FILE STRUCTURE AND DATA DESCRIPTION IN THE IMP LANGUAGE

A. Freeman
International Computer Limited
United Kingdom

ABSTRACT

This paper describes work undertaken for ICL and Edinburgh University on the conversational Multi-Access system being implemented on the University's ICL 4-75 Computer.

The language IMP, a derivative of Atlas Autocode, is being used to write the system.

The basic data types available in IMP are:

byte integer
short integer
integer
long integer
real
long real
string

Any data object may exist and be handled in one of four reference modes: direct reference, indirect reference, function call, or function call followed by indirect reference.

As well as arithmetic assignment, address assignment statements are provided which 'point' reference mode variables at data objects.

Compound data objects may be synthesized from simple ones, or from other compound ones by repetition or by structuring.

Repeated objects are termed arrays: array components are referenced by indexing.

Structured objects are known as records. To describe a structured object, a specification of the structure, the format is given separately, which lists in order the components of the structure. This format may then be used to declare records of the given format.

Access to record components is by means of the subfield operator _ (underlined space). If A is a record with components i,j,k, then A_i specifies the component named i.

The compiler guarantees that for all records:

i) The objects of which it is composed will be located in the order described.

ii) Each component will be aligned on the next relevant hardware word boundary.

iii) No hidden address words or code data will be included in the record.

The organisation for record access for optimum efficiency consistent with these constraints is described, with examples.

Examples are given of applications to ~~page and segment~~ list processing, and dictionary description.

1 Introductory Remarks

1.1. The Edinburgh Multi-Access System.

The machine being used is a large paged, segmented machine with drum, replaceable discs and large fixed disc store. The basic software uses the paging and segmentation to provide apparently one-level 'virtual memories' for the users and for much of the system software.

Inside a virtual memory reside all programs relevant to a problem, and all data used by these programs. Data and programs are organized into 'files' which are also used for all I/O and interprocess communication.

The machine is a byte-organized machine in which the principal hardware data objects (i.e. those entities manipulated by the machine order code) may be thought of as

eight bit unsigned BYTES
sixteen bit signed SHORT INTEGERS
thirty-two bit signed INTEGERS
sixty-four bit signed LONG INTEGERS
thirty-two bit floating point REALS
sixty-four bit floating point LONG REALS
up to 256-byte CHARACTER STRINGS

In addition there are sixteen general-purpose high-speed 32-bit registers, and four 64-bit floating point registers.

The order code is very similar in its actions to that of the IBM 360 series, and is compatible at the usercode level.

.2. The language IMP

he system software is being written in the high-level anguage IMP, (IMP lementation language). The language s similar in structure to ALGOL, being actually derived rom the language Atlas Autocode.

t will be released for use with the system, together ith an associated conversational debugging monitor, as he principal programming language for University user's.

1.2.1 General Remarks concerning IMP.

'I don't want it perfect, I want it Thursday' - attrib. Henry Ford.

The first use of the IMP language, and initially the most important one, is for writing systems in. The most important feature of these systems is that they have to work.

The second most important feature is that the system, when written, must be easy to debug, maintain, and modify. It must therefore be written in a form which aids comprehension and communication.

Besides the expected bonuses of speed of writing and ease of communication, IMP is required to provide.

(i) Efficient operation and data handling, within an addressing framework which readily facilitates optimization by the program-writer of key program sections.

Optimization is conceived as something to be undertaken by the user rather than the compiler: The compiler must, however, provide an acceptable framework. The principal optimization techniques are the reservation of high-speed registers for certain purposes, and the rewriting of key sections in machine code. Machine code instructions are provided as a subset of the language, but are generally introduced via user-defined macros to preserve clarity and logical separability.

(ii) Accurate and consistant knowledge of the layout of data objects. Thus the elementary data types of the language correspond closely to those of the machine; and in the definition of complex entities, strict rules may be applied to determine the position of the constituent parts of the entity.

Nevertheless, the primary goals of clarity, logical separability, and speed and ease of use introduce further constraints on the language. Whilst as many as possible of the features requested by system designers are included in the language emphasis has been placed on a sound and consistent logical structure.

2. Data Description in IMP

2.1. OBJECTIVES and EFFECTS OF STORAGE DESCRIPTION

Storage description serves two basic purposes: first, the description in logical terms of the data objects handled by the program; secondly, the specification of the physical layout of these objects.

2.1.1. Logical Description

The techniques of storage description available to IMP Program writers involve using DECLARATIONS to associate symbolic identifiers with data objects. The objects may be simple or complex. Complex objects are synthesized from simple ones by repetition (arrays) or by structuring (records).

The objects may be accessed in several ways: directly, by indirect reference, or by function call. In the case of objects not accessed directly, the identifier will be logically associated, not only with the object itself, but with one (or more) intermediate objects. The latter will be termed ACCESS INTERMEDIARIES. The object itself will be termed the EFFECTIVE OBJECT when it is to be distinguished from access intermediaries. The term 'IMMEDIATE OBJECT' will be used to mean the first access intermediary for indirectly accessed objects, and the object itself for directly accessed objects. The term 'object' will generally signify the immediate object unless EITHER

(i) it is specifically distinguished, OR

(ii) reference is made to attributes such as type or components, which make it obvious that the effective object is under discussion.

Access intermediaries are to be distinguished from objects like dope vectors and array heads to which the user normally has no access and of which he need have no knowledge. The latter will be termed CODE DATA.

The process of using an intermediary to access the effective object (or at a further intermediary) will be termed REALIZATION.

2.1.1.1. Declarations

Declarations comprise a sequence of underlined words termed a DECLARATOR, an identifier or list of identifiers, followed by further QUALIFYING INFORMATION.

Further lists of identifiers with different qualifying information may follow, the same declarator applying.

Example

```
integer i,j,k
integerarray A(1:n),C(1:3)
record format F(integer i, shortintegerarray
                A(1:n*2))
record format F2 (record name R(F))
record R1,R2(F2),R3,R4(F)
record array RI,RJ(1:64)(F),RK(1:3)(F)
string S,T(20)
```

2.1.1.2. Attributes determined by declarations

Declarations associate certain attributes with symbolic identifiers pertaining to the objects they represent. These attributes determine:

(i) The TYPE of the effective object. Types may be simple or complex. Simple types are:

byteinteger
shortinteger
integer
longinteger
real
longreal
string

complex types are:

array
record

in the case of records, qualifying information concerning type is also supplied by giving a RECORD FORMAT IDENTIFIER
n the case of strings, qualifying informa-:tion specifies the maximum length of the string in bytes; in the case of arrays, it specifies the number of dimen-:sions and the upper and lower bound of the index for each dimension.

(ii) The ACCESS MODE of the object: i.e. how the access intermediaries are to be used to access the effective object. Access modes are:

direct access
indirect reference
function call
combination of these.

Indirection is specified by the inclusion of the word name in the declarator following the type descriptor. Function call is similarly

specified by the word fn, and the combination by the words namefn or, alternatively, by the word map.

In the case of an object of access mode 'function call' or 'combined', qualifying information specifies the formal parameters of the function.

In the case of a combined access mode, the attributes also determine in what order succes-:sive realizations are to be carried out, and what intermediary results after each realization.

For example, if 'I' is an integernamefn then the first realization comprises the evaluation of the function to yield an intergername, and the second realization is the use of this to yield the effective object, an integer.

For further information see section 3 on assign-:ment operations and arithmetic expressions.

(iii) The way in which the component parts (if any) of the effective objects are to be reached: by record access or by array access. Objects with component parts can either be handled whole or as a means of referencing component parts. In the case of a record, the type and access mode of the component part is given in a RECORD FORMAT.

A record format associates a layout of component parts with a symbolic identifier (the FORMAT IDENTIFIER). The identifier can then be used to complete the information supplied by a 'record' declarator, so that any number of records in any access mode may be declared, with the given format.

The layout is a bracketed list of declarations; each identifier declared inside the layout is used to identify a component of the record by means of the subfield operator '_' (underlined space). Thus:,

record format F (integer i,j,k)

specifies a layout of three integers in direct access mode, named i,j,k respectively.

record R (F)

declares a record of format F ; i.e. it reserves a block three physical words and associates its address with the symbolic name R.

The record components may be referred to individually by the identifications

R _ i

R _ j

R _ k

respectively.

Record components of any type and access mode may be specified in a format. The subfield operator may be repeated as often as necessary to get at components of deeply nested records: e.g.

man _ address _ town _ country _ ideology

In the case of an array, the declarators preceding the word 'array' also specify:

(iiia) The type of the component parts of the object.

(iiib) The access mode of the components of the object.

The questions of arrays with components of type 'array', and of access modes involving repeated indirection, will be discussed later in the section on restrictions.

2.1.1.3

EXAMPLES

integer A

integer name B

specify A and B as objects of type 'integer' (with no component parts). The access mode of B is 'indirect reference'.

integer array X(1:10)

integer array name Y

define X and Y to be of type 'array'; the component parts being of type 'integer' and access mode 'direct'. The access mode of X is direct reference; and that of Y is 'indirect reference'.

short integer name array Z (-1:N-2)

defines Z to be of type 'array', direct access mode; the components are of type 'short integer', access mode 'indirect reference'.

record name array X (-1:N-2)(F)

defines X to be of type 'array', access mode 'direct

reference', the components being of type 'record (format F)', access mode indirect reference.

record Y(F)

defines Y to be of type 'record(format F)'. The type and mode of the components will be determined by the format F.

3. Assignment Statements and Parameter Passing

Data objects in IMP, having been declared, are specified in other contexts by strings of characters which will be termed DATA SELECTORS.

Example

A
AR(20)
R_i
RA(30*j)_K
R_B(j,k+2)_i

The contexts in which data objects appear are: on the left hand side or the right hand side of address assignment statements; on the left hand side of arithmetic assignment statements; as actual parameters of a function or routine where the formal parameter is declared to be of indirect reference mode (name-type), and in arithmetic expressions.

Arithmetic expressions may appear on the right-hand side of arithmetic assignment statements; as actual parameters of a function or routine where the formal parameter is declared to be of direct reference mode (value-type); as array indices or in array declarations, when integer-valued.

In addition certain special operations on strings exist which will not be dealt with here.

Byte and short integers are always expanded to integer length in expressions.

3.1 Definition of data selector

The simplest data selector is a single identifier, e.g.

i
fred

number of variables.

A data object may also be selected from a complex object by the operations of array indexing and record subfield selection. If X is any data selector, then

(i) if X represents an array of m dimensions, a component of X is represented by X followed by a bracketed list of m integer-valued expressions.

(ii) If X represents a record, and S1...Sn are the names identifying its components, then
X_Si
represents component i

3.2 Arithmetic assignment and expressions

The statement
X = Y
where X is a data selector and Y an arithmetic expression, assigns the value Y to the object X. The expression Y must yield a value of the same type except if X is real and Y integer, and if Y and X are records they must be of the same format.

At present no infix operations on records or arrays are available.

A data selector in an expression which is not in direct reference mode is always realized (section 2.1.1) to produce a value. A data selector on the left-hand side of an arithmetic expression or as a name-type actual parameter is always realized to procuce an address. Thus after
internname n,m
the statement
n = m
means 'put the value referenced by m in the location referenced by n'.

3.3 Address assignment

The statement
X == Y
where X is a data-selector specifying a name type object and Y a data selector of the same type, assigns the address specified by Y to the reference variable X.

In future when X is realized it will produce the effective objec specified by Y.

4. Physical Effects of Declarations

Data objects can be divided into two further categories; DYNAMIC and FIXED-LENGTH objects. Dynamic objects are strings, arrays, or records containing dynamic objects of direct reference access mode.

The significant property of dynamic objects is that their length is unknown at compile time; hence, if space is being allocated in a block or a record, nothing can be located beyond a dynamic object unless referred to via a pointer word.

This has two potential effects:

(i) Objects are physically re-ordered to place the dynamic objects at the end of the stack, in the normal course of events.

(ii) Code data, to which the user has no access, is intermingled with data on the stack, with the result that the user cannot know the physical layout corresponding to his declarations unless he has some knowledge of the internal workings of the compiler.

5. Files

All files on the 4-75 system appear to the user as contiguous areas of his virtual memory; there is no concern on his part with the physical disposition of the file, since the paging and segmentation hardware automatically maps each reference to virtual memory into a physical address.

Files may be created, destroyed, or acquired from outside, by the use of system routines available to the user. Files may also be shared between two users, occupying different virtual addresses in the virtual memories, and may be relocated in a single virtual memory.

5.1 File Structure

A file has associated with it a certain definite structure. This can be considered in two parts.

5.2 Physical Structure

This is imposed on the file by its creator in that he places the components of the file in a certain order. It may be generally assumed that this order is amenable to systematic description.

5.3 Logical Structure

The file creator has some concept of the logical meaning of the objects in the file, and attempts to embody this in the physical structure. He considers the file to contain objects of certain types and precisions, and conceives certain of the objects as being logically grouped together.

5.4 File User

The file user is not obliged to accept the file creator's assumptions concerning the file's logical structure, although he may, of course, be well advised to do so. He must, however, accept the physical structure whether he likes it or not.

In general it may be hoped that the physical structure will to a certain extent reflect the logical. The extent to which it does depends on constraints such as the machine's capabilities, coupled with the need for efficiency and ease of handling. It also depends on the functioning of the program which creates the file. It must be remembered that files will be created by many means beside IMP programs. In addition, it is not reasonable to insist on a fixed and immutable system standard for the layout of all types of objects on a file. In particular, it is not reasonable to expect code data in place on a file.

5.5 Conclusions

(i) A file description must be applicable to a wide variety of layouts, among the layouts not created by IMP programs.
(ii) A descriptive system must not impose the inclusion of code data in ordinary file data, nor can it rely on finding code data on a file.

6. Restrictions

(i) It is not possible to permit the declaration of objects wh
access mode is repeated indirection or repeated function call,

```
integer name name  i,j
integer fn fn  x
```

This is because statements like

```
i = = j
```

(or function assignment if implemented) would be ambiguous unde
circumstances.

(ii) Whilst possible, it is confusing and undesirable to permit declaration of identifiers which could then occur in arithmetic expressions with two sets of actual parameter or index lists:

For example,

```
integer array array  A (1:10) (1:20)
        integer fn array name  B (integer i, j)
        integer array map  K (integer name i)
```

would involve writing statements like

```
x = A(1) (d+2)
x = B(x-3) (20,4)
k (i) (20) = 3
```

For this reason, objects of type 'array' may not have as componen functions, maps or other arrays, nor may arrays be of access mode 'function call' or 'indirect reference followed by function call'

There is no restriction on the types and access modes of record components or the access modes of records.

Thus

```
record fn
    record map
    record name :
```

are all permissible and will be implemented.

For this reason the restrictions (i) and (ii) above may always be 'programmed round' without loss of efficiency:

e.g. for `integer array array A(1:10) (1:20);` `x = A(i) (i+2)`

write/

write

record format ARR (integer array B (1:20))

record array A (1:10) (ARR)

x = A(1) _ B(i+2)

(iii) A fundamental restriction on the use of name-type variables which is necessary to avoid possible self-damage by the user is the following; no name-type object may reference an object to which it is global;

e.g.

```
begin
  integer name i
    routine fred yet again
      integer j
      i = = j
      end
end
```

[This restriction now lifted]

If the routine were called, 'i' would be left pointing to an undefined location.

(iv) It is desirable to allow record formats to mention each other cross-recursively, e.g.

record format HUSBAND (integer age, height, record name a (WIFE))

record format WIFE (integer age, height, record name b (HUSBAND))

This will be allowed, provided

(i) No format may mention records of another format which is not yet described, except in indirect access mode.

(ii) No records may be declared with formats which are not yet described, or which contain records with formats not yet described.

7. Declarations as Applied to Files

In the light of the conclusions at the end of sections 1 and 2, it did not seem that standard declaration procedures, such as stackwise allocation, could reasonably be applied to file description.

However, it would have been unfortunate to have to evolve a completely new storage description scheme for files alone.

The solution proposed was a compromise: declarations were adapted to meet the more rigorous demands of file description in the particular case of records, and a method was devised for applying these descriptions to files.

8. System Standards

Owing to the multi-lingual nature of the Multi-Access System it was necessary to define standards for array access, strings and routine parameter passing to facilitate communication between routines or programs written in various languages.

Because of the relevance to the problem of record access these standards are quoted here: it should be borne in mind that this is only one solution to a difficult problem.

8.1 System Standard For Arrays and Strings

Arrays are accessed through a system standard ARRAY HEAD and DOPE VECTOR. The array head contains four words as follows:

(i) A pointer to the theoretical zero element of the array
(ii) A pointer to the actual start of the array
(iii) A pointer to the dope vector
(iv) A multiplier, for efficient two-demensional array access.

The dope vector contains:

(i) the precision of the array, i.e. how many bytes each element occupies.
(ii) For each dimension, a lower bound and an upper bound on the index for that dimension.

For normal (monitor mode) access, the dope vector is used, For production mode programs, the index is added directly to the pointer to the hypothetical zero element to obtain the address of the element concerned.

Two system standards for strings exist:

A) the string head contains

(i) A pointer to the string elements
(ii) A short integer containing the maximum length of the string.
(iii) A short integer containing the current length of the string.

The string comprises the characters making it up and no furtner information.

B) The string head is as in (A) but the current length is to -1. The string comprises the characters which make up, preceded by a byte containing the current length of the string.

8.2 Layout of Physical Contents of Records

1) All the data objects contained in record are laid out in t order described in the corresponding record format.
2) Each object is aligned on the next relevant boundary, and the record itself starts on the boundary corresponding to the largest element it contains.
3) No code data is stored in the record except that which var for each instance of the object concerned, and is therefor an integral part of the object (e.g. string current length)

8.3 Access to Records Declared On The Stack

1) Each fixed length record declared in the main program is placed on the stack in the scope of a base register, and is accessed by base-displacement addressing relative to this register.
2) Each dynamic record is located after all static objects in the routine in which it is declared. A pointer will be put on the stack in the scope of a base register, via the record will be accessed.

8.4 Access to Record Components

I assume that the address of the record, called aR throughout this section, has either been calculated and is in a register, or known as a displacement relative to some register.

1) All components preceding the first dynamic component are accessed as a base-displacement relative to aR.

2) For each subsequent array, a dope vector and modified array head are allocated in the corresponding record format, henceforth called F. The array head conforms to system standards except that all addresses are relative to the start of the record. An array element address is calculated in the normal way, except that aR is added before use.
3) For each subsequent string, a modified type (B) string head is allocated in F, and the string is accessed by adding aR to the string address in F as for arrays.
4) For each subsequent record or simple object a pointer is allocated in F, which is again relative to the start of the record. The object is accessed by adding aR to this address.

8.5 Name-type variables

All name type variables consist of one word only, which contains the address in virtual memory of the referenced object, or, if an array, to the array head. An exception to this rule may be made for the case of arrays where the four-word array head may be used directly as an array name variable.

Any name type variable may be declared as being relative to some address in problem memory by qualifying the declaration with the phrase.
(relative N)
where N is an address specifier i.e. any IMP data selector
e.g. integer name (relative A (0)) A,B,C

The address specified by N at the time of declaration will always be added to the name - type variable before it is used.

8.6 Mapping Records onto Files

One statements is used for this purpose
(i) map (N1, N2)

[Now written $N_1 <<- N_2$]

Where N1 and N2 are as in Section 5.5. The effect is the same as the address assignment (= =) statement except that no restriction is placed on the types of N1 and N2; i.e. the name-type variable N1 is modified to point at the area of core specified by N2, whatever the type of N2; this area may now be treated as if it were of type N1 by referencing it via N1.

Both N1 and N2 may, of course, be record and/or array components.

It is by means of this statement that programs wishing to communicate via, or to access, external files, can connect to them and reference them.

Functions and procedures to obtain the address of the file, say from external parameters, are provided as part of the File system library mechanism. In addition it is hoped to write a number of machine coded routines to obtain addresses like, for example, the address of the end of a record.

9. Examples

9.1 List processing

A simple list cell may be represented in general by a record plus a pointer Assuming a record format F to have been defined, the construction of a list is as follows:

```
record format CELL(record head (F),record name tail (CELL))
record array list (1:1000) (CELL)
record name next cell, p (CELL)
integer i
i = 1
comment set up the list
LOOP:  list (i) _ tail = = list (i+1)
i = i + 1
-> LOOP unless i = 1000
list (1000) _ tail = = end cell
next cell = = list (1)
comment a function to yield the contents of the
1       top cell of a list and return the cell to the
1       main list
record pop up (F) (record name top (CELL))
top_tail = = next cell
next cell = = top
result = next cell _ head
end
comment a routine to take a cell off the main list,
1       put information in it, and push it down
1       on another list.

routine push down (record info (F), record name top (CELL))
p = = top _ tail
top = = next cell
next cell = = next cell _ tail
top _ head = info
top _ tail = = p
end
```

9.2 Simple Dictionary

This dictionary associates records with identifying tags. The record is assumed to be of format F

```
record format DICT ENTRY (record r (F), string name (40))
record array dictionary (1:1000)(DICT ENTRY)
integer next entry
next entry = 0
comment to add a record R2 identified by 'fred'
dictionary (next entry) _ name = 'fred'
dictionary (next entry) _ r = r2
next entry = next entry + 1
comment a more efficient method
record name index (DICT ENTRY)
index = = dictionary (next entry)
next entry = next entry + 1
index _ name = 'fred'
index _ r = r2
comment a look up function
record namefn look up (F) (string T (40))
integer search
search = 0
LOOP: -> EMPTY if search = next entry
if dictionary (search) _ name = T c
then result = dictionary (search) _r
-> LOOP
EMPTY:
print string ('no luck')
stop
end
```