AB23.3.2. Further Thoughts on Record Handling AB21.3.6.

C.A.R. Hoare

1. Introduction

The concepts of record handling are comparatively novel and unfamiliar in the field of general purpose programming languages, and it is desirable that they should be discussed, refined, and extended before they reach their fullest usefulness. This paper is intended to initiate and contribute to such discussion, rather than to present final solutions to the problems involved.

The numbering of the sections is not continuous, but reflects the numbering of sections of AB21.3.6. In some cases, the sections are new.

2.3. References

The concept of a reference as a unique representative of a record is, in fact, superfluous; everything which is said of the reference could equally well have been said of the record itself. For example, instead of saying "the value of the variable is a reference to a record which..." one could have said "the value of the variable is a record which...". A reference-valued expression" or field can now be described as a "record-valued expression" or a "record-valued field", etc.

3. Example

A less trivial example of record handling has been kindly sent me by E.W. Dijkstra, and deserves wider publicity:

The problem is to find the shortest route on a map between two towns, referenced by "start" and "finish". It is assumed that the map is given in terms of direct town to town connections (roads), each of which has a known length and destination. Note that two-way roads must feature twice. Each town (In general) has a first road leading out of it, which is referenced by the field "first out". Since the town may have more than one road leading out, each road has a field "next out" which references the next road leading out of the same town, if any, and is otherwise null. All the variables and fields mentioned above are assumed to have had their values assigned before the start of the algorithm.

During the execution of the program, the "distance" field of a town indicates its least known distance from the start, and the "previous" field refers to the town that would be visited first on the shortest known route to the start. The field "fol" is used to build up a simple chain of towns, which falls into two halves: the first half starts with the town "start", and contains in order of least distance, the nearest towns. These towns are said to have been placed, and the last member of this half of the chain is referenced by "last placed".

The second half of the chain consists of unplaced towns, and the last town of this half is referenced by "end". The second half contains, in arbitrary sequence, all towns which have a direct connection with any of the placed towns. Furthermore, for each unplaced town, every placed town will have been previously considered as a possible town of first visit on the route back to the start. Now it is readily seen that the first town visited on the route back from the true (n+1)th nearest town must be one of the n nearer towns already placed, so that it must now be valid to place that one of the unplaced towns which has the shortest known distance from start. The algorithm repeats this process until the last placed = finish, in which case the solution can readily be traced through the "previous" fields of the towns; but if there is no route leading from finish to start, the algorithm sets last placed = null and terminates.

```
record class town;
     begin reference first out (road), fol, previous (town);
          integer distance end;

record class road;

     begin reference destination (town), next out (road);
          integer length end;

reference start, finish, scanned, prescanned, last placed, goal, premin,
     end (town), trial (road);

end := last placed := start; fol (start) := previous (start) := null;
distance (start) := 0;
while last placed ≠ finish and last placed ≠ null do
begin trial := first out (last placed);
     while trial ≠ null do
     begin comment examine every road leading from last placed;
     goal := destination (trial); scanned := start;
          while scanned ≠ goal do

          begin comment make sure goal is on chain;
          if scanned = end then
          begin comment if not, put it at end of unplaced towns;
               fol(end) := goal; end := goal; fol(goal) := null;
               distance (goal) := + infinity
          end;
          scanned := fol (scanned)
     end;
```

```
                if distance (last placed) + length (trial) < distance
                    (goal) then
                begin comment the last placed town gives a better route than
                    before; distance (goal) := distance (last placed)
                    + length (trial);
                    previous (goal) := last placed
                end;
                trial := next out (trial)
            end;
        if last placed = end then
        comment there are no further towns accessible from any of the placed
            towns;
            last placed := null
                           else
        begin premin := last placed; prescanned := fol (last placed);
            while prescanned ≠ end do
            begin comment find town with best distance;
            if distance (fol (prescanned)) < distance (fol (premin))
                then
                premin := prescanned;
                prescanned := fol (prescanned)
            end;
            if premin ≠ last placed then
            begin comment change its position to the end of the first half of
                the chain;
                scanned := fol(premin);
                fol (premin) := fol (scanned);
                fol (scanned) := fol (last placed);
                fol (last placed) := scanned
            end;
            last placed := fol (last placed)
        end
end
```

5.5. The AED Project

D.T. Ross has kindly pointed out that the defects attributed to AED-0 were fully recognised in the description of that language; they were accepted only as an interim measure, and have been removed in the subsequent development of AED-1. In particular, AED-1 contains the concept of a record class, and can check the validity of field designators at compile time. It also uses the record class identifier as a record-creating function. In fact, AED-1 seems to have forestalled all the major features of the record handling proposal.

5.7. SIMULA (O.J. Dahl and K. Nygaard).

The concepts of record handling also feature in the SIMULA language, which is an extension of ALGOL 60 oriented towards simulation studies. A record class corresponds to a SIMULA "activity", a record corresponds to a "process", and a field to a local variable of the process. References as such do not feature directly in SIMULA; instead there is a general facility for linking records into sets. The element/set mechanism is a useful one, especially in simulation studies, but it can readily be constructed from the more fundamental reference mechanism; and it is probably better in a general purpose language to provide the fundamental building blocks than to incorporate a built-in mechanism which is essentially more complex.

5.8. PL/I (IBM)

The list processing facility recently introduced into PL/I is quite similar to record handling. A reference corresponds to a "pointer" in PL/I, and a record to a "major structure" of controlled storage class. The field designator "father (B)" would be written in PL/I as "B->FATHER", and the record creating assignment "B := person" would be written "ALLOCATE (PERSON) SET(B)".

The PL/I pointer suffers from the same difficulties as the EULER reference (see 5.3). In addition, PL/I provides a further potential source of error and confusion, in that a field name may be used as if it were an ordinary variable without specifying explicitly which record is being referred to; the translator then makes an implicit assumption about what record is intended.

The trouble is that the programmer can change the identity of this implicit record, with results which are sometimes unpredicted, and always difficult to trace.

6. Implementation

Fully dynamic random storage allocation for groups of consecutive locations of any size is provided both in the implementation of SIMULA and in that of AED-0. The proven success of the methods used in these languages is a strong recommendation for their adoption in the implementation of record handling. Neither method involves or requires "Store collapsing", but (in SIMULA at least) automatic garbage collection is possible.

6.6. Random Access Backing Store

The use of a random access backing store to simulate large single-level stores will, on most computers, involve very much greater inefficiencies than if all records were confined to main core storage. In view of this, it seems essential to give the programmer some control over whether a record is to reside on backing store, and be called into main store only when required, or whether it is to remain permanently in main store. Furthermore, for records residing on backing store, the programmer should be given some control over the grouping of the records, so that he can ensure, for example, that records which are likely to be referred to in quick succession of each other shall be situated on the same track of the drum or disc.

This information should be given at the time when the record is first created by means of a record constructor. It is suggested that the record constructor should be preceded by:

bs or bs (<integer expression>)

in cases where the record is to be allocated on the backing store. In the second option, the integer expression specifies the number of track required.

7.4. Initial Values

The proposal for initial values given here seems to be over-elaborate and mistaken. It would be much better to adopt the following approach:

A record constructor is a function designator which yields as value a reference to a newly created record. It consists of a record class identifier which specifies the class of the new record, optionally followed by an actual parameter list. Each actual parameter is an expression which defines the initial value of one of the fields in the new record.

The correspondence between the expressions and the fields is given by matching the sequence of the expressions to the sequence of declaration of the fields in the record class declaration. The length of the two sequences must obviously be the same. (Thus, in effect, all fields are identified with parameters called by value; and the examples in section 8 can be much simplified).

This leaves the problem of specifying the subscript bounds and initial values of array fields. This problem may be solved by introducing the concept of an array constructor, which can feature in a record constructor as an actual parameter corresponding to an array field. An array constructor consists of:

1. an expression defining the lower subscript bound.
2. an expression defining the upper subscript bound.
3. an identifier which acts as a formal counting variable taking values between the lower and upper subscript bounds.
4. an expression (usually containing, and dependent on, the formal counting variable), which for any value in the range of the counting variable, yields an itial value for the array element which has that value as its subscript.

The notation of an array constructor might be:

(for <formal counting variable> := <lower subscript bound>
: <upper subscript bound> take <expression>)

Examples

```
(for i := 1:10 take 0)                                                    (1)
(for j := 1:m+1 take (i+j)/2)                                             (2)
(for i := 1:n take (for j := 1:m+1 take (i+j)/2))                         (3)
(for i := 1:n take (for j := 1:i take if i=j then 1 else 0))              (4)
(for i := 1:6 take case i of (true, false, true, AvB, false, true))       (5)
```

In example (1) the array field has subscript bounds from 1 to 10, and all elements are initially set to zero.

In example (2), if A is the array field, then A[j] takes as value the result of evaluating (i+j)/2 for j = 1, 2,... m + 1.

In example (3) a two-dimensional array is defined, in which A[i,j] takes the initial value (i+j)/2 for I=1, 2, ... n and for j=1, 2, ... m+1.

Example (4) illustrates the possibility of defining non-rectangular arrays as array fields. It defines a lower triangular array, with ones on the major diagonal and zeroes elsewhere.

Example (5) whows how an array can be defined by simple enumeration of the values of its elements.

7.5. Ambiguous References

The record union declaration can be abandoned in favour of a facility for permitting a reference variable or field to be declared without specifying at all which record class it is going to refer to. Such references are known as universal; universal references may feature as parameters and in assignments, but not normally in field designators.

A technique for determining the class of record actually referenced by universal reference variables is proposed in 7.5. Essentially the same technique is used in the SIMULA "connection block"; it is the only safe and elegant way of using universal or ambiguous references in field designators.

7.7. Input and Output of Records

In order to justify the claim made in 4.1, that record handling would be able to deal with conventional data processing applications, it is essential to introduce some form of output and reinput of records on a serial input/output medium (for example, magnetic tape).

A convenient way of doing this is to introduce two new statement forms, the output statement and the input statement.

The output statement selects a channel number, and then gives a list of record constructors, which construct new records on the output medium, and specify the (initial) values of their fields. One restriction would have to be imposed: that the values of any reference fields in the records to be output must be null. This precludes the possibility that an address which has been output from the store should become invalid because the record which it originally pointed to has been moved or deleted.

The input statement also selects a channel number, and then gives a list of reference variables. This statement causes the appropriate number of records to be input from the medium, and references to them are assigned to the variables in sequence. The class of each record must, of course, be appropriate to the class of the variable, and an implementation would be recommended to insert a check to verify this.

This proposal for input/output corresponds to the PL/I option for buffered input/output. It enables output records to be constructed, and input records to be processed, actually whithin buffer areas associated with the appropriate channel. The advantages of such a scheme of buffering are:

(1) "logical records can be grouped together into physical records of a size contributing to efficient use of the medium.

(2) multiple buffering techniques can be used to minimise delays associated with peropheral transfers.

(3) these advantages can be obtained without unnecessary copying of information in and out of the buffers.

48 Sebright Road,
Barnet, Herts.

March 1966.