Areas and record-classes

H. D. Baecker

Department of Computing Science, University of Calgary, Calgary, Alberta, Canada T2N 1N4

To date list processing facilities have been imperfectly incorporated into general-purpose algorithmic languages. A brief survey of available forms of list processing is followed by an outline of required features of a list processing facility. The syntax and implementation of such a facility in an ALGOL like framework is then outlined.

(Received December 1972)

1. Introduction

Many problems in computation are partly or wholly soluble using the techniques of list-processing. Should a programmer wish to use these techniques then he has in general five options:

1. To construct his own list processing environment in machine code or within a language such as FORTRAN IV, or to use another's subroutine package if available
2. To use the random access file system of his installation to provide the list space and access routines

   (Note: the above two approaches can be combined profitably)
3. To use a special purpose list processing language such as LISP or IPL-V
4. To use the generalised facilities available in PL/I (*BASED* attribute) or ALGOL 68 (**heap** modes) or ALGOL-W or SIMULA 67
5. To use the more disciplined facilities available in PL/I (*BASED* variables within *AREA*s) or PASCAL (record-classes).

The remainder of this paper will be concerned with alternative 5, and with its relationships to alternative 4, but we shall first state the present author's prejudices and his reasons for not expanding on alternatives 1-3. Software people are, metaphorically, in the business of making two blades of grass grow where one grew before. We are toolmakers. In a world of limited resources the only justification for our resource-consuming activities is that we can ultimately help to ameliorate the human condition. In any case the constraints upon and goals of our activities are only minimally those of a natural science, they are largely those of the social sciences. There is now little problem in designing a computer, using it effectively thereafter is another matter. The niceties of syntax and automata theory are not the summit of computing, they are the underpinnings in the basement, just as James Clerk Maxwell's equations are to the BBC or CBS.

It thus behoves us to forge adequate tools for the problem-solver to apply computers to his needs effectively. Just as a saw is a compromise between the human handgrip and strength on the one hand, and the properties of the materials of which it is made on the other, so a programming language must be a compromise between the conceptual-linguistic habits of the user and the physical characteristics of the machine. This paper explores aspects of the tools needed to use list-processing techniques.

To superimpose a private list-processing package upon FORTRAN or APL or ALGOL 60 is an unacceptable solution on two grounds:

1. The resulting program statements obscure rather than clarify the user's intent, they do not describe the problem solution within the terms of the problem statement
2. The source language compiler does not provide error-detection aids for this class of operations.

Use of the random access file system (if, say, you have a sophisticated index sequential file facility) is appropriate if the list structures to be manipulated will exceed working storage, however even in such cases extensions to the area/record-class concept would allow more fluent problem statement. This will be discussed below.

The special purpose list processing languages are just that. The reality we seek to model in computer programs is not that homogeneous (nor as homogeneous as the simple data types of ALGOL 60). We need a more versatile tool for most real problems.

2. Survey

Both PL/1 (IBM, 1965), by means of the *BASED* attribute, and ALGOL 68 (van Wyngaarden *et al.*, 1969), by means of the **heap** values permit the declaration of complex data objects instances of which can be generated independently of the scope discipline that governs variables with the *AUTOMATIC* attribute or of *loc* values respectively. These complex data objects may include fields with the *POINTER* attribute or of **ref** mode, thus permitting these objects to be linked together to form lists, trees, etc.

These data objects are presumed to be generated in whatever is left of the user's working storage allotment after accommodating his program and his dynamic stack. Neither language includes facilities for declaring the amount of *BASED* (and *CONTROLLED*) storage or size of **heap** required. Neither does either language include library procedures that allow the user to interrogate the system to determine how much space he has left at his disposal. If he misjudges he gets bounced. The total program space is a function of the implementation, the installation, and of the job control language.

The ALGOL 68 implementor is free to provide a garbage collection routine in his run-time system so as to recover the space occupied by **heap** objects that have become inaccessible. He is also at liberty to implement the heap in more than one level of storage (Baecker, 1970), so obviating some of the difficulties raised in the previous paragraph and giving the user access to random access secondary storage.

The above freedoms do not apply to the PL/I implementor, garbage collection, or any system action to relocate *BASED* variables, is prohibited, as is the use of secondary storage except in a virtual memory system. The reasons for these restrictions will become apparent in our discussion of *AREA*s.

SIMULA 67 (Dahl *et al.*, 1970), ALGOL W (Wirth and Hoare, 1966; Bauer *et al.*, 1968), and PASCAL (Wirth, 1970) include **record-classes** as data types. However, since a **class** instance in SIMULA 67 is generated within an area of storage very much like the ALGOL 68 **heap** and also need not be purely a data structure we shall not discuss it further.

In ALGOL W a **record** is a structured value composed of fields the declaration of which serves as a template. When the declared record identifier is later used as a record designator a new instance of the record is constructed, and may have its fields initialised. The user may associate one or more **reference** variable with each **record** class, such variables may themselves be fields of records, so that list structures may be constructed.

Instances of records are generated in some undefined space and become inaccessible according to the normal scope rules. A garbage collector is assumed to exist.

PASCAL record variables correspond closely to the **record** of ALGOL W. In addition PASCAL provides the class variables. A **class** is declared to comprise a declared maximum number of instances of a given **record.** The **record** instances are generated by a system procedure alloc. Two or more class variables may be 'containers' for the same **record.** Thus in PASCAL a pointer variable is bound at declaration to a given **class**, not to a **record.** Class variables follow the normal scope rules, on exit from a scope a **class** disappears and on re-entry to that scope the **class** is there once more, empty. There is no garbage collection.

Before launching into a full-scale discussion of areas and record-classes the reader's attention should be drawn to the Virtual Core System provided in the BASIC-PLUS language under the RSTS operating system on PDP-11 computers (DEC 1972a, 1972b, 1972c) as well as to NOVA ALGOL (Data General Corporation, 1971).

3. Desiderata

It is possible to state several requirements for a useful list-processing facility within an algorithmic language:

1. The syntax rules should enable as much checking of the user's intent as possible to be done at compile time
2. Those users who wish to make small scale use of list processing facilities should not be penalised to provide for the large scale user, whilst yet the large scale user should be free to use the facilities fully and should bear the overhead costs he generates
3. It should be possible to reclaim the memory space allocated to a given list structure simply and cheaply when that structure is no longer needed without affecting other structures that may coexist in the program
4. If garbage collection is provided by the system then such an event should only be triggered by overflow of and should only affect the critical set of objects
5. The facility should be so designed as to aid the user in achieving reference locality within a virtual memory, thus avoiding page exceptions
6. It should be easy and efficient to file and retrieve a complete list structure, and to transfer it from program to program.

4. AREAs

A PL/I *AREA* may be of any storage class. The language does provide the facility to file a complete *BASED AREA* and then to retrieve it, perhaps in another program activation, with all internal cross-references in the *AREA* intact. This is achieved by having two types of reference variables in the language. A *POINTER* is a reference the value of which is the hardware location at which the referenced variable is stored. An *OFFSET* is a reference the value of which is the relative address of the referenced variable within the *AREA* to which the *OFFSET* variable was uniquely bound by declaration. An *OFFSET* bound to a certain *BASED AREA* may be declared as a field of a *BASED* structure that is subsequently allocated in an entirely different *AREA*. Thus cross-references between *AREA*s may be established. The *EMPTY* built-in function reinitialises an *AREA* to have no contents.

An injudicious sequence of *ALLOCATE* and *FREE* statements and/or of locator variable (*POINTER*, *OFFSET*) assignments may leave a block of allocated storage inaccessible. As no garbage collection is provided this space is lost to the user.

Similarly, careless use of *FREE* and/or *EMPTY* may leave locator variables referring to garbage.

PL/I in no way meets desideratum (1). Locator variables are not bound by declaration to any particular set of variables having some declared attribute. This complete lack of discipline makes both debugging and documentation too difficult. It is a sufficiently severe fault to make the facilities unacceptable by any programming standards worth having. Equally unacceptable is the run-time havoc resulting from being able to *FREE* or *EMPTY* whilst leaving locator variables referencing limbo.

Our other desiderata are met fairly well by the PL/I facilities, which makes it sad that there should be two such major deficiencies in the language that make it unacceptable.

5. The ALGOL 68 heap

The **heap** and **ref** values of ALGOL 68 are not open to the objection advanced against PL/I. However, ALGOL 68 facilities fall pretty flat when matched against desiderata 2 to 6. Clearly, there has to be some extension beyond the heap, it is too undifferentiated a morass to be really useful and efficient.

6. Record-classes

Classes, as previously discussed, meet the objections made against PL/I and also meet most of our other desiderata. PASCAL comes closest to our requirements, particularly now that a later version of the language incorporates a limited facility to free part of the space allocated in a **class.** However, PASCAL does not meet desideratum 6, at all, nor does it incorporate garbage collection.

At this point we may as well state the primary problem that arises if we wish to implement together areas or classes, garbage collection, and file facilities. If a **record** of a **class** can reference a **record** in another **class** then garbage collection must embrace all **classes** at a time as long as references are direct hardware addresses. Secondly, if the contents of a **class** could be on a file then references from that **class** to others cannot be taken into account during garbage collection. Thus a validly referenced **record** could disappear for lack of full information. Yet both filing facilities and garbage collection would seem essential for maintaining long-lived list and tree structures.

7. Outline of a solution

A rather immature solution to this problem has been proposed elsewhere (Baecker, 1973) by the author in the context of ALGOL 68. In later sections we shall borrow ALGOL 68 syntax to give flesh to the present proposal. Initially let us confine ourselves to **class** objects that follow the scope rules, corresponding to ALGOL 68 **loc** values. We shall call them **areas.**

To declare an **area** it must have a name and some size must be given it upon generation. This size could be a language or implementation or installation default, but that is not helpful to the user. But once we place the size under user control that raises the question of what unit the size is expressed in. It must be a unit that is independent of hardware and of implementation if we are to achieve a satisfactory high-level language. We shall borrow from PASCAL and express the size of an **area** as a multiple, or upperbound, of the objects to be generated in the **area.** This removes the need to know the particular hardware and implementation details of the size of integers, strings, etc. But it does not remove the problem of fields in these objects that are arrays with dynamic bounds. PASCAL avoids the problem by not permitting such objects. We could take the same course, or the language could specify some default average expected size per dimension, or we could resort to some facility like an ALGOL 68 **pragmat** to permit the user to give an estimate. If the third option is made available then the second would probably be needed as a default in case the user gave no estimate.

To build list structures we require reference variables. We can try to bind each reference variable at declaration to a particular area, not to the objects that populate it. This follows PASCAL and the reason is clear if we consider the following fragment of ALGOL 68:

```
begin
real y ; ref real yy ;
        .
        .
        .
  begin real x ;
    x := 3·14 ; yy := x ;
  end ;
y := yy ;
end
```

What is assigned to y? In ALGOL 68 it is undefined. The PASCAL solution makes it necessary that the scope of a reference variable is conterminous with that of the object referenced, and so this confusion cannot arise.

We shall here state the rule that if the user wishes to construct rings or other re-entrant list structures then such a substructure must be entirely contained within a single **area.** The rule will be explained below.

If a user is allowed to build complex structures that straddle many **areas**, and to file these in whole or in part, then it must be possible to retrieve only the relevant part of a structure for some later processing. The parts that are retrieved may contain references to **areas** not retrieved, these references can have the well defined value of the null reference. Note that two distinct possibilities for the absence of a part of a structure exist, that no **area** to contain it has been declared in the current program, or that the **area** exists but its expected contents have not yet been retrieved from file.

The obverse of the above discussion is the fact that some program may need to process concurrently **areas** that originate in more than one previous program. The problem that this poses is that of a unique mapping of any cross-references in the current program activation. Declaration ordering cannot provide this. For example: Program 1 generated **areas** A1, A2, A3, which contain cross-references and were declared in that order. Program 2 generated **areas** it identified locally as A1 and A2, which also cross-reference each other. In Program 3 we wish to process all five **areas**. Declaring them locally with different identifiers is no problem, but resolving the cross references is. To achieve this we are forced to seek the same sort of solution as PL/I *OFFSET* locator variables. The type of reference we have described above, bound to an **area**, corresponds to the PL/I *OFFSET*. If, in any program elaboration, each **area** has a locally unique identifier then its whereabouts can be determined as for any other variable. However, the stored reference within an **area** object has to be relative to the target **area.** Elaboration of the reference consists of combining the relative address held as the value of the reference with the origin of the **area** to which it has been bound by declaration.

8. Syntax

The simplest illustration of the proposed syntax for the facility would be a sample piece of program following the style of ALGOL 68.

```
begin
struct student = (int number, ref (course) left, right, ref
  (detail)data);
struct data = (co some structure of fields of student attributes,
  co);
area course [1000*student], detail [1000*data];
ref (course) root := nil, last, where;
data buffer;
int ident;
proc find = (int ID) ref (course):
co find finds a student by his ID on the binary tree in area
  course, if no match is found find returns nil co
begin
ref (course) this := root;
last := nil
while this ¬ = nil and number of this ¬ = ID do
  begin last := this;
        this := if ID > number of this then right of this else
      left of this fi
  end;
this
end;
proc add = (int ID) ref (course):
co add adds a student to the binary tree in area course returning
  his location or nil if a duplicate co
begin
ref (course) here := find (ID), temp;
if here ¬ = nil then nil else
  temp := student := (ID, nil,nil,nil);
  if last = nil then root := temp else
      if ID > number of last then right of last else left of last
      fi := temp
  fi temp
fi
end;
while ¬ logical file ended (stand in) do
  begin read ((ident, buffer));
        where := add (ident);
  if where = nil then ERROR else
     data of where := data := buffer fi;
end;
```

all of which builds a data structure in two areas and does nothing with it.

9. Implementation

Clearly a reference to an **area** object cannot be a hardware address. We propose the technique suggested elsewhere (Baecker, 1970, pp. 407-409). In this case an **area** declared to hold n objects would have a reference vector of length n at its beginning. Each element of the reference vector has two components, a reference count (Collins, 1960) of the number of references to this kth object from other areas and the current relative address of the object within the **area.**

A reference to an **area** object is its cell number in the reference vector, which does not change. This reference, note, is just that, the compiler uses the declarations in the program to generate the association with a particular **area.**

When an **area** object is generated its name becomes the index of the first free cell in the reference vector of its **area.** Garbage collection from time to time will free reference cells and space.

If variable length objects are prohibited then the relative address component of the reference vector is unnecessary, the reference itself can be the relative address, as on garbage collection relocation is then unnecessary. The reference count would then be a hidden field of the object.

Garbage collection is normal in tracing references from the program reference variables to **area** objects and within a given **area.** References from other **areas** are accounted for by the reference count. An object is only collected if it is unreachable from the program local variables or from within the **area** and if its reference count is zero. The reference count is needed to account for references from **areas** not currently present in the program activation. It should now be clear why circular or recursive lists may not straddle **areas.**

When an object is collected the reference count of any referenced object in another **area** that is co-resident is decremented. However, the reference counts of absent objects cannot be decremented. In time **areas** will come to contain inaccessible objects that have not been collected. Should this be a problem in some application then the solution is a utility program that simultaneously garbage-collects all the **area**s on file that cross-reference each other. Such a program is feasible but the parameters it would require are complex.

10. Extension

If this scheme is embodied within a language that already provides a **heap** and an orthogonal system of references, as does ALGOL 68, then the mode **heap area** can be accommodated. The extension is obvious. So is the accommodation of orthogonal references from program variables to **area** object fields. What could not be permitted is that any field of a structure that is an **area** object should be an orthogonal reference, rather than one bound to an **area.** Thus would orthogonality be violated.

11. Conclusion

The desiderata outlined are achievable within an algorithmic language. APL has not been considered and would prove a difficult case.

The run-time overhead of the proposed solution would be high but would be no higher than a user's attempts to provide these facilities without the proposed language extensions, and the overhead is proportional to the use made of the proposed extensions.

Acknowledgement

The author wishes to thank the National Research Council of Canada for their support of this work under Grant No. A7130.

References

BAECKER, H. D. (1970). Implementing the ALGOL 68 Heap, *BIT: Nordisk Tidskrift for Informations behandling*, Vol. 10, No. 4, pp. 405-414.
BAECKER, H. D. (1973). On a missing mode in ALGOL 68, *Machine Oriented Languages Bulletin*, No. 2, April 1973.
BAUER, H. R. *et al.* (1968). *Algol W Language Description* CS110, Computer Science Department, Stanford University.
COLLINS, G. E. (1960). A method of overlapping and erasure of lists, *CACM*, Vol. 3, No. 12, pp. 655-657.
DAHL, O-J., MYRHAUG, B., NYGAARD, K. (1970). *Common Base Language*, Norsk Regnesentral, Oslo.
DATA GENERAL CORPORATION (1971). *EXTENDED ALGOL User's Manual* 093-000052-02.
DEC (1972a). *BASIC-PLUS programming manual*. PL-11-71-01-01-A-D.
DEC (1972b). *BASIC-PLUS Language Manual*. DEC-11-ORBPA-A-D.
DEC (1972c). *RSTS*-11 *System User's Guide*. DEC-11-ORSUA-A-D.
IBM (1965). *PL/I Language Specifications*. Form C28-6571-4.
VAN WIJNGAARDEN, A., *et al.* (1969). Report on the Algorithmic Language ALGOL 68, *Numerische Mathematik*, Vol. 14, pp. 79-128.
WIRTH, N., HOARE, C. A. R. (1966). A contribution to the development of ALGOL, *CACM*, Vol. 9, No. 6, pp. 413-418.
WIRTH, N. (1970). The Programming Language PASCAL, *Acta Informatica*, Vol. 1, No. 1, pp. 35-63.

Book review

Signals, Systems and Controls, by B. P. Lathi, 1974; 524 pages. (Intertext, £8·50.)

In this book Professor Lathi attempts to bring together all the basic ideas of network theory, signal analysis and processing, and control systems. The main approach is with control problems. It is an ideal text for those who wish to learn the elements of modern control theory but as such has little to do with digital computing systems directly (excepting of course those applications of control which contain some element of digital working). The principal use of this book and indeed the one for which it is written is as a text accompanying a lecture course.

The first third of the book presents time domain and frequency domain analysis. As is now customary Professor Lathi presents a unified view in that essentially they are the same apart from using different methods of representing the input signal. Some considerable effort is justifiably expended in showing that the Fourier and Laplace transforms are tools for representing a signal in the complex frequency domain and are not mere mechanical aids to solving integral and differential equations. After treating feed back and control, a chapter is devoted to the increasingly important method of state space analysis. This chapter, which is really introductory to the space state method of representation and analysis, is very clear in its exposition and should help dispel any difficulties in the concept of state of system. For this purpose the author uses the simple device of identifying the initial conditions with the initial state. A seemingly obvious but often ommitted step. The final chapter of the book begins to treat discrete time systems. It introduces the transform and relates it to the Laplace transform for continuous systems and treats transform and state space analysis methods for both discrete data and sampled time systems.

Specialised appendices on differential operators, partial fraction expansion, Bode plots, vectors and matrices and the Nyquist stability criterion are included. The text is extensively sprinkled with mathematics. These are not used to illustrate the virtuosity of the author but to assist the physical understanding. In most cases these mathematical results are interpreted heuristically and further illustrated by simple yet sufficient examples. Professor Lathi has an engaging style which gives the impression that he is communicating personally.

For instance (page 367) 'Anyone who tries to solve a problem this way is bound to say . . . "There must be a better way!". A better way there is.' 'Some readers might find this slightly annoying but for myself it enhances the text.' Indeed Professor Lathi's style leads the reader along at a very fast pace. It is possible to read through quite long sections to obtain an overview of the development and then to return for more detailed study of the mathematics and its full implications.

C. A. MERCER (Southampton)