AB21.3.6 RECORD HANDLING

C. A. R. Hoare

Entia non sunt multiplicanda praeter necessitatem -

William of Occam.

1. Introduction

A method is proposed for the representation in a computer of complex structured objects, and for their manipulation by a program written in a general purpose language, which is here assumed to be an extension of ALGOL 60. The properties of such objects can be represented by groups of the familiar Boolean and arithmetic quantities. Furthermore, arbitrarily complex networks of relationships holding between the objects can be represented and manipulated by the program.

The main recommendation for the proposal is that it extends the scope of a general purpose programming language to many fields of application which have hitherto been regarded as the preserve of special purpose languages. This suggests that the proposal is no arbitrary extension to an existing language, but represents a genuine abstraction of some feature which is fundamental to the art or science of computation.

A further recommendation may be found in the fruitful way in which the basic nucleus of essential facilities suggests a number of elegant and useful extensions, some of which are summarised in section 7.

2. Summary

2.1 Records and Record Classes

The proposal envisages the existence inside the computer during the execution of the program, of an arbitrary number of records, each of which represents some object which is of past, present or future interest to the programmer. The program keeps dynamic control of the number of records in existence, and can create new records or destroy existing ones in accordance with the requirements of the task in hand.

Each record in the computer must belong to one of a limited number of disjoint record classes; the programmer may declare as many record classes as he requires, and he associates with each

class an identifier to name it. A record class name may be thought of as a common generic term like "cow", "table" "house", and the records which belong to these classes represent the individual cows, tables and houses with which a particular activation of a program is concerned. It is obvious that no object can be both a cow and a table, so it is quite natural to insist that every record shall belong to one and only one class.

2.2 Fields

Each of the objects represented by records in the computer is likely to possess a number of properties of which the programmer would like to keep an account. For each object, the answers to questions about its properties can be recorded as real numbers, integers or logical values. For example, the height of a table can be represented as a real number, the number of rooms in a house as an integer, or the lactation status of a cow as a logical value. Any question which might be asked about a property can be represented by a field, and the answer to the question is the value of the field. Each field of a record acts as a special case of a variable; it may be assigned a value of the appropriate type, and this value may be used subsequently in the calculation.

The sort of question which it is meaningful to ask of an object, and the type of the numeric or Boolean answer which is given, depends on the class to which that object belongs; and it will usually be absurd to ask the wrong questions about an object of a given class. Thus it is meaningless to ask the number of rooms in a table, or the lactation status of a house. It is natural, therefore, to associate declarations of the fields, which are relevant to a class of record, with the declaration of the record class itself. Each field is declared inside the record class declaration in exactly the same way as a variable is declared in a block head: its type is specified, and an identifier given to name it. The effect of the declaration, however, is not to cause an allocation of storage once for all, as on block entry; rather, the storage to hold the value of the field is allocated within every record of that class when that record is created in the computer.

Note that the sequence in which the fields are declared is not significant, and that exactly the same effect is achieved by declaring them in a different sequence. The situation is the same as with variables declared in the normal way in a block head.

2.3 References, and Functional Relationships.

In addition to the properties of objects, the programmer may be interested in certain relationships which hold between objects of the

same or different classes. Thus he may be interested in knowing that some of the cows are mothers or sisters of other cows, or that some of the tables are situated inside some of the houses. Provided that the relationships are functional relationships (partial or total, many-one or one-one), it is meaningful to enquire of any given object what other unique object (if any) stands to it in a given relationship; for example, which house contains a given table, or which cow gave birth to a given cow. In this proposal we concentrate exclusively on functional relationships, since these have certain desirable properties for the programmer, and have a very convenient and efficient method of implementation in a computer. Non-functional relationships are fairly easy to construct in a number of alternative ways, as series or lists of functional relationships.

In order to represent relationships as well as properties, an entirely new type of value is introduced into the language; it is called a reference. We also require a new type of field which will take references as values. The value of a reference field refers from the record in which it occurs, usually to some other record; and this is used to represent the required relationship between the objects represented by the records. For example the question "who is the mother of" is answered by asking the value of the reference field "mother" which is allocated for this purpose in the declaration of the record class for cows.

In real life, most relationships are confined to holding between members of given classes; for example a house cannot be the mother of a cow, nor can a cow contain a table. Thus it is natural to associate with a reference field the name of the class to which all records referred to by it will belong. This is done on the occasion of declaration of the reference field itself.

2.4 Declared Reference Variables.

The most important feature about records is that they are created dynamically by statements in the program, and they are in this way quite different from other quantities which come into existence on passing through the declarations of a block head. For these other quantities, the method of referring to them inside the block is simply to use the identifier which has been associated with the quantity in the declaration.

However this technique is inapplicable to dynamically created records, so that some other method must be found. In fact, all that is required is to declare a variable of type reference, not inside a record, but actually a block head, in the same way as other declared quantities. Thus, apart from representing relationships between records, a reference may be used to point from one of the declared variables of some activated

block to some existing record, which is thereby signled out as being of current interest to the program. Once a declared reference variable gives access to a single record, the use of reference fields within that record can give access to further records when required. For example, suppose B is a declared reference variable pointing to a record of class "person" which has a reference field "farther"; then another record can be referred to by the so-called reference-valued field designator, "father(B)" or even "father(father(B))".

Such records are said to be indirectly accessible as opposed to the original record which is directly accessible. Records which cannot be accessed either directly or indirectly through a chain of some length are called inaccessible.

The declared reference variable also provides the means of accessing directly or indirectly other fields of the record besides reference fields, for example

age(B)

age(father(B))

which(given suitable declarations) would be integer-valued field designators with an obvious interpretation.

2.5 Assignment.

All kinds of field designators (including reference field designators) can appear, in the same way as other variables, on either side of an assignment statement.

age(B):= 34

age(father(B)):= age(B) + 30

B:= father (C)

father(B):= youngest son (eldest brother(C))

Of course, when a reference variable appears on the left hand side, the value of the expression on the right must be a reference, and a reference to the same class of record to which the left hand side is permitted to point.

When the value of a reference variable is changed by assignment, the record to which it previously referred is in no way affected;

the record remains in existence and the values of its fields are unchanged. Even when the reference variable goes out of existence as a result of block exit, the record to which it refers is unaffected (although it may become inaccessible).

2.6 Other operations defined for references.

In order to bring records into existence in the first place, the record class identifier should be used as if it were a function designator without parameters. The value which it delivers is a reference to a completely new record of that class. The values of the fields of the record will, naturally be undefined until assignments have been made to them in the normal way.

In order to give the programmer some chance to save storage by getting rid of records in which he is no longer interested, a standard procedure "destroy" is proposed, which takes as parameter a reference to a record, and which reverses the effect of having created that record.

In order to enable references to represent partial functional relationships, i.e. ones which do not necessarily yield a value, a special reference value null is introduced. This value fails to refer to a record, and any attempt to use it to refer to a record leads to an undefined result.

The Boolean operation of equality is defined for references. Two references are equal if and only if they are both null, or both refer to the same records.

Note that they are not necessarily equal merely because the values of all the fields of the records they point to are equal.

3. Example - Family Relations.

We take as an example the representation in a computer of a number of persons connected by family relationships. We define a record class "person" with fields representing the properties which interest us, for example, date of birth, sex, etc. In addition we have reference fields to point to a person's father and mother. Now it may be that on occasion the program wishes to scan through all the children of a given person, so that it is also necessary to introduce additional references to a person's youngest offspring, and to his elder sibling, who is defined as his father's previous child. A suitable record class declaration would be:

```
record class person;

        begin integer date of birth;
```

```
Boolean male;

reference father, mother,

youngest offspring, elder sibling (person)

end;
```

A declaration of reference variables providing access to records of this class would be:

```
reference Jack, Jill (person);
```

First we quote a section of program representing the birth of a son (John) to the man pointed to by the reference variable "Jack" and the woman known as "Jill".

```
begin reference John (person);

    John:=person; comment this creates a record to represent the son;

    date of birth(John):=today;

    male(John):=true;

    father(John):=Jack;

    mother(John):=Jill;

    youngest offspring(John):=null;

    elder sibling(John):=youngest offspring(Jack);

    youngest offspring(Jack):=John

end
```

Next we quote a portion of program which will determine whether Joe is a paternal uncle (or step-uncle) of Jack:

```
    uncle:=false

    if male(Joe) then goto complete;

    if Joe=father(Jack) then goto complete;
```

```
begin reference search(person);
        search:=youngest offspring(father(father(Jack)));
try again: if search=null then goto complete;
    if search=Joe then begin uncle:=true;
                        goto complete
                    end;
    search:=elder sibling(search);
    goto try again
end;
complete:
```

4. Scope.

4.1 Conventional Data Processing.

The most obvious application of record handling is to problems in data processing, where a record can be used to represent the pay information for an employee, an item on an invoice, a catalogue entry in a stock ledger, etc.

The technique of references enables records to be grouped together in files with either serial or random access. A serial access file is a group of records, each of which contains a reference field pointing to its successor; the relation of pointing must, of course, be a well ordering relation. When processing a serial file, a reference variable will be allocated to point to the current record. If B is the pointer, the statement

```
B:=successor(B);
```

accomplishes the "input" of the next record; "output" requires a more complex sequence of statements:

```
C:=B;
B:=payrecord;
successor(C):=B;
```

Random access files with access by given key can be readily programmed with the assistance of references, using standard techniques. References also provide natural and efficient techniques for random access of part and subassembly records in a parts explosion algorithm for production control; and their application to Critical Path Analysis is fairly obvious.

4.2 Simulation Studies.

In the simulation of complex situations in the real world, it is necessary to construct in the computer analogues of the objects of the real world, so that procedures representing types of even may operate upon them in a realistic fashion. For example, tankers belonging to an oil company can be represented by records whose fields give the relevant operational characteristics of the ships; similarly, the ports of arrival, and departure can be represented by records of a different class. The ship records will probably contain references to the current port of destination, the port of departure, etc.

In a customer servicing application, the serving stations and the potential customers will be represented by records; each customer record will contain a reference field which will point, when relevant, to the next customer in a queue. Each serving station will have a reference field pointing to the current first member of the queue at that station. If a customer can belong to several "queues" simultaneously, several reference fields may be allocated to point to queue successors: in this way, it is possible to represent finite sets of customers, as if each set is a serving station servicing a queue.

4.3 Information Retrieval.

References can assist in the representation of complex tables of information which are built up and referred to in the application of heuristic techniques; for example, in information retrieval the construction of a thesaurus of concepts with a complex networks of cross-references. Each entry in the thesaurus would be represented by a record, giving its title, certain weighting factors, and references to more embracing concepts, synonyms, antonyms, subconcepts, etc.

4.4 Design Automation.

In a finite geometry, a record class can be assigned to each of the major kinds of objects, for example, point, line, polygon, etc. The existence of references in these records would make it possible to access all lines passing through a given point, all points incident to a given line, all lines enclosing a given polygon, and (if necessary) all polygons abutting

to a given line. The usefulness of highly dense inter-connections will depend on the sort of processing which has to be done, and is entirely at the discretion of the programmer. For example, in some applications it is essential that, for every record pointed to, one should be able to gain access to all records pointing to it, so that a backward pointing chain must be constructed. In other applications this is quite unnecessary, and its omission will save both space and time;

4.5 Compiler Writing.

In the construction of compilers for a complex language, records can be used to hold the salient characteristics of variables, arrays, blocks, procedures, labels, for statements, etc. As well as containing such information as type and address, each record representing a variable could refer to the block or procedure to which it is local, the for statement for which it is currently a counting variable, etc.

4.6 Manipulation of Symbolic Expressions.

In representing in a computer the nested structure of an arithmetic expression, a record may be used to correspond to each operator/operand combination. One of the fields would be allocated to indicate the nature of the operation, for example, addition, subtraction multiplication, or division. Two reference fields would contain pointers to the left and right operands of the combination.

If the operand so referenced is not itself a combination, the operator will contain a special value indicating that the record represents a variable; in this case the two operand references are meaningless and a third reference which exists in every record will be found to point to a record of a different class containing the relevant details about the variable.

This rather cumbersome method of dealing with the ultimate components of an arithmetic expression is due to the rule that each reference is restricted to pointing to records within a single specified class, so that the operand references which are specified as pointing to other operator/operand combinations cannot be used to refer to records of a different class representing variables. A more elegant solution to this problem is proposed in section 7.5 (Ambiguous references).

5. Derivation.

5.1 Business Oriented Languages.

The concept of a record consisting of a number of named fields is

taken from the common practice of Business Oriented Languages.

5.2 List Processing Languages.

A list may be regarded as a group of records, (words), some of which contain references (addresses of other words), and some of which contain actual information. However, most list processing languages impose restrictions on the layout and content of the record, and do not use programmer-chosen names to refer to the constituent parts of a list structure.

5.3 EULER (N. Wirth and H. Weber).

The concept of reference which is used in this proposal is similar to the ideas of reference which have appeared in languages such as CPL and EULER. However, the motivation for introducing references to represent functional relationships appears to be quite different from the motives of the authors of EULER or CPL, who were probably interested in providing some analogue of the FORTRAN parameter mechanism. The current proposal avoids or evades some of the difficulties associated with EULER references.

1. references cannot be used to point to the declared variables of a block, which go out of existence surreptitiously on block exit. They can only point to records, which remain in existence for as long as any reference points to them.

2. references are restricted to pointing to records of a given class, so that the compiler can keep an account of the types of all components of expressions, and the running program can keep account of the whereabouts of every reference.

5.4 Proposal for New Types (Prof. J. McCarthy).

The current proposal represents part of the cartesian suggestion made by Prof. J. McCarthy as a means of introducing new types of quantity into a language.

The main restrictions of the current proposal are that records cannot be declared as part of the local workspace of a block, nor as part of a record itself.

The reference facility described here has one important characteristic not possessed by the new type proposal, that it is possible to have several references pointing to the same record, so that many-one relationships can

be represented as readily as one-one relations.

5.5 AED-O (D. T. Rose)

The records of the current proposal correspond to "beads" of AED-O; a "plex" is the AED-) term for a group of records mutually interconnected by references. The idea that the reference structure of the data should be placed entirely under the control of the programmer is a most important contribution of AED-O, and is taken over in this proposal.

The main difference between the current proposal and AED-O is that the current proposal ensures that the implementation knows exactly where every reference exists in the store, and what it points to, so that far more comprehensive techniques are available for type checking at compile time and for garbage collection at run time.

5.6 Comparison with a Proposal on Trees.

The main advantages of the current proposal over a scheme presented and discussed at the Princetown meeting of the ALGOL Committee appear to be as follows

(1) The most important feature of the current proposal is that it permits the use of meaningful identifiers rather than numeric indices to identify the individual components of a structured record.

(2) It allows each record to contain information of many different types, and yet it permits the type of each component of an arithmetic expression to be determined purely by a lexicographic scan.

(3) The concept of a reference provides a quite general and acceptable way of representing arbitrarily complex relationships between objects, which would be rather difficult in trees without "sharing".

(4) The size of each record remains constant between the time of its creation and its ultimate demise, so that more efficient and economic techniques of storage allocation can be adopted.

(5) The programmer is encouraged to differentiate between records to which he is at any given time making frequent direct access, and those which he is not currently interested in, and can only refer to indirectly. This improves the prospects of using two-level storage in a convenient and elegant fashion.

6. Implementation.

There are many possible methods of implementation and it is not intended to favour one of them against another; in order to assist a concrete visualisation, the following description is given.

6.1 Records, Fields and References.

Each record is represented by a group of consecutive locations in store, which will hold the values of all fields associated with the record in the record class declaration.

Each field of a record is represented by that part of the total storage of the record which has been allocated to hold its value; the amount of store will often depend on the type of the attribute. e.g. boolean, integer, real or reference.

Each reference is represented by placing the address of the first word of the designated record in the space allocated for the reference variable.

6.2 Store Administration.

The implementation of the proposed record handling facilities requires a rather more sophisticated method of storage administration than ALGOL 60, since allocation and reallocation of storage can occur at arbitrary times; it is not confined to the beginning and end of blocks, and it is not nested in a way which permits simple stack administration.

In practice, however, many implementations of ALGOL, particularly those which made use of random access backing store, already supplement a stack administration of core by a fully randomised one for program chapters and even arrays.

6.3 Page Control.

A simple method of implementing record allocation and deallocation is to use the other end of store which is not being used by the stack. This store should be split into pages of (say) 256 words; each page is reserved to hold records of a particular size. If more records of a particular size are created than will fit into a page, a new page is allocated, and this is linked into a chain of pages holding records of that given size.

As records are deleted, holes will appear in the pages; these are chained together into a free store list for each record size. Whenever a new record of that size is requested, a hold record is always presented in preference to a non-hole one.

In general, the number of different sized records used in a program will be quite small; in fact, it cannot exceed the number of record classes. In practice, also, the size of larger records will be "padded out" so that an exact number of records fit into each page. This limits the number of different record sizes to 31, viz.

1 to 19, 21, 23, 25, 28, 32, 36, 42, 51, 64, 85, 128, 256.

Techniques for dealing with records longer than 256 words are fairly simple, but will probably not be required.

6.4 Store Collapsing.

If the dynamic stack and the record storage ever meet in the middle, this means that the program is making demands near the limit of available core, and simple implementations may stop at this point. However, a series of more advanced techniques, are available to retrieve the situation if required.

If often happens that even though the two stacks have met, there are still a large number of holes in the record area, which added together would provide sufficient space to continue the program. In this case, the solution is to attempt to release several whole pages, by copying the records out of them into holes occurring on other pages of the same chain. If the page made free is not contiguous with the stack the most recently allocated page can be copies bodily into its place.

The most complex part of this reshuffle is that all the references in the declared data or in the records themselves must be adjusted to point to the new positions of the records rather than the old.

This is only made possible because the compiler knows the positions of every reference used, and can pass the information on to the store collapse routine.

6.5 Automatic Deletion.

The techniques for referencing records ensure that no record ever goes out of existence without the explicit intervention of the programmer, which he may be unable or unwilling to give.

This situation is partly remedied by the fact that when a record class is declared in an inner block all records of that class go out of existence when that block is exited. However, it is quite possible that the programmer fails to make use of this technique, so that his store becomes choked with records

which in fact he does not ever want to refer to again.

If this is the case, one of the surest mechanically recognisable criteria for the situation is that a record has no declared reference pointing to it, either directly or indirectly through reference fields in other records. Such records can be safely deleted automatically, since there is no conceivable way of which the programmer could ever subsequently access them. This idea is taken from J.McCarthy's LISP garbage collection.

6.6 Random Access Backing Store.

If a high volume random access backing store is available to hold records, it should be split into pages in exactly the same way as was described for main store, except that the page size will probably be dictated by hardware considerations. All references in the records themselves will now, of course, refer to backing store addresses rather than main store.

In the main store will be held a certain amount of book-keeping information about the chaining of the pages, etc; and further, there will be at least one page-worth of core storage allocated for each reference declared in the head of the program blocks.

This means that all direct references are carried out with the efficiency of mainstore indirect addressing, and only indirect references require access to backing store. Of course, all indirect references will be backing store addresses; and it is therefore essential that they should be redirected to the main store copy of the relevant page if it happens to be in main store. For efficiency, a table of all such pages must reside in main store, except on machines with special hardware associative page address memories. On such computers, even indirect references can be made with no further difficulty if the page is in store.

If additional pages of core are available, they can be used as buffers to hold information which is undergoing transfer to or from backing store, or to hold pages which have recently been referred to and are therefore likely to be soon referred to again. Various strategies for selecting which pages to keep have been proposed, but it is probably not worth while to get too sophisticated, since the programmer who is interested in efficiency can take care to declare enough reference variables so that the frequency with which he makes indirect references is reduced to a minimum.

6.7 Collapsing and Deletion.

The same techniques for store collapsing and automatic deletion as are described for core storage can be applied to random access backing store. The amount of time to carry out the process will be very much greater, but it is to be hoped that the need for it will be very much less

frequent.

6.8 Homogeneous store control.

If a mechanism for fully dynamic storage allocation is set up, it becomes tempting to use it for allocation of storage to variables declared in block heads. The mechanism of references can then be used to deal with the necessary pointers which have to be set up to trace through the lexicographically enclosing blocks and the dynamically calling blocks, and to refer to the code representing the body of the block. The value of the sequence control is, of course, a local variable of the block concerned.

There are three advantages in using the fully dynamic method for local variables as well as records: firstly, it avoids the requirement of contiguity of large sections of store, which tends to interfere with the working of the randomised storage administration; secondly, it enables the local variables of blocks not currently accessible to be dumped to backing store if they are not required for long periods. Thirdly, it is more elegant to use only a single storage control method for all purposes.

The full use of non-nested storage allocation for local quantities opens up the prospect of permitting several procedures to be active simultaneously. This possibility of asynchronous working can sometimes be used to minimise the effect of delays due to the slow access time of random access backing store.

7. Possible Further Extensions.

7.1 Procedures and Parameters.

The most obvious omission is the failure to relate references, records and fields to the procedure concept of ALGOL. The relationship is in fact fairly obvious; field identifiers, and reference identifiers should be permitted to appear as actual parameters called by name, and field designators as actual parameters called by name or by value.

Furthermore, it should be possible to declare reference valued procedures and use them as function designators in a reference-valued expression.

7.2 Procedure Fields.

In many cases, certain characteristics of the object represented by a record can be computed as a function of other characteristics. In such cases it is sometimes useful to avoid allocating a field to hold the value of the

characteristic, and instead to compute the value whenever it is required. A declaration of the type procedure to compute the value could appear among the other field declarations. If this procedure were parameterless, its use as a function designator would have the same appearance as a field designator.

For example, the length of a line can be defined in terms of the co-ordinates of its end points, or the grandfather of a person can be defined as the father of his father.

7.3 Array Fields.

In addition to permitting procedure declarations to be associated with a record class, it is clearly suggested that array declarations be given the same privilege. Certainly there are cases where it is useful to permit an array or arrays to be regarded as part of a record. In particular, an array of references is a good way of representing a relationship which is not functional.

The incorporation of arrays in records gives the language designer the problem of choosing at what time the subscript bounds of the array are fixed. There seem to be three possibilities; the most efficient is that the bounds are calculated on entry to the block in which the record class is declared, so that all records of the class will be the same size as all co-existent records (though they may be different on different activations of the block). The least restrictive is to permit the size of the array in each record to depend on the actual assignments made to the subscripted field designators. The second proposal is similar to the treatment of arrays in EULER.

The third intermediate proposal, combines some of the efficiency of the first and some of the flexibility of the second; it allows the size of the array fields to be determined at time of creation of a record, but insists that the size of each individual record remains constant throughout its life.

In detail, the third proposal requires that formal parameters be supplied in a record class declaration; these would feature in the subscript bound positions of the array field declarations. When the new record is created, by mentioning the record class name, an actual parameter part is added; and the parameters supply the values which will be used in the calculation of the actual subscript bounds.

7.4 Initial Values.

If a record class is to have a parameter mechanism, it is tempting

to use it to enable initial values of some or all of the fields to be specified at the time of creation of the record. This can be done be associating with the record class declaration a statement which will compute and assign initial values to the fields, taking both parameters and global quantities into account.

If the proposals 7.2, 7.3, and 7.4 are accepted, we reach a situation in which a record class declaration has exactly the same form and effect as a (function) procedure declaration, except that it delivers as its value not a simple quantity, but a reference to the record consisting of the values of all its local quantities at the time of exit from the procedure. Thus we have worked our way through to a startling simplification, which enables record class declarations to be treated largely by the same mechanism as procedures. At run time the mechanism is also very similar.

The main differences are:

1. at run time the space allocated for local quantities is not freed on exit; rather, the "display" value for them is passed on as a reference to the calling program.

2. at compile time, a record of the identifiers and types of the local quantities are preserved throughout the translation of the remainder of the block.

7.5 Ambiguous References.

We have seen from the example of 4.6 that the rule which restricts the range of a reference variable to a single record class may lead to difficulties and unnatural expressions in certain applications. This restriction can be relaxed by permitting the programmer to associate with a reference variable declaration not a single record class name, but a whole list of record class names; and the reference is permitted to refer to records from any of these classes. Such a reference is called ambiguous, and the list of classes is called a union. It is recommended that a union be declared separately from the references, and be given a separate name thus:

union thing (cow, house, table);

reference B (thing)

The remaining problem is to ensure that the translator can still establish the types of all the components of an arithmetic expression, and

that the run-time administration can keep an account of the whereabouts of every address. The first problem can be solved by an adaptation of the technique of the case expression and case statement.

Thus, if B is a reference variable which may point to a cow, a house, or a table, it will generally be used in a construction such as:

if B is a house then .... height(B) ...

or if it is a cow then ... height(B) ...

or if it is a table then .. height(B) ... ;

The translator now knows that in the text between the first then and the first or if it is a the reference B may be taken to designate a record of class "house"; and similarly for the "cow" and "table" terms. This means that the translator can even cope with a situation when each class has a field "height", and in one class it is an integer and in the other a real number, and in the third, perhaps a reference!

A reference which is ambiguous can be represented in the computer at run time by two values; one of them is a class indicator, which is a small integer indicating which of the possible record classes the reference is actually pointing to, and the other is the reference itself. The extended case expression uses the class indicator to compute the destination of a switch jump; and the store administration uses it to keep an account of the reference structure of store.

The availability of ambiguous references fulfils a similar function to McCarthy's proposed class union declaration from which the idea has been taken;

7.6 Finite Set Declarations.

In programming the sort of problem for which records are used, it is frequently necessary to represent as a small integer a value from some fairly small finite set; for example, in a representation of playing cards, the suit of the card might be represented by an integer between one and four, using the following convention:

1 = a Club

2 = a Diamond

3 = a Heart

4 = a Spade

In ALGOL 60, and many other programming languages at present, this sort of correspondence table has to be constantly borne in mind by the programmer or the reader of the program; and every time a test of the suit is made it must be expressed numerically in a manner which is as unnatural as the use of integer addresses in primitive machine coding.

What is required is a declaration which associated a named set of values with a finite set name, and then permits the programmer to assign and test the values using their mnemonic names. Examples of finite set declarations might be

set sex(male,female,unknown);

set suit(clubs, diamonds, hearts,spades);

set alpha(A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z);

Variables which take values from the finite set can be declared thus:

member cardA, cardB(suit),firstchar,lastchar(alpha);

The same technique of case expressions and statements as is proposed for ambiguous references can also be used to discriminate between values of a member variable, thus:

if cardB is a diamond then .....

or if it is a club or a heart then ..... ;

In the above statement, the programmer obviously believes it can't be a spade; and if it turns out to be, the result is underfined.

8. Example of Extended Facilites.

The first example of extended facilities is a recasting of the example of section 3. The declaration for the record class can now become:

record class person(m,f,mo);

Boolean m; reference f,mo(person);

begin integer date of birth;

```
        Boolean male;
        reference father, mother,
          youngest offspring, elder sibling (person);
          date of birth:= today;
          male:= m;
          father:= f;
          mother:= mo;
          elder sibling:= youngest offspring(father);
          youngest offspring:= null
    end
```

Now the section of code representing the birth of a son can be written as a single statement:

```
youngest offspring(Jack):=person(true, Jack, Jill);
```

A shorter version of this can be obtained by extending the use of value parameters, thus:

```
record class person(male, father, mother);
    value male, father, mother; reference father, mother(person);
    Boolean male;
    begin reference elder sibling, youngest offspring (person);
        youngest offspring:= null;
        elder sibling:= youngest offspring(father)
    end
```

A more significant example is taken from the field of symbol manipulation, - the representation and processing of arithmetic expressions. For the purposes of the exercise, an expression is either a pair of

operands together with an operation, a function designator of a function with a single parameter, a variable or a constant. The following declarations suffice:

```
set operation(sum, difference, product, quotient);

union expression(pair, function designator, variable, constant);

record class pair (op, left operand, right operand);

        value op, left operand, right operant;

        member op(operation); reference

                left operand, right operand(expression);

        comment this is a dummy statement;

                ;

        record class function designator(function part, parameter);

                value function part, parameter;

                reference function part(function), parameter (expression);

                ;

        record class function(bound variable, body);

                value bound variable, body;

                reference bound variable(variable), body(expression)

                ;

        record class variable(name);

                value name;

                string name;

                ;

        record class constant(value);
```

```
value value;

real value;

;
```

This is quite a long list of declarations, especially in comparison with most list processing languages which would not require data declarations at all. However the declarations in fact represent exactly the explanatory material which must accompany a list processing algorithm to enable the reader to understand what it is doing. There is little hardship and much benefit in expressing this information in a standard way, so that it can be understood by a programming system.

As an example of an expression which can be represented as a structure in accordance with the declarations, take the expression:

$$\frac{p-q}{p+q}$$

where $p = g(x)$

$q = g(x^2)$

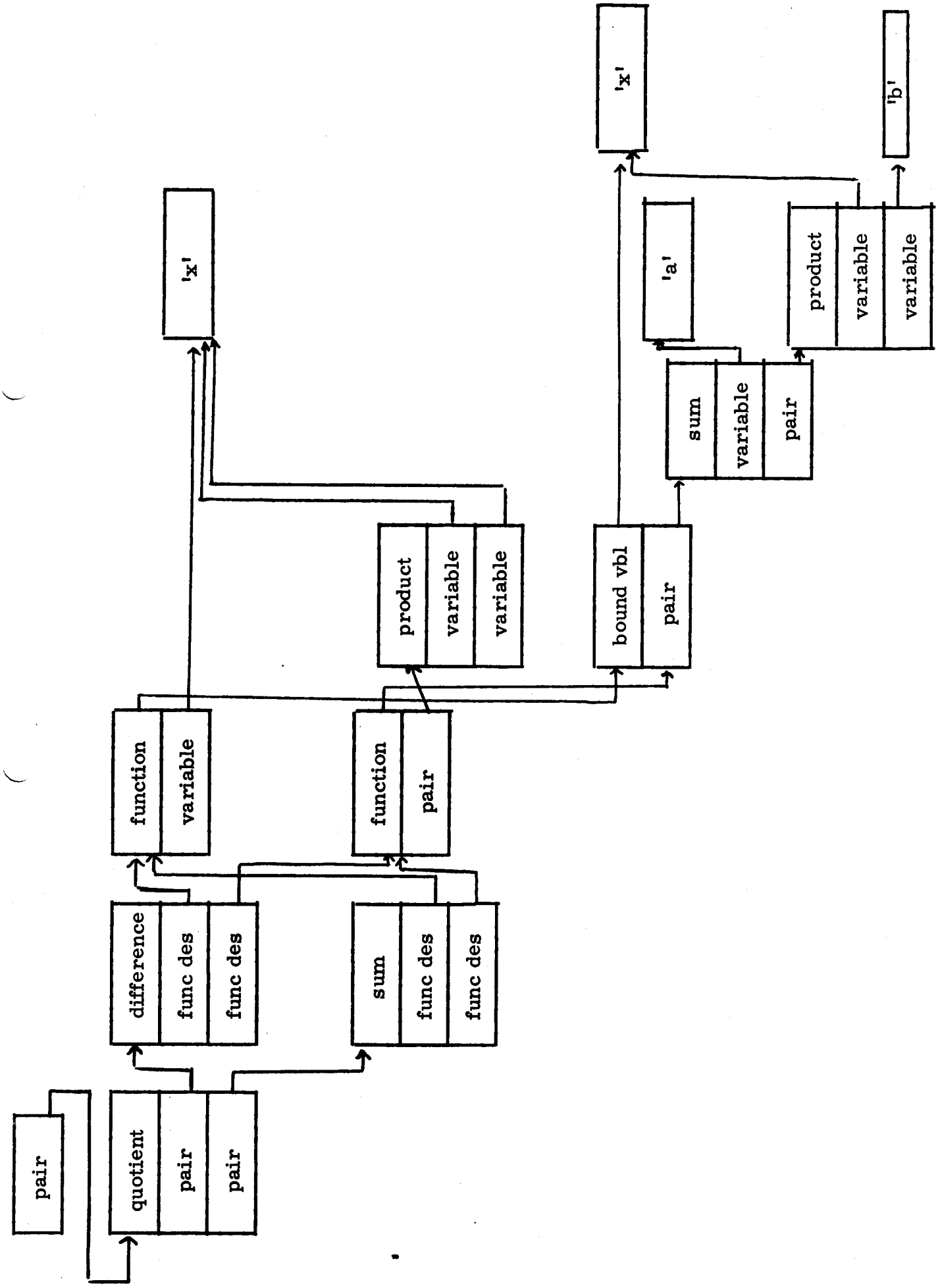
and $g = x.\ a+bx$

This can be pictured in the computer by thinking of references as arrows, and records as a box with compartments containing (in the case of a reference) an indication of the class of record to which the reference points, or (for a non-reference field) its value.

There are two features worthy of note; firstly that common sub-expressions (p and q) are dealt with in a manner most economic of storage, and without the introduction of auxiliary variables; and secondly that the technique of dealing with bound variables avoids all problems of collision, since they are clearly shown as different variables, even if they happen to have the same name.

As an example of a procedure operating on expressions of this form, we take the elementary, but commonly quoted, case of symbolic differentiation.

```
reference (expression) procedure derivative(e) with respect to: (x);
```


pair
quotient
pair
pair
difference
func des
func des
sum
func des
func des
function
variable
function
pair
'x'
product
variable
variable
bound vbl
pair
'x'
'a'
sum
variable
pair
'b'
product
variable
variable

```
        reference e(expression), x(variable);
        begin
if e is a variable then derivative:= if e=x then
                constant(1)
                else constant(0)
or if it is a constant then derivative:= constant(0)
or if it is a function designator then
begin reference f, fdash(function), y(variable), g(expression);
        g:=parameter(e);
        f :=function part(e);
        y:=bound variable(f);
        fdash:=function(y, derivative(body(f), y));
        derivative:=
        pair(product, function designator(fdash, g), derivative(g, x)
end
or if it is a pair then
        begin reference u, v, udash, vdash(expression);
                u:=left operand(e);
                v:=right operand(e);
                udash:= derivative(u, x);
                vdash:= derivative(v, x);
derivative:=if op(e) is a sum then
                                pair(sum, udash, vdash)
```

```
        or if it is a difference then

                        pair(difference, udash, vdash)

        or if it is a product then

pair(sum, pair(product, u, vdash),

                        pair(product, v, udash)

        or if it is a quotient then

                        pair(difference, pair(quotient, udash, v),

                                        pair(product, e,

                                                pair(quotient, vdash, v));

comment this way of expressing the derivative of a quotient is more likely
        to be computationally successful than that of the traditional formula;

end

end
```

9. Implications for ALGOL 6Y.

If ALGOL 6Y is to be a language in which it is possible to process the language text itself, it is essential that it should embrace some conceptual frame work for the construction, analysis, and processing of complex networks of elements. It is obvious that the only structuring feature of ALGOL (the array) is inappropriate for the purpose. It appears also that the generalisation of the array) is inappropriate for the purpose. It appears also that the generalisation of the array structure provided by EULER and elaborated by the Proposal on Trees is somehow inadequate. It is possible, however, that a development of the record handling techniques will be found to meet the requirements.

10. Formalities (for basic facility).

The following pages give the main additions and changes to the Revised Report on ALGOL 60, which are required to add a basic record handling facility to the language.

2.2.3 Reference Value.

<reference value> :: null

A reference value null fails to designate a record; if a reference variable has this value, the result of its use in a record designator is undefined.

2.3 Delimiters.

Add to the end of the definitions of declarator an additional term:

" reference record class".

2.7 Quantities, Kinds, and Scopes.

At end of first sentence, replace "and procedures" by ", procedures, references and records".

2.8 Values and Types.

At end of first paragraph, replace " or a label" by ", a label or a reference".

Add after third paragraph.

"The record class associated with a reference defines the class to which all records referenced will belong."

3.1.1 Syntax.

Replace the definition if simple variable by the following five definitions:

< simple variable >::= <variable identifier > |

< field designator> | <record designator>

< field identifier > ::= <identifier>

< field designator> ::= < field identifier >(<record designator>)

< reference identifier > ::= <identifier>

< record designator> ::= < reference identifier> | <record class identifier > |

< field reference identifier > (>record designator>)

<field reference identifier> ::= <identifier >

3.1.2 Examples.

Add the following:

record designator:

father (Jack)

field designators:

date of birth (elder sibling(Joan))

operator (left operand(expression))

3.1.3 Semantics.

Add at end of paragraph:

The class of a record referred to by a record designator is defined by the declaration of the corresponding reference identifier (of Section 55 Reference Declarations).

The type of the value of a field designator is defined by the declaration of the corresponding field identifier, which occurs in the relevant record class declaration (cf Section 5.6 Record Class Declarations). In the case of nested record and field designators, the class and type can be determined by recursive application of the above rules.

Add:

3.2.6 Record Destroying Procedure.

In addition to standard functions, it is recommended that certain standard procedures be defined. Among these, it is recommended that there should be one, namely:

destroy(V)

which removes from existence the record designated by the variable V.

This procedure has the effect of rendering undefined any program which attempts subsequently to refer to the destroyed record. It has no effect on any program which avoids such reference, other than possibly to expedite its successful execution on a computing machine.

5. Declarations.

Replace last part of this section

("No identifier may be ... <procedure declaration> ")

by:

No identifier may be declared more than once in any one block head, with the exception of a field identifier declared several times in disjoint record class declarations.

Syntax.

<declaration>::=<type declaration>|<array

declaration>|<procedure declaration>|

<reference declaration>|<record class

declaration>

Add after the end of 5.4.6:

5.5 Reference Declarations.

5.5.1 Syntax.

<reference segment>::=<reference identifier>

(<record class identifier>)

<reference identifier>, <reference segment>

<reference list>::=<reference segment>

<reference segment>, <reference list>

<reference declaration>::= reference <reference list>

5.5.2 Examples.

reference left operand, right operand

(combination), vbl(variable)

reference mother, father(person)

5.5.3 Semantics.

A reference declaration serves to introduce one or several identifiers to represent variables which will take values of type reference. All the variables declared in a reference segment are associated with the record class designated by the identifier in parenthesis, and the range of reference values which they are permitted to take is restricted to refer to records of this class.

5.6 Record Class Declarations.

5.6.1 Syntax.

< record class declaration >::=

record class <record class identifier>;

begin <field declaration list> end

< field declaration list > ::= <field

declaration> |

<field declaration > , <field declaration list>

<field declaration >::= <type declaration>

<reference declaration>

5.6.2 Examples.

```
record class customer;
        begin integer time of joining queue,
                requirement;
        reference next (customer), station for
                (station)
        end
```

```
record class line;
    begin real length;
        reference next parallel, previous
            parallel(line),
            starting point, finishing point
            (point), lower polygon, upper
            polygon (polygon)
    end
record class pair;
    begin integer operator;
        reference left operand, right operand
            (pair), vbl(variable)
    end
```

5.6.3 A record class declaration serves to define the structural properties of records belonging to the class. The principal constituent of a record class declaration is a sequence of declarations which define the fields of the record their types and names.

Every record belongs to one and only one class; with every reference variable there is associated the identifier of the record class to which any record designated by it will belong.

The constituent fields of a record are designated by field designators, consisting of the relevant field identifier followed in parentheses by a record designator. This record designator must either be, or begin with, a reference variable associated with the record class in the declaration for which the field was declared.

A record class identifier appearing in a reference-valued expression acts as a parameterless function designator, and delivers as its value a reference to a new record of that class.

11. Acknowledgment.

The author wishes to thank Mr.M.Woodger for stimulating many of the ideas expressed here, and for pointing out a number of defects in an earlier draft, some of which have been removed.

Elliott Bros. (London) Ltd.,
Boreham Wood,
Herts, England.

29th July 1965.