INFORMATION PROCESSING 80, S.H. Lavington (ed.)
North-Holland Publishing Company

A BASIS FOR THE CONSTRUCTIVE APPROACH TO PROGRAMMING

Bertrand MEYER
Electricité de France, Direction des Etudes et Recherches (service IMA)
1, avenue du Général de Gaulle, 92141 Clamart, France

Although programming is a difficult and creative activity, useful strategies and heuristics exist for solving programming problems. We analyse some of the most fundamental and productive among them; their knowledge and conscious application should help the programmers in constructing programs, both by stimulating their thinking and by helping them to recognise classical situations. The precise framework for the analysis is provided by the specification language Z. For editorial reasons the description in some sections of this paper has had to be curtailed.

1. THE NEED FOR A CONSTRUCTIVE APPROACH TO PROGRAM ANALYSIS

1.1 Introduction

The evolution of the various domains of computer science has led to the development of powerful program analysis methods. They make it possible to study many properties of problems and programs; e.g., to determine whether a problem can be solved at all, and, if so, whether there exist realistic algorithms; to evaluate the abstract and concrete complexity of a program; and to prove it correct relative to some specification.

Useful as these techniques may be, they do not provide a completely satisfactory answer to the practicing programmer, whose immediate concern is to *build* programs which will solve given problems. Programming is a difficult intellectual activity, and it can hardly be expected that straightforward "methods" will ever be discovered, let alone "algorithms", to deduce programs from problems. To anyone seriously concerned with programming, however, it is obvious that certain fruitful thought patterns do recur with a remarkable frequency, and it is quite a temptation to try to analyze and formalize them with the hope that their knowledge will be of some help for those who construct programs. Such is the aim of this paper.

Several authors have remarked that program proving techniques are of less use for proving the correctness of existing programs than as tools to help the programmers write programs which will be correct in the first place. An important work in this direction is that of Dijkstra [5]. We will try to elaborate on these methods and give a precise basis for their application.

For any precise reasoning about programming, the use of a formal notation is unavoidable. We will rely on such a notation, the "Z" specification language, the essentials of which are summed up in section 2. Section 3 is devoted to the presentation of our framework for the constructive study of programs, and a brief comparison with other approaches. Section 4 contains several significant examples. In section 5, we analyze the scope and implications of the concepts and techniques discussed.

Before going into detailed analysis, it may be useful to set the general tone of the work by informally introducing some of the ideas on a toy example. More serious examples are treated in section 4.

1.2 A toy example

Assume we are looking for an algorithm to compute square roots. The problem is to find a method which, given any $a \geq 0$, will yield x such that

$$x \geq 0 \ \underline{and} \ x * x = a \qquad (1)$$

Looking at the form of eq. (1), we decide to try a heuristic called "uncoupling" (see 4.1), which roughly suggests that "one variable appearing twice may be replaced by two equal variables", i.e. here eq. (1) may be replaced by

$$x \geq 0 \ \underline{and} \ y \geq 0 \ \underline{and} \ x * y = a \ \underline{and} \ x = y \qquad (2)$$

The heuristics used also suggests that we call "invariant" the conjunction of the first three clauses in eq. (2), and "goal" the last one $(x = y)$, and look for an algorithm of the form:

```
establish invariant;
while not goal do
        begin
        get x and y closer to each other;
        restore invariant
        end
{goal and invariant} {i.e., the desired
                      conclusion}
```

establish invariant is readily implemented by the assignments

$$x := 1 \ ; \ y := a$$

get x and y closer to each other may be chosen (among many other possibilities : this is probably the central design decision) as :

$$x := (x + y)/2$$

restore invariant may then be :

$$y := a/x$$

What we have obtained is the classical algorithm known as Newton's method; it is indeed easily seen that the sequence of values taken by x satisfies $x_{n+1} = (x_n + a/x_n)/2$. Of course, one must show that the algorithm terminates in a finite number of steps (adding to "$x = y$" the mention "within a prescribed margin of accuracy"), i.e. that $|y_n - x_n|$ is a converging sequence.

After this example which shows that few "creative" decisions may be needed in order to discover a good algorithm, we come to the description of the notation used in the rest of this paper.

2. A NOTATIONAL BASIS : THE Z SPECIFICATION LANGUAGE

In order to precisely define the constructive meaning of programming structures, we need a notation which is both formal and readable. Moreover, it should be purely static, i.e. involve only well-known mathematical constructs.

The "Z" specification language satisfies these requirements. This language has been used [1] to model all kinds of information processing problems, ranging from text editing to "on-the-fly" garbage collection, system problems, programming language semantics, business data processing problems etc. Z is based on formal set theory and logic; it uses a notation similar to that of programming languages, and has in particular been influenced by the syntax of ADA [8].

We shall in no way attempt a complete description of Z, which is to be found in [1]. For further discussion and examples see [9].

A Z text is divided into "chapters". As an example which will introduce notions used below, we write a small chapter describing order relations, and in particular "well-founded" order relations, also called "noetherian" [3].

```
STRICT_ORDER ≙

    chapter <some basic chapters; see [1]> def
    -- transitive, irreflexive and order
    -- relations :
    trans[X] ≙ set r for r : X ↔ X where
       (r ∘ r) ⊂ r end;
    irreflex[X] ≙ set r for r : X ↔ X where
       r ∩ id[X] = null end;
    strict_order[X] ≙ trans[X] ∩ irreflex[X];
    integer_order ≙ theorem op(<) ∈ strict_
       order[NAT] end;
    minimum[X] ≙ rel A, r ↔ m for
                      A : subset(X);
                      r : strict_order (X);
                      m : X
                   where
                      A ⊂ r(m) -- i.e. r(m↔a)
                        --for all a in A
                   end;
    strict_well_founded[X] ≙
                   set r for r : strict-order[X]
                   where
                      forall A for A : subset(X)
                        then minimum (A,r) ≠
                        null
                      end
                   end;

-- Let f be a function from X into X and n a
-- NAT. Then iter(n) (f), defined in another
-- chapter, is the n-th iterate of f.
-- A being a subset of X, f ⌈A is the restric-
-- tion of f to A.
converge[X] ≙ theorem -- see reference[3],
                      -- III.51, prop. 6.
              forall A, f, a for
                      A : subset(X) - {null};
                      f : (X → X);
                      a : A
              where -- f on A is the inverse
               --of a strict well founded
               --relation :
                      f ⌈A ∈ inv(strict_well_
                        founded[X]);
              then
                      exist n for n : NAT where
                           iter(n) (f) (a) ∉ A
                      end
              end;

limit[X] ≙ func A, f → g for
                   A : subset(X) - {null};
                   f : X → X;
                   g : A → X - A (-- A's comple-
                                  -- ment)
           where
                   f ⌈A ∈ inv(strict_well_
                     founded[X]);
           then
                g ≙ func a → b for a : A;
                                   b : X-A then
                     b ≙ iter(n) (f) (a)
                given
                     N ≙ set m for m : NAT
                       where iter(m) (f)
                       (a) ∉ A
                       end;
                     n ≙least (N)
                       -- smallest elt.
                proof

                     N ≠ null from
                     theorem "converge"

                end
           end
end
```

3. A FORMAL BASIS FOR THE CONSTRUCTION OF PROGRAMS

3.1. Guidelines

The framework for our representation of programs is the concept of a solution to a programming problem, which will be characterized by :

- two(generic) sets : the input set I and the output set O;

- a relation on $I \times O$, which is the problem to be solved;

- a function from I to O, which represents a program "implementing" the relation. Programming is the search for explicit functions compatible with relations given in implicit form.

This is readily expressed in Z by a class which we call *solution* :

```
solution[I, O] ≙ class with
                     problem : X ↔ Y;
                     program : X → Y
                 where
                     program ⊂ problem
                 end
```

In this approach, any constructive theory of programming may be considered as a set of rules for constructing solutions for certain programs, and deducing solutions to new problems from solutions to simpler ones.

3.2 Sequence

The rule for program composition is as follows :

```
sequence [I,X,O] ≙
    func s1, s2 → s for
            s1 : solution[I,X];
            s2 : solution[X,O];
            s  : solution[I,O]
    then
            s ≙ cons solution [I,O] with
                problem ≙ problem (s2)
                  ∘ problem(s1);
                program ≙ program (s2)
                  ∘ program(s1)
              proof
                program is a function and
                  program ⊂ problem
              end
    end
```

The constructive interpretation of this rule could be phrased as : if you can't go there directly, then go indirectly. The rule is a functional equivalent of Hoare's and Dijkstra's axioms for composition [5][7].
Note the *proof* clause used to justify the *cons*.

3.3 Choice

The two-way choice, or alternative, may be described as follows :

```
alternative[I,O] =
    func s1, s2, A → s for
       A : subset (I);
       s1 : solution [A,O]; s2 : solution [A',O];
       s  : solution [I,O]
    then
       s ≙ cons solution [I,O] with
           problem ≙ problem (s1)
             ∪ problem (s2);
           program ≙ program (s1)
             ∪ program (s2)
         proof
           program ∈(I → O) and program
             ⊂ problem
         end
    given
       A' ≙ I - A
    end
```

The constructive interpretation of this rule is that when looking for a solution to a programming problem with I as input domain it may be useful to look for partial solutions defined on disjoint subsets of I. The rule could of course be generalized to a partition involving more than two subsets. It does not, however, gracefully generalizes to Dijkstra's non-deterministic *if* ... *fi* construct. This is quite natural since we stated from the beginning that we were looking for functions.

3.4 Loops

We choose to model the so-called *while* loop. It is well-known from the work of Floyd [6] and Hoare [7] that one of the basic concepts in connection with loops is that of "invariant" or "inductive assertion". However, invariants are often presented after the construction of a loop; on the other hand, in [5], although invariants are used throughout as a constructive technique, the corresponding rule is not an axiom of the proposed semantics, but a consequence of the axioms for loops. It seems to us that the concept of invariant is so important that it should be part of the definition of loops. As we shall see, this can be done in a simple way; loop initialization and loop invariant turn out to be one concept.

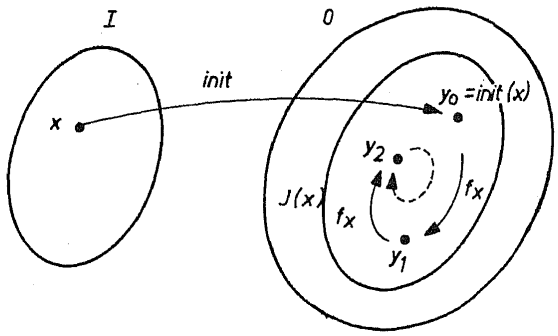
Before we turn to the formal description, it is useful to present a closely related mathematical analogy which gives much insight into the essence of loops. Let $p(x,y) = 0$ be an equation where x is given and y is the unknown. x and y are usually vectors. Under certain conditions, this equation may be transformed into the fixpoint equation $y = f(x,y)$ where $f(x,y)$ is defined as $y + Ap(x,y)$ A being some linear application. This equation is solved by taking the limit of the sequence y_n, defined by :

$$y_0 = init(x);\ y_{i+1} = f(x,\ y_i)$$

For the sequence to converge in a certain domain $J(x)$, f must satisfy a certain condition, called the Lipschitz condition, under which for all y, y' in $J(x)$:

$$|f(x,y') - f(x,y)| < k_x\ |y' - y|$$

for a value $k_x < 1$ independent of y. The choice of A helps meet this requirement. Moreover, the initialization function *init* must ensure that $init(x)$ belongs to the convergence domain $J(x)$.



In programming a loop is a fixpoint computation method which generalizes the above scheme. It has an initialization part *init* which ensures the initial validity of the invariant assertion; the latter expresses membership in the "convergence domain" $J(x)$. Then the loop has a body which is a transition function f_x; for all x, the transition function has the property that $f_x(y)$ belongs to $J(x)$ if y does (except perhaps when y is equal to the limit of the sequence), i.e. that the invariant is indeed invariant under f_x; and that f_x "converges" i.e. satisfies a kind of "Lipschitz" condition usually expressed by an associated variant function v_x such that $v_x(f_x(y)) < v_x(y)$ and the range of v_x is a strict well-founded set; of course, programming, unlike classical analysis, usually requires that sequences reach their limits.

With this suggestive analogy in mind, we are ready to express the loop as a constructive problem-solving method.

```
loop[I,O] ≙
    func start, body, exit → ℓ for
        start, ℓ: solution[I,O];
        body    : solution[I × O, O];
        exit    : I ↔ O
    where
        inv : transition (invariant-exit)
              ⊂ invariant;
        dec : transition ∈ inv(strict well_
              founded[invariant-exit]);
    then
        ℓ ≙ cons solution[I,O] with
              problem ≙ invariant ∩ exit;
              program ≙ proj2
                  ∘ limit (transition) ∘ init2
            proof
              theorem "converge", section 2.6
            end
    given
        invariant ≙ problem (start);
        transition ≙ proj1[I,O] & program (body);
        initialization ≙ program (start);
        init2 ≙ id[I] & initialization
    end
```

The constructive interpretation of this rule is that it may be useful to try and express the goal as the conjunction of two relations *invariant* and *exit*, and solve the new problems thus obtained in a quite dissymetric way : no strategy is implied for obtaining *invariant* (i.e. the initialization), whereas *exit* is to be reached by a fixpoint method keeping *invariant* true right up to the end.

3.5 A few remarks

The above functional definition corresponds to a programming language construct of the following form (in PASCAL notation) :

```
var i : I {input},
    o : O {output};
.....................
o := initialization(i);{invariant (i,o) is
     true}
while not exit (i,o) do
  o := transition (i,o);
{invariant (i,o) and exit(i,o)}
```

In view of the key role played by the initialization in any loop, as evidenced by the above Z model, it seems regrettable that initialization is not syntactically part of the loop in common programming languages. It is well-known that omission of the initialization part is both a frequent and a serious programming error. It may thus seem advisable to include it in the syntax for loops, giving something like :

```
from
    <initialization part>
until
    <exit condition>
keeping
    <invariant assertion>
loop
    <transition>
end;
```

The "*keeping* ..." clause should be optional since one cannot expect all programmers to use formal methods.

4. STRATEGIES FOR PROGRAM CONSTRUCTION AND EXAMPLES

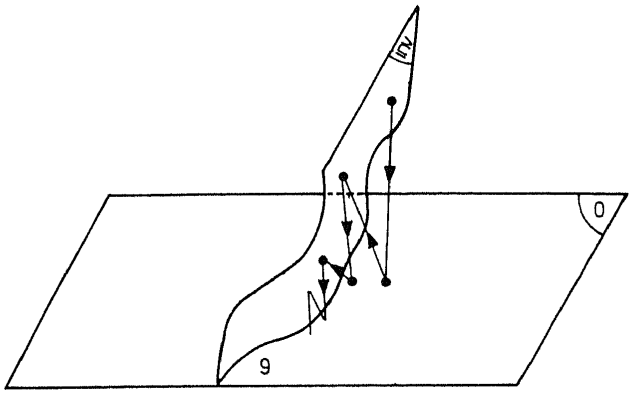
4.1 Embedding, constant relaxation and uncoupling

One of the lessons we draw from the previous section is that loops may be considered as a program construction strategy whereby the goal is expressed as the conjunction of an "invariant" which is easier to establish than the goal itself, and an "exit" condition, in such a way that a "transition" function can be found, which will not destroy the validity of the invariant while decreasing a "variant" function so long as the exit condition is not satisfied.

It is easy to find many prototypical examples which fit nicely into this framework. For instance the "simplex" algorithm is nothing else [4] than keeping a certain point on the edges of a convex polyhedron (the invariant relation) while minimizing a cost function (the variant). Note that the latter may not be decreasing everywhere, which is a well-known

theoretical problem in linear programming. On the other hand, so-called "relaxation methods" in numerical analysis vary the shape of a certain surface which is kept isomorphic to itself (invariant) and gradually decrease its energy function (variant).

A basic problem of program construction is : how do we weaken the goals to get the invariants ? The general method may be called embedding : use a larger domain D, a subset of which is isomorphic to the output set O. Find a relation *inv* defined on D which is easier to establish on D than the goal g is on O, but such that *inv* implies g on O. Then use *inv* as invariant, and membership in O as the exit condition. The method can be visualized a succession of "frog leaps" between the "surface" *inv* and the "hyperplane" O in D (see figure).



Many algorithms are direct applications of embedding. For example, *for* loops, such as operations on matrices, usually set out to solve a problem on the set of (n,n) matrices by embedding it in the set of (i,i) matrices for $0 \leq i \leq n$. Initialization is usually trivial; transition adds one to the dimension.

A particular case is "constant relaxation" : replace the goal $P(n)$, where n is a constant belonging to some set X, by $P(i)$ *and* $i = n$, where i is a variable constrained to range over X . This is also typical of *for* loops : to compute $s = \sum_{k=1}^{n} a[k]$, we replace this goal by $s = \sum_{k=1}^{i} a[k]$ *and* $i = n$ and let i range from 0 to n.

Another closely related heuristics in uncoupling, which applies to a goal of the form $P(\ldots, i, \ldots, i, \ldots)$ where i appears twice, replacing it by $P(\ldots, i, \ldots, j, \ldots)$ *and* $i = j$. We saw a simple example of this method in section 1.

It is surprising to see how many loop algorithms may be recognized as instances of the latter two variants of embedding. In fact, it proves quite hard to find loop algorithms which escape these categories - which is rather disappointing when one has embarked upon a tentative classification of useful heuristics. However, these two are useful without any doubt, and we shall now conclude by analysing the way they apply to two examples: array partitioning, and the QR algorithm for computing matrix eigen-values. For further examples, see [9].

4.2 Array partitioning

A straightforward case of uncoupling is Hoare's method for partitioning arrays, as used in Quicksort. If we use the first element as the pivot [13], the problem is to establish for some s in $i..j$:

```
forall k for k : i + 1 .. j then
        k ≤ s => a(k) ≤ a(i)
        s+1 ≤ k => a(k) ≥ a(i)
end
```

Uncoupling the two clauses with respect to s, i.e. replacing s by t in the second one to get the invariant, will yield an algorithm schema of the form

```
s := i ; t := j ; {invariant is true}
while s ≠ t do
        "get s closer to t, maintaining the
           invariant"
```

To represent the quoted statement, the partitioning method moves s and t towards each other, then restores the invariant :

```
begin
while s ≠ t and a(s) ≤ a(i) do
        s := s+1 ;
while t ≠ s and a(t) ≥ a(i) do
        t := t-1 ;
{t = s or (a(s) > a(i) and a(t) < a(i))}
exchange elements a(s) and a(t)
end
```

A variant is the algorithm for the "Dutch National flag" problem [5].

4.3 The QR algorithm for computing matrix eigenvalues

We turn now to a quite difficult numerical algorithm. Assume we wish to compute the eigenvalues of a matrix a. A possible course of action is based on the following two properties of eigenvalues :

1. The eigenvalues of a and b are the same if a and b are similar matrices.
2. Eigenvalues are particularly easy to compute for some classes of "good" matrices, e.g. orthogonal and triangular ones.

Algorithms which for given a compute a "good" b similar to a will thus yield the solution. The subproblem may be expressed as finding b and s (the similarity matrix) such that b is "good", s is regular, and

$$b = s^{-1} a s$$

If we write this as $sb = as$, uncoupling with respect to s is once again very tempting. Knowing that a relatively simple algorithm is known for factoring (i.e. for given m find regular s, and "good" r, such that $sr = m$), we are led to an algorithm of the following form :

```
s := 1 ; t := 1 ; b := 1 ;
while sb ≠ at do
        begin
        (s,b) := factoring (at); *;
        t := s
        end
```

(Note that here $s = t$ is the invariant and $sb = at$ the goal. The reverse choice would also work). Now if we define, in location marked *, q as $t^{-1}s$, we recognize an efficient algorithm known as QR or LR depending on the class of "good" matrices chosen (resp. orthogonal or triangular) [14]. This algorithm computes :

$$q_0 r_0 \triangleq a$$
$$q_1 r_1 \triangleq r_0 q_0$$
$$\text{---------}$$
$$q_i r_i \triangleq r_{i-1} q_{i-1}$$

which converges towards a pair (q_i, r_i) where r_i and q_i are "good" matrices and $r_i q_i$ is similar to a.

Of course the method shown only yields an algorithm schema; a proof of convergence, which is mathematically far from trivial, is required. It looks remarkable, however, that such a "technical" algorithm may be obtained through the application of very general rules.

5. CONCLUSION

We hope to have shown that basic programming concepts such as control structures may be described in a simple way, using no particular mathematical apparatus other than well-known notions such as sets, relations, functions, partitions, orders, etc.; that this can be done in a clear and persuasive way thanks to the use of a rigorous yet readable formalism, namely Z; and that such a description paves the way for expression of powerful mechanisms which are basic in the design of algorithms. As was mentioned before, we do not mean to imply in any way that programs can be invented through application of recipes of any kind. The rules presented here do however provide much insight, as it seems to us, into the structure of programs; they should be part of any set of rules used in work toward program synthesis. These methods, as well as the general formalizing approach presented here, have proved helpful both in teaching programming, and in looking for new algorithms.

ACKNOWLEDGEMENT

Many ideas come from numerous discussions with J.R. Abrial and A. Bossavit.

REFERENCES

[1] J.R. Abrial, S.A. Schuman and B. Meyer, *Specification Language;* Proceedings of School on Program Construction, Belfast; Cambridge : Cambridge University Press, 1980.

[2] A. Bossavit and B. Meyer, *On the Constructive Approach to Programming : The Case for Partial Cholesky Fastorization (A Tool for Static Condensation)*; in Advances in Computer Methods for Partial differential Equations III, Vichnevetsky and Stepleman (Eds.), IMACS, 1979.

[3] N. Bourbaki, *Eléments de Mathématiques - Théorie des Ensembles;* Paris : Hermann, 1970.

[4] G.B. Dantzig, *Linear Programming and Extensions;* Princeton (N.J.) : Princeton University Press, 1963.

[5] E.W. Dijkstra, *A Discipline of Programming ;* Englewood Cliffs (N.J.) : Prentice-Hall, 1976.

[6] R.W. Floyd, *Assigning Meaning to Programs;* Proc. Sym. in Applied Mathematics 19, Schwartz J.T. (Ed.), American Mathematical Society, pp. 19-32.

[7] C.A.R. Hoare, *An Axiomatic Basis for Computer Programming;* Communications ACM, 12, 10, pp. 576-583.

[8] J.D. Ichbiah et al, *Preliminary ADA Reference Manual; Rationale for the Design of the ADA Programming Language;* SIGPLAN Notices, 14, 6, Parts A and B.

[9] B. Meyer, *The Z Language as a basis for the Constructive Approach to Programming;* Internal Report, Electricite de France, Direction des Etudes et Researches, May 1980

[10] B. Meyer and C. Baudoin, *Méthodes de programmation*; Paris : Eyrolles, 1978.

[11] J.T. Schwartz, *Program Genesis and the Design of Programming Languages;* in Current Trends in Programming Methodology, Vol. IV, *Data Structuring*, Yeh (Ed.), pp. 185-215; Englewood Cliffs (N.J.) : Prentice-Hall, 1978.

[12] D.S. Scott, *The Lattice of Flow Diagrams;* in Symposium on Semantics of Programming Languages, Engeler (Ed.), Lecture Notes in Mathematics, pp. 311-366; Berlin : Heidelberg : New-York : Springer-Verlag, 1971.

[13] R.S. Sedgewick, *Quicksort;* Ph. D. Thesis, Stanford University, 1975.

[14] J.H. Wilkinson and C. Reinsch, *Linear Algebra (Handbook for Numerical Computation, vol. 2)*; Berlin : Heidelberg : New York : Springer-Verlag, 1971.