\ \underline{and}\ j \neq m$; it presents no problem because we had already decided that $j \neq n$ as a guard is weak enough to conclude R upon termination. The weakest pre-condition that the assignment

$$allsix:= allsix\ \underline{and}\ f(j) = 6$$

will establish the other term is

$$(allsix\ \underline{and}\ f(j) = 6) = (\underline{A}\ i: 0 \leq i < j + 1: f(i) = 6) \quad ,$$

a condition that is implied by P. We thus arrive at the program

```
allsix, j:= true, 0;
do j ≠ n → allsix:= allsix and f(j) = 6;
                j:= j + 1
od
```

(In the guarded command we have not used the concurrent assignment for no particular reason.)

By the time that we read this program --or perhaps already earlier-- we should get the uneasy feeling that as soon as a function value $\neq 6$ has been found, there is not much point in going on. And indeed, although $(P \text{ and } j = n) \Rightarrow R$, we could have used the weaker

$$(P \text{ and } (j = n \text{ or non allsix})) \Rightarrow R$$

leading to the stronger guard "$j \neq n$ and allsix" and to the program

```
allsix, j:= true, 0;
do j ≠ n and allsix → allsix, j:= f(j) = 6, j + 1 od     .
```

(Note the simplification of the assignment to allsix , a simplification that is justified by the stronger guard.)

Exercise. Give for the same problem the correctness proof for

```
if n = 0 → allsix:= true
 [] n > 0 → j:= 0;
                do j ≠ n - 1  and f(j) = 6 → j:= j + 1 od;
                allsix:= f(j) = 6
fi
```

and also for the still more tricky program (that does away with the need to invoke the function f from more than one place in the program)

```
j:= 0;
do j ≠ n cand f(j) = 6 → j:= j + 1 od;
allsix:= j = n
```

(Here the condition conjunction operator "cand" has been used in order to do justice to the fact that $f(n)$ need not be defined.) The last program is one, that some people like very much.

Eighth example.

Before I can state our next problem, I must first give some definitions and a theorem. Let $p = (p_0, p_1, \ldots, p_{n-1})$ be a permutation of n $(n > 1)$ different values p_i $(0 \leq i < n)$, i.e. $(i \neq j) \Rightarrow (p_i \neq p_j)$. Let $q = (q_0, q_1, \ldots, q_{n-1})$ be a different permutation of the same set of n values. By definition "permutation p precedes q in the alphabetic order" if and only if for the minimum value of k such that $p_k \neq q_k$ we have $p_k < q_k$.

The so-called "alphabetic index_n" of a permutation of n different values is the ordinal number given to it when we number the $n!$ possible permutation arranged in alphabetic order from 0 through $n!-1$. For instance, for $n = 3$ and the set of values 2, 4 and 7 we have

$$\text{index}_3(2, 4, 7) = 0$$
$$\text{index}_3(2, 7, 4) = 1$$
$$\text{index}_3(4, 2, 7) = 2$$
$$\text{index}_3(4, 7, 2) = 3$$
$$\text{index}_3(7, 2, 4) = 4$$
$$\text{index}_3(7, 4, 2) = 5 \quad .$$

Let $(p_0§\ p_1§\ \ldots\ §p_n)$ denote the permutation of the n different values in monotonicly increasing order, i.e. $\mathrm{index}_n((p_0§\ p_1§\ \ldots\ §p_n)) = 0$. (For example $(4§\ 7§\ 2) = (2, 4, 7)$ but also $(7§\ 2§\ 4) = (2, 4, 7)$.)

With the above notation we can formulate the following theorem for $n > 1$:

$$\mathrm{index}_n(p_0, p_1, \ldots , p_{n-1}) =$$
$$\mathrm{index}_n(p_0,(p_1§\ p_2§\ \ldots\ §p_{n-1})) + \mathrm{index}_{n-1}(p_1, p_2, \ldots , p_{n-1})$$

(e.g. $\mathrm{index}_3(4, 7, 2) = \mathrm{index}_3(4, 2, 7) + \mathrm{index}_2(7, 2) = 2 + 1 = 3$.)
In words: the index_n of a permutation of n different values is the * index_n of the alphabeticaly first one with the same leftmost value increased by the index_{n-1} of the permutation of the remaining rightmost $n-1$ values.
As a corrolary: from

$$p_{n-k} < p_{n-k+1} < \ldots < p_{n-1}$$

follows that $\mathrm{index}_n(p_0, p_1, \ldots , p_{n-1})$ is a multiple of $k!$ and vice versa.

After these preliminaries we can describe our problem. We have a row of n positions $(n > 1)$ numbered in the order from left to right from 0 through $n-1$; in each position lies a card with a value written on it such that no two different cards show the same value.

When at any moment c_i $(0 \leq i < n)$ denotes the value on the card in position i , we have initially

$$c_0 < c_1 < \ldots < c_{n-1}$$

(i.e. the cards lie sorted in the order of increasing value). For given value of r $(0 \leq r < n!)$ we have to rearrange the cards such that in the

R: $\text{index}_n(c_0, c_1, \ldots, c_{n-1}) = r$.

the only way in which our mechanism can interfere with the cards is via the execution of the statement

$$\text{cardswap}(i, j) \quad \text{with} \quad 0 \le i, j < n$$

that will interchange the cards in positions i and j if $i \ne j$ (and will do nothing if $i = j$).

In order to perform this transformation we must find a class of states --all satisfying a suitable condition P1-- such that both initial and final states are specific instances of that class. Introducing a new variable, s say, an obvious candidate for P1 is

$$\text{index}_n(c_0, c_1, \ldots, c_{n-1}) = s$$

as this is easily established initially (viz. by "s:= 0") and (P1 <u>and</u> $s = r$) $\Rightarrow$ R .

Again we ask, whether we can think of restricting the range of s and in view of its initial value we might try

P1: $\text{index}_n(c_0, c_1, \ldots, c_{n-1}) = s$ <u>and</u> $0 \le s \le r$

which would lead to a program of the form

```
s:= 0 {P1 has been established};
do s ≠ r → {P1  and  s < r}
           increase  s  by a suitable amount under
           invariance of P1 {P1 still holds}
od {R has been established}
```

*

Our next concern is, what to choose for "a suitab:e amount". Because our increase of s must be accompanied by a reaarngement of the cards in order to keep P1 invariant, it seems wise to investigate whether we can find conditions, under which a single cardswap corresponds to a known increase of s . Let for a value of k satisfying $1 \leq k < n$ hold

$$c_{n-k} < c_{n-k+1} < \ldots < c_{n-1} \qquad ;$$

this assumption is equivalent with the assumption $k!|s$ --read: "k! divides s"--. Let $i = n-k-1$, i.e. c_i is the value on the card to the immediate left of this sequence. Let furthermore $c_i < c_{n-1}$ and let c_j be for j in the range $n-k \leq j < n$ the minimum value such that $c_i < c_j$ (i.e. c_j is the smallest value to the right of c_i exceeding the latter). In that case the operation cardswap(i, j) leaves the rightmost k values in the same monotonic order and our theorem about permutations and their indices tells us that $k!$ is the corresponding increase of s . It also tells us that when besides $k!|s$ we have

$$s \leq r < s + k!$$

c_0 through c_{n-k-1} have attained their final value.

* I therefore suggest to stre$\overset{g}{\wedge}$nthen our original invariant relation P1 with the additional relation P2 --fixing the function of a new variable k --

P2: $1 \leq k \leq n$ <u>and</u> $k!|s$ <u>and</u> $r < s + k!$

which means that the rightmost k cards show still monotonicly increasing values, while the leftmost $n-k$ cards are in their final positions: we have decided upon the "major steps" in which we shall walk towards our destination.

In order to find "the suitable amount" for a major step the machine first determines the largest smaller value of k for which $r < s + k!$ no longer holds -- c_i with $i = n-k-1$ is then too small, but to the left of it they are all OK-- and then increase s by the minimum multiple of $k!$ needed to make $r < s + k!$ hold again; this is done in "minor steps" of $k!$ at a time, simultaneously increasing c_i with cards to the right of it. In the following program we introduce the additional variable kfac , satisfying

P3: $\quad$ $kfac = k!$

and for the second inner repetition i and j , such that $i = n-k-1$ and either $j = n$ or $i < j < n$ and $c_j > c_i$ and $c_{j-1} \le c_i$.

```
s:= 0 {P1 has been established};
kfac, k:= 1, 1{P3 has been established as well};
do k ≠ n → kfac, k:= kfac *(k + 1), k + 1 od
   {P2 has been established as well};
do s ≠ r → {s < r, i.e. at least one, and therefore
            at least two cards have not reached their
            final position}
      do r < s + kfac → kfac, k:= kfac / k, k - 1 od
            {P1 and P3 have been kept true, but in P2
             the last term is replaced by
             s + kfac ≤ r < s +(k + 1)* kfac};
      i, j:= n - k - 1, n - k;
      do s + kfac ≤ r → {n - k ≤ j < n}
            s:= s + kfac; cardswap(i, j); j:= j + 1
      od {P2 has been restored again: P1 and P2 and P3}
od {R has been established}
```

Exercise. Convince yourself of the fact that also the following rather similar program would have done the job:

```
s:= 0; kfac, k:= 1, 1;
do k ≠ n → kfac, k:= kfac *(k + 1), k + 1 od;
do k ≠ 1 →
          kfac, k:= kfac / k, k - 1;
          i, j:= n - k - 1, n - k;
          do s + kfac ≤ r →
                    s:= s + kfac; cardswap(i, j); j:= j + 1
          od
od    .
```

(Hint: the monotonicly decreasing function $t \geq 0$ for the outer repetition is $t = r - s + k - 1$.)