PROGRAMMING DISCIPLINE

O.J. Dahl

Institute of Mathematics, Univ. Oslo, Norway

Abstract.

Good programming discipline is to produce programs which are: easy to use and to understand, reliable and easy to debug (if not already correct), and easy to adapt to changes in the environment. In order to fulfill these requirements programs must well structured and well documented. Research on techniques for program correctness proofs has shed some light on what good structure and adequat documentation is. Indeed a program easily proved correct is easy to understand, and vice versa.

Programming language features and certain mental techniques are aides to produce well structured programs. Discipline is required to obtain good documentation. The latter is even more important.

Short Summary of Proof Techniques.

The idea of program correctness and program proofs used here are those introduced by R.W. Floyd and C.A.R. Hoare. Thus conditional correctness means that a program behaves as specified provided that it terminates properly. The notation

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

where P and Q are logical assertions about program variables (and possibly auxiliary variables), and S is a program statement or statement sequence, means that Q is true immediately after an execution of S, provided P is true immediately before, given that the execution terminates.

$\{P\}$ is called a precondition of S and $\{Q\}$ a postassertion. When embedded in a larger program text an assertion $\{R\}$ specifies the truth of R at that particular program point, in general provided that the precondition of the program was valid on program entry. The following are rules valid for proving the validity of program assertions.

Simple assignment. $\{P^x_e\}\ x:= e\{P\}$

holds for arbitrary P, where P^x_e is obtained from P by textual substitution of e for every free occurrence of x.

Concatenation.

$\{P\}\ S_1\{Q\}$ and $\{Q\}\ S_2\{R\}$ gives $\{P\}\ S_1;\ S_2\{R\}$

Logical Consequence.

$P \supset Q$ and $\{Q\}\ S\ \{R\}$ gives $\{P\}\ S\ \{R\}$,
$Q \supset R$ and $\{P\}\ S\ \{Q\}$ gives $\{P\}\ S\ \{R\}$.

Conditional.

$\{P \wedge B\}\ S_1\{R\}$ and $\{P \wedge \neg B\}\ S_2\{R\}$ gives $\{P\}$ if B then S_1 else S_2 fi $\{R\}$

alternatively

$\{P_1\}S_1\{R\}$ and $\{P_2\}S_2\{R\}$ gives $\{P_1 \wedge B \vee P_2 \wedge \neg B\}$ if B then S_1 else S_2 fi $\{R\}$.

Free loop.

$\{P\}\ S_1\{Q\}$ and $\{Q \wedge B\}S_2\{P\}$ gives $\{P\}$ loop: S_1 while B: S_2 repeat$\{Q \wedge \neg B\}$

Note: This rule must in general be supplemented with additional reasoning in order to prove termination. A sufficient proof might be that a specified integer valued function f of program variables decreases during each execution of S_1 followed by S_2, and $Q \wedge B \supset f \geq 0$. Note also that P or Q (the "loop invariant") cannot in general be constructed from the program text alone, but must be provided as additional information.

Example.

```
{real x = 1 ∧ real y = a ∧ integer d = b ≥ 0}
loop: {x*y^d = a^b ∧ d ≥ 0}
      if odd (d) then x := x*y fi ;
      d := d-2 ;
while d ≠ 0: {d decreases}
      y := y↑2 ;
repeat
{x = a^b ∧ d = 0}
```

for-loop.

Assume that S does not change any of k,a,b. Then

$\{a \le k \le b \wedge R^k_{k-1}\}$ S {R} gives $\{R^k_{a-1}\}$ for k:= a to b: S repeat $\{R^k_c\}$,

where c = max(a-1,b).

This rule follows by applying the free loop rule to the program

k:=a; loop while k ≤ b: S; k:=k+1 repeat

choosing $R^k_{k-1} \wedge k \le c+1$ for P and Q of the free loop rule. Termination is proved by considering the function c + 1 - k.

Subscripted assignment.

Given ⟨type⟩ array a [m:n]; m, n constant, then

$\{m \le k \le n \wedge P^a_{a(k|e)}\}$ a[k]:= e{P}

holds for arbitrary P, where a(k|e) stands for the array value obtained by the assignment

$\forall i\ (m \le i \le n \supset a(k|e)[i] = \underline{\text{if}}\ i=k\ \underline{\text{then}}\ e\ \underline{\text{else}}\ a[i])$.

The alternative notation (a[m:k-1],e, a[k+1:n]) is sometimes useful.

Aggregation of operations.

Let the statement (-list) S contain assignments to the variables $v_1, v_2, \ldots, v_n$ (only). Then functions $f_1, f_2, \ldots, f_n$ of program variables w accessed in S exist, such that

$$\{P\}\ S\ \{Q\}\ \text{gives}\ \{Q^{v_1, v_2, \ldots v_n}_{f_1(w), f_2(w), \ldots f_n(w)}\}\ S\ \{Q\}$$

where $\forall w(P \supset Q^{v_1, v_2, \ldots, v_n}_{f_1(w), f_2(w), \ldots f_n(w)})$ holds. The latter formula expresses what is known about the functions f_i (apart from the fact that they exist).

This important rule allows us to view the total effect of a section of program as a simultaneous assignment of new values to the variables which are (or may be) altered.

Example.

Given the operation swap(x,y) which satisfies $\{R^{x,y}_{y,x}\}$swap(x,y) {R} for arbitrary R. Consider the statement

S ≡ if x < y then swap(x,y) fi

which is equivalent to the concurrent assignment $(x,y) := (f(x,y), g(x,y))$ for definable functions f and g. Choose the postassertion $x = a \wedge y = b$, where a and b are arbitrary numbers. Using the swap rule and the second Conditional rule (with S_2 empty) we prove

$$\{y = a \wedge x = b \wedge x < y \vee x = a \wedge y = b \wedge x \geq y\}\ S\ \{x=a \wedge y = b\}$$

and conclude

$$\forall x,y(y = a \wedge x = b \wedge x < y \vee x = a \wedge y = b \wedge x \geq y \supset f(x,y) = a \wedge g(x,y) = b) .$$

This gives

$$x < y \supset f(x,y) = y \wedge g(x,y) = x, \text{ and}$$

$$x \geq y \supset f(x,y) = x \wedge g(x,y) = y,$$

which defines f and g for all values of x and y. These functions are usually called max and min, thus S is equivalent to

$$(x,y) := (\max (x,y), \min (x,y)) .$$

Procedure call.

The general substitution rule above is valid for arbitrary postassertion R.

$$\{R^{v_1,\ v_2,\ \dots,v_n}_{f_1(w), f_2(w),\dots f_n(w)}\}\ S\ \{R\},$$

which is useful if S is invoked at several places in the program. This leads to the following rule for procedures.

Given <u>proc</u> $p(v_1,v_2,\dots,v_n)$; <u>name</u> $v_1,\dots v_k$;
⟨specification of $v_1,\dots,v_n$⟩ S;

where S does not defer directly to nonlocal variables, and $k \leq n$.

Then

$\{P\}\ S\ \{Q\}$ gives $\left\{R^{a_1,a_2,\dots,a_k}_{f_1(A),f_2(A),\dots f_k(A)}\right\} p(a_1,a_2,\dots a_k,\dots,a_n)\{R\}$,

provided that $a_1,\dots,a_k$ are different variables, and where A is the list $a_1,\dots,a_n$ and $f_1,\dots,f_k$ are as above. The rule is easily extended to procedures with nonlocal variables. It is valid for recursive procedures.

Blocks.

$\{P\}\ S\ \{Q\}$ gives $\{P\}$ <u>begin</u> ⟨declare $v_1,v_2,\dots,v_n$⟩; S <u>end</u> $\{Q\}$, provided that P and Q contain no free occurrences of $v_1,v_2,\dots,v_n$.

Abstraction.

Aggregating operations and data, both at the same time, provide a mechanism of abstraction. Let p be a procedure updating nonlocal variables $v_1,v_2,\dots,v_n$ and whose parameters x are called by value.

<u>proc</u> p(x); ⟨specify x⟩; S;

Then {P} S {Q} gives $\{R^{v_1,\dots,\ v_n}_{f_1(a,v_1,\dots,v_n),\dots,f_n(a,v_1,\dots,v_n)}\}p(a)\{R\}$

for arbitrary R, where $f_1,f_2,\dots,f_n$ satisfy

$$\forall x,v_1,\dots,v_n(P \supset Q^{v_1,\dots,\quad v_n,\quad x}_{f_1(x,v_1,\dots,v_n),\dots,f_n(x,v_1,\dots v_n),g(x,v_1,\dots,v_n)}).$$

We collect the procedure p and the variables $v_1,v_2,\dots,v_n$ by a class declaration.

<u>class</u> C;
<u>begin</u> ⟨declare $v_1,v_2,\dots,v_n$⟩;
<u>proc</u> p(x); ⟨specify x⟩; S;
<u>end</u> of C;

Given C <u>var</u> V; which declares an instance named V of the class body, we may take V to be a variable of an abstract type C, represented by the variables $v_1,v_2,\dots,v_n$, and whose abstract value is a function of the latter, the "abstraction function", [3].

$$V = F(v_1,v_2,\dots,v_n)$$

The procedure p, local to V, is an abstract operator updating the abstract value of V. We use the notation V·p(a) for invoking the operator. Then the rule

(*) $\{R^V_{f(V,a)}\}$ V.p(a) {R}

holds for arbitrary R, where

$$f(V,a) = F(f_1(v_1,\dots,v_n,a),\dots,f_n(v_1,\dots,v_n,a))$$

and $f_1,\dots,f_n$ are as above.

Often the abstraction function F is meaningful only if a certain invariant relation I holds for the arguments $v_1,\dots,v_n$. The invariant I may be established initially by statements S' in the block tail of C, and I must be preserved by p.

Then the rule (*) is established by proving

$$\{P\wedge I\}\ S\ \{Q\wedge I\} \quad \text{and} \quad \{P_0\}\ S'\ \{Q_0\wedge I\}.$$

Furthermore $\{P_0\}$ C <u>var</u> V $\{V=V_0\}$ is true provided $Q_0 \wedge I \supset F(v_1,\dots,v_n) = V_0$. It is assumed that the variables $v_1,\dots,v_n$ are not updated textually outside C, except through invoking the local procedure p.

Informal examples of abstraction are given in [4], pp. 205-208, and in the following section.

References.

[1] C.A.R. Hoare: An axiomatic basic for computer programming. CACM 1970.

[2] C.A.R. Hoare, N. Wirth: An axiomatic definition of Pascal, ETH 1972.

[3] C.A.R. Hoare: Proof of correctness of data representation, Acta Informatica 1972.

[4] O.-J. Dahl, E.Q. Dijkstra, C.A.R. Hoare: Structured Programming. Academic Press, 1972.

Bottom-Up Construction, an Illustration.

Problem: Process sequence of telegrams for accounting purposes. (Cf. Henderson and Snowdon: An experiment in structured programming, BIT 12,1 (1972) pp. 38-53.) Each telegram should be printed out and in addition its number of words should be counted and printed, and a warning message should be given if any of its words is longer than K characters. Each telegram ends with the word ZZZZ. The words STOP and ZZZZ do not count. The sequence ends with a telegram containing no countable words.

The telegrams are stored on an input medium as a record sequence. Each record contains N characters. No word is divided across records, and blanks are used for filling up. The same rules apply to the output medium. Output records have length M.

Given: the type char (character value) with the operators =, ≠ , and the following I/O-mechanisms.

proc read (A); char array A;
which reads the next input record into the first N positions of A , where the length of A is N or more.

proc print (A); char array B;
which outputs an output record from the first M locations of A, possibly padded with blanks if the length of A is less than M.

A string notation is available for char array constants. Also the equality operator is assumed to apply to character arrays.

```
class incharseq;
{An input character sequence is formed by "concatenating" the
records of the input file, each extended by a blank character.}
begin char array buf [1:N+1]; int i;
    {i points to the current character of buf, which contains
    the current record.}
    char proc c; c:=buf[i]; {the current character}
    proc adv; {advance to the next character}
        if  i ≤ N then i:=i+1
        else read(buf); i:=1 fi;
    {The initial character is a simulated blank considered the last
    character of a mythical record preceding the input sequence.}
    i:=N+1; buf[i]:=blank
end of incharseq;

class string(n); int n;
begin char array w[1:n]; int lg;
    {A string of length lg is contained in w[1:lg], where
    0 ≤ lg ≤ n}
    proc clear;lg:=0;
    proc add(x); char x;
        if lg ≥ n then error ('string overflow')
        else lg:=lg+1;  w[lg]:=x fi;
    clear {a string is empty initially}
end of string;

class inwordseq(n); {time sequence of words from input}
begin string(n) var word; incharseq var inc;
    {word contains the current word read from inc}
    proc adv; {advance to next word}
        begin word.clear;
              loop while inc.c=blank: inc.adv repeat;
              loop: {collect letters,including trailing blank}
                    word.add(inc.c);
```

```
            while inc.c ≠ blank: inc.adv; repeat
    end of adv;
end of inwordseq;

class outwordseq; {sequence of words for output}
begin char array buf[1:M+1]; int i;
    {buf[1:i] has been filled. buf[M+1] can only be filled with
    a redundant trailing blank}
    proc throw; {output buffer, unless empty}
        if i > 0 then
        for j:= i+1 to M: buf[j]:=blank repeat;
        print [buf]; i:=0
        fi;

    proc out(s); string val s;
        begin if i+s.lg > M then throw fi;
              for j:= 1to s.lg:
                  i:= i+1; buf[i]:=s.word.w[j]
              repeat
        end;
i:=0 {empty buffer initially}
end of outwordseq;

Main program:
begin inwordseq(50) var Wi; outwordseq var Wo;
     int wcount; Bool longw;
     {wcount
     loop: {zero or more telegrams have been processed}
          wcount:=0; longw:=false;
          {start processing another}
          loop:{zero or more words have been read of the current
               telegram. wcount of them were countable. longw
               means one or more were too long}
               Wi.adv;
          while Wi.word.≠ 'ZZZZ':  Wo.out(Wi.word);
               if Wi.word ≠ 'STOP' then wcount:=wcount+1 fi;
               if Wi.word.lg > K then longw:= true fi;
          repeat;  {another telegram has been processed}
     while wcount ≠ 0:
          Wo.throw; printnum(wcount);
          if longw then print ('warning message') fi;
     repeat
end of main program
```

Reading List.

Books.

O.-J. Dahl, E.W.Dijkstra, C.A.R.Hoare:
Structured Programming. Academic Press, 1972.

G.Birtwistle, O.-J. Dahl, B.Myhrhaug, K.Nygaard:
SIMULA BEGIN. Studentlitteratur & Auerbach 1973.

O.-J. Dahl, D.Belsnes:
Algoritmer og Datastruktur. Studentlitteratur, 1973. (In Norwegian).

P.Brinch-Hansen:
Operating System Principles. Prentice-Hall, 1973.

Short Selection of Articles:

E.W. Dijkstra:
The Structure of The Multiprogramming System. CACM 11,5,pp.341-346, (May 1968).

E.W. Dijkstra:
Goto statement considered harmful. CACM 8,9,pp.147-147 (Sept.1968).

R.W. Floyd:
Assigning meanings to programs. Proc. of Symposia in Applied Mathematics, vol.19, pp 19-32 (1967).

P.Hendersen and R.Snowdon:
An experiment in structured programming. BIT 12, pp. 38-53 (1972).

C.A.R.Hoare: An axiomatic approach to computer programming. CACM 12, 10, pp.576-580, 583 (Oct. 1969).

C.A.R. Hoare:
Proof of a program: FIND. CACM 14,1, 39-45 (Jan.1971).

C.A.R. Hoare:
Proof of the correctness of data representations, Acta Informatica 1, pp. 271-281 (1972).

C.A.R. Hoare and N.Wirth:
An axiomatic definition of the programming language PASCAL. Acta Informatics 2, pp.335-355 (1973).

D.E. Knuth:
A review of "Structured Programming". Stanford Computer Science Department report STAN-CS-73-731 (June, 1973).

Miller:
The magical number 7 plus or minus two: Some linits to our capacity for information processing. Psychol. Rev. 63, pp.81-87.

P.Naur:
Programming by action clusters. BIT 9, pp.250-258 (1969).

P.Naur: An experiment on program development. BIT 12, pp.347-365 (1972).

A.Wang and O.-J. Dahl:
Corontine sequencing in a block structured environment BIT 11, pp. 425-449 (1971).