Data structures and descriptors in the ICL 2900 series and beyond

W. T. Izatt† and E. A. Schmitz‡

This paper studies descriptors and their use as a technique for the implementation of data structures, especially Pascal data structures. After presenting the basic notions of descriptors and types we examine the case of the descriptor on the ICL 2900 series computer, its use and problems for the mapping of Pascal data types. A solution for the problems posed by the ICL 2900-type of descriptor is then proposed. The solution makes use of a more extensive set of descriptors, basically one for each of the Pascal types, and a set of primitive operations on this set enabling the descriptors to be computed and manipulated at run time. This facility is designed to cope with the recursive character of Pascal data types. The resulting scheme is general and simplifies the work needed by the compiler in the translation of names.

(Received March 1980)

1. Introduction

In most computer architectures the semantic information about a type or variable is embedded within the code, without any structure. Information about an array, for example, is distributed in instructions like 'compare bounds' or 'load an element of size x'. An organised technique would have data about bounds and element type stored in a special position which is read each time an access to the array is executed. This position is here called the (array) descriptor.

Since all accesses to data structures of the same kind require the same set of operations, it seems natural to associate with the descriptor some implementation of the primitive access operations required for the particular structure. In the case above, the instruction 'compare bounds' is a primitive of all array access therefore it can be merged in a more general operation 'access array through descriptor' which would execute this checking automatically, so when talking about descriptors it is useful to remember that the term connotes, with its semantic data, a set of basic operations used in data structure access.

In the implementation of language data structures using descriptors, the latter will act as a bridge connecting abstract data structures to concrete computer memory. Since the terms type, data structure and descriptor are very frequent in this report, we start by stating their definition and associated symbols.

Type determines the class of values that may be assumed by a variable or expression. *Structured type* is a type defined in terms of other types. *Data structure* is a structured type together with some operations on that data type (Coleman, 1978).

Descriptor is a data object containing the semantic specification of a type or variable. We refer to a descriptor field using the same dot notation as in the reference to a Pascal record item.

Descriptor template defines the class of values that may be assumed by the descriptor. The template acts as 'type' for the descriptor. The definition of a descripter template is made using the same notation used for Pascal records. When defining physical fields

$$bit\ [n] = \textbf{array}\ [1\ .\ .\ n]\ \textbf{of}\ boolean$$

is used.

Descriptor operations are the set of basic addressing and type evaluating primitives working on descriptors. The description of these operations will be made using the same form as a Pascal function.

2. Pascal data structures and the ICL 2900 descriptor

2.1 *The ICL 2900 descriptor*

The ICL 2900 was designed originally to act as a target language machine, i.e. to match the needs of the intermediate forms of various compilers. The ICL 2900 descriptor is the mechanism designed to handle data structures and pointer variables (Buckle, 1978). There are several types of descriptors in the ICL 2900 (ICL, 1976) but for our present discussion we need only consider the vector descriptor.

The vector descriptor template has the following format:

```
vector-descriptor-template = record
                               tag : bit [2];
                               size : bit [3];
                               s : bit [1];
                               usc : bit [1];
                               bci : bit [1];
                               bound : bit [24];
                               address : bit [32]
                             end
```

All descriptors are 64 bits in length. The fields of the vector descriptor have the following significance

tag identifies the type of descriptor and hence is common to all descriptors. It is 0 for a vector descriptor.

size identifies the size of the elements of the vector described. The size in bits of the element is given by the formula '2 to the power *size*'. The field either does not exist or is not significant in other types of descriptor.

s is ignored. In future machines it will select the extension and truncation rules mentioned in Section 2.2.

usc (unscaled) specifies whether or not any offset applied to the address field should be multiplied by the byte size of the element of the vector. A value of 1 denotes that the scaling should not take place.

bci (bound check inhibit) specifies whether any offset to be applied to the address field should be checked against the bound field to detect an address exception. The offset is never scaled before checking and will be treated as an unsigned integer.

bound contains an unsigned integer which, if the *bci* bit is zero, should be larger by the value of one than the largest permitted modifier.

†Department of Computing Imperial College, 180 Queen's Gate, London SW7 2BZ, UK.
‡Nucleo de Computacao Eletronica University of Rio de Janeiro, Brazil.

CCC-0010-4620/81/0024-0301 $03.50

address is the byte address of the vector's first element.

The operation of the descriptor can be described briefly as the following function, where *D* is the descriptor and *B* is some modifying field.

```
function address-via-descriptor : addresstype ;
  {defines the ICL 2900 descriptor operation}
 begin
  if D.usc
  then scalefactor := 1
  else scalefactor := D.size ;
  if ( not D.bci ) and ( not(B < D.bound ))
  then error
  else address-via-descriptor := D.address + B*scalefactor
 end
```

In the above function the returned value is the resulting memory position of the accessed element.

2.2 *The ICL 2900 addressing modes*

The ICL 2900 instruction set is orthogonal, which is to say that the addressing mode of an instruction may be considered independently of its function.

The ICL 2900 primary instruction set is a set of register operations with the register defined by the function part, but with the associated item specified by the address part.

The size of item loaded into a register is defined by the size of the register. The size fetched from store will be the same size except when it is fetched via a descriptor when it will be the size specified in the descriptor. Then on loading into the register it will be extended or truncated as necessary according to defined rules. The same applies for transfers from a register to main store.

The machine has an accumulator **A** which can take, from time to time, several sizes and performs normal arithmetic. A 32 bit (i.e. full addressing range) modifier register **B** and a 64 bit descriptor register **D** are provided for indirect addressing. Various pointer registers, which we shall group together under the letter **P** are available for direct addressing. In addition there is a conceptual *location* **T**, the top of stack. It can be considered as the first free address above the stack front for transfers to the stack or as the top element (of the size specified as above) for transfers from the stack.

The fetched value is specified in **Table 1,** where brackets mean 'contents of the location evaluated' and *b* means the contents of **B** etc. **N** is the operand field of the instruction.

Table 2 gives the corresponding mnemonics used in the address part of the instruction.

When descriptors are used in addressing, the descriptor may be in main memory or in the special descriptor register **D.** If it is in main memory the first action of the addressing operation is to store it in the descriptor register. The addressing operation can locate the descriptor by the value in one of the pointer registers **P** modified by the instruction's operand field **N,** or by the top of the stack **T.** If this is not possible, the descriptor must be preloaded into **D** using the 'load D' instruction LD.

The location obtained is that specified in the address field of the descriptor, possibly modified by the contents of the modifier register **B** or, where the descriptor is held in the descriptor register by an offset held in main memory or in the operand field **N** of the instruction.

Table 1

Immediate	Via address	Via descriptor		
		In **D**	In store	In store
		(*d*)		
n	(*n*)	(*d*+*n*)		
	(*p*+*n*)	(*d*+(*p*+*n*))	((*p*+*n*))	((*p*+*n*)+*b*)
t		(*d*+*t*)	(*t*)	(*t*+*b*)
b	(*b*)	(*d*+*b*)		

Table 2

Immediate	Via address	Via descriptor		
		In **D**	In store	In store
		.D		
N	.G N	.D N		
	.P N	.DP N	.IP N	.MIP N
.T		.DT	.IT	.MIT
.B	.GB	.MD		

{P = P or X or L or C}

2.3 *Use of the descriptors in array access*

Since arrays are collections of homogeneous items which are typically accessed by a varying and calculated offset, addressing of items is most conveniently achieved by modified addressing via descriptors (columns 3 and 5 of Table 2).

Therefore it is obvious to represent arrays as items addressed by Iliffe vectors, which are the descriptors, modified by the calculated value of the offset. This is all the more true as there are 'move' and 'compare' instructions which operate on whole vectors specified by descriptors.

One dimension of an array can therefore be represented as a vector of elements pointed to and described as to size and number of elements by a descriptor. This arrangement allows automatic scaling and bound checking of the index, so reducing the size of the generated code and simplifying compilation.

2.4 *Use of the descriptors in record access*

Records are collections of non-homogeneous items, which items (in a language like Pascal) are accessed by a fixed identifier. Hence scaling and bound checking are not required and the fetch rules of descriptor access make it impossible for a descriptor to locate and describe a record.

The solution is to reverse the significance of the descriptor and modifier. Then each named field of the record has a descriptor in which the address field now contains the offset in bytes of the item from the beginning of the record.

The record is then located by the modifier and described by *a record template* which is an array of descriptors of its fields in order. This template is common to all records of the same type. It is in fact the description of the type (cf. the descriptor template) and is set up at type declaration time and not at variable declaration time. The actual instruction generated will be the same as for the array case. It is the semantics held in the descriptor and modifier which change.

Since the modifier in fact contains the actual address of the record, a template descriptor (unlike an array descriptor) will not scale the modification, nor will it check it against the bound field. To distinguish this new purpose of the modifier register B we will, in the context of record access, call it the *base* register and refer to the 32 bit addresses suitable for loading into it as *record bases.*

Case variants are considered as separate named items, each with its descriptor in the template. The offsets of these descriptors may, or may not, hold the same values. Pascal does not define which case applies.

2.5 *Use of the descriptors as pointer variables*

Pointer variables are, in this scheme, descriptors. The characteristics of the descriptor depend on the type to which the pointer is bound, as is discussed further in Section 2.8.

2.6 *Use of the descriptors in complex data structures*

It is not possible in the ICL 2900 series to have two modifier/base fields, so that elements of arrays embedded in records would not be accessible without extra coding to manipulate record base and array index offset. To allow descriptors to give automatic memory access we further structure the types such that the *record* is the record base and the template of descriptors and its items whilst an *array* is the Iliffe vector

descriptors and the elements of the array.

Then when we say that a named item is an array we refer in the first instance to its first Iliffe vector descriptor. When we say 'record' we refer in the first instance to its base. Hence when an array appears in a record, it is its first descriptor which appears physically in the record.

Similarly, when one has an array of records, the items of the array are the bases of the records. This is just as well since descriptors, the array access mechanism, only access primitive data items of bit, byte, half- , single- , double- or quadruple-word (i.e. 1, 8, 16, 32, 64 or 128 bits in length).

Arrays of arrays are then arrays of descriptors. Thus since Iliffe vectors are used for multidimensional arrays, multi-dimensionality is sematically identical to arrays of arrays.

Records within records are represented by their bases. Thus like arrays, records contain only the primitive items.

In summary, an array descriptor has scaling and bound checking switched on and a record template descriptor has them switched off.

In this scheme, addressing is automatic and consistent with the data structure being accessed at the time. Furthermore, it is independent of any other data structure which the present one may contain or be contained by, so simplifying compilation.

2.7 *Examples*

Suppose the following Pascal declaration:

```
type
 alfa   = 1 . . 10 ;
 beta   = set of 1 . . 6 ;
 gama   = array [ alfa ] of char ;
 delta  = record
            x : alfa ;
            y : beta ;
            z : gama ;
            u : ↑delta
          end ;
 epsilon = array [alfa] of delta ;
var
 sigma : epsilon ;
```

The data item *sigma* is an array of record bases of type *delta*, that is, it is an array of single word (32 bit) values with ten elements. The descriptor of *sigma*, *d*(*sigma*), is therefore

$d(sigma) = (tag=0, size=5, s=0, usc=0, bci=0, bound=10, address=m)$

where m is the address of the first record base, and the element length in bytes is '2 to the power $(size - 3)$'.

The record type *delta* has a template of four descriptors. They are

$d(x) = (tag=0, size=5, s=0, usc=1, bci=1, bound=0, address=0)$

Field *address* holds the offset of x in *delta*.

Note: because of architectural considerations one 32 bit word is the smallest practical single item appropriate for x.

$d(y) = (tag=0, size=5, s=0, usc=1, bci=1, bound=0, address=4)$

Offset of y = size of x in bytes.

Note: as above, architectural considerations impose a size of 32 bits. Larger sets may use 64 or 128 bits. The latter is an architectural limit after which arrays of bits must be used.

$d(z) = (tag=2, size=6, s=0, usc=1, bci=1, bound=0, address=8)$

A descriptor descriptor. The element in the record is the descriptor of the array z.

$d(u) = (tag=2, size=6, s=0, usc=1, bci=1, bound=0, address=16)$

A descriptor descriptor. The field described is a Pascal pointer which is itself a descriptor, its type depending on the data type to which it is bound.

Note: the previous item in the record was 8 bytes long.

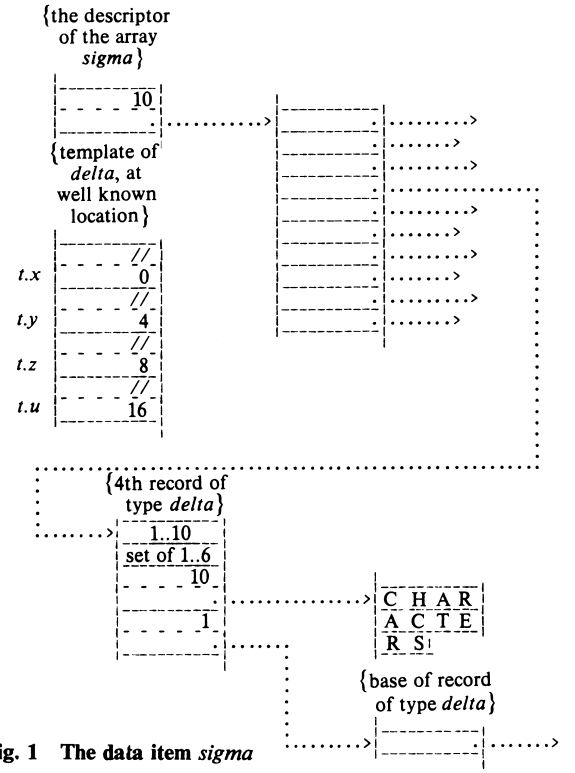

Fig. 1 The data item *sigma*

A graphic representation of the structure of *sigma* is shown in **Fig. 1.**

Consider accessing *sigma*[2].*z*[3]

sigma is an array, therefore load its descriptor into **D**.

LD.L N {in the ICL 2900 assembler}

Evaluate the index on the top of stack, saving and restoring **D**, if necessary. No scaling of the index is necessary because it is done by the descriptor.

sigma [2] is a record, so load its base into **B**.

LB.DT { in the mnemonics }

sigma[2].*z* is a record item. Load its template descriptor into **D**.

LD.X N { from area previously located by register X }

sigma[2].*z* is also an array. Its template descriptor is therefore a descriptor descriptor. (This may be checked if desired). Load the array descriptor from the record into **D**.

LD.MD { located via **D** modified by **B** }

Evaluate the index on top of stack, saving and restoring **D**, if necessary.

Obtain the item *sigma*[2].*z*[3] :

L.DT { one byte is loaded and extended to accumulator size }

Consider obtaining the value of *sigma*[2].*u*↑. As before

LD.L N

LB.DT

LD.X N

sigma[2].*u* is a pointer to a record. The pointer is therefore a descriptor of a 32 bit item which, we may assume, is required in register **B**.

LD.MD {via **D** modified by **B**}

LB.D {load record base in **B**}

Then by loading **D** with descriptors from the template of *delta* we can access the record *sigma*[2].*u*↑ in the same way as we accessed *sigma*[2] above.

2.8 *Problems*

The primary difficulty is in the creating and assignment of complex data items.

The standard Pascal procedure NEW usually allocates an undifferentiated space of the size required by the data item and allocates the address of this space to the pointer variable which is NEW's parameter.

In the present scheme, the process is much more complex. The descriptors and record bases in the space must be written by a complex routine analogous to the process of the compiler itself. Each type will have such a routine which allocates the necessary space as before, but also structures it.

NEW will return not a descriptor to the structure but an *escape descriptor* to the routine for the type to which the pointer variable is bound.

The first operation on a structure accessed via a pointer will be to load into **D** the descriptor which is that pointer. When this is first done, an escape descriptor is detected and at that point space is allocated and structured. The last action of the type's allocation routine is to replace the escape descriptor in **D** with the correct pointer descriptor for the data item allocated. The interrupted instruction is then re-executed and will function normally since the **D** register now points correctly towards the structure (e.g. directly to a simple item, to a descriptor for an array, to a 32 bit base for a record). Now the pointer variable itself must be changed from an escape descriptor to prevent the above action happening again. Therefore the pointer must be stored immediately, i.e.

```
LD.L    N { may cause escape }
STD.L   N { D may be changed }
```

Assignment of complex structures is also more involved. The embedded bases and descriptors prevent a simple MOVE of the total size of the item type as is done in most compilers. The compiler has to insert code for a series of assignments and moves operating on the *terminal nodes* of the structures using the descriptors and bases at the intermediate nodes.

An obvious objection to the current scheme is the extra storage required for structuring a complex item. In the case of the item *sigma* of Fig. 1 the data occupies 180 bytes and the structuring information 240 bytes. This is probably in practical terms the worst case. Only in large programs are space requirements important and the larger structures should have a more favourable ratio. For a more detailed study of data structure composition in large well written programs see Schmitz (1979).

3. An improved descriptor mechanism for Pascal data structures

3.1 *Introduction*

Due mainly to the 2900 array descriptor being a large item (64 bits) the standard architectural solution presented in the preceding section does not give a good ratio of data to structural information in Pascal types mapping (Rees *et al.*, 1978). Although an ad hoc method can improve this ratio, it imposes a penalty on compiler simplicity. Nor does the standard architectural solution give a simple method for assignment of data structures due to the fact that descriptors and data are mixed as seen in record type *delta* of Fig. 1. Generation of dynamic data structures is complicated because structural information must be evaluated at generation time.

This section describes a descriptor mechanism for mapping Pascal data types to computer memory. It consists of a set of type descriptors and three descriptor operators. The idea is to transform a valid Pascal name into a *semantic expression* which when evaluated at run time will give as a result the semantic attributes of the name : address and type.

The semantic expression consists of operands and operators. The operands are descriptors and the operators (which operate on descriptors giving descriptors) have a one to one correspondence with Pascal data selectors.

Our solution tries to cope with the ICL 2900 descriptor problems cited above. First the ratio of data to structural information can be improved by attaching descriptors to types instead of to variables. Assignment and creation of data structures are simplified because data structures are laid down linearly in memory and descriptors are kept separate from data. The main advantage of this method over the traditional compiler evaluation is the obvious simplification of the translation procedure which is one of the aims of language-oriented computer design. This is achieved by delaying all the work related to address and type evaluation of data structure elements to run time.

3.2 *Descriptor objects*

Descriptors are data objects. As such they have a *name* and a *value*. The descriptor value is a set of attributes which characterise some computer object, e.g. variable, file, procedure or another descriptor. Since this paper is discussing data structures, 'descriptor' will hereafter denote a descriptor for data objects only.

Descriptors are complex data objects. The basic units forming the descriptor are called *descriptor fields*. Each one describes one of the attributes of the object. A descriptor field can itself be a complex data object depending on the particular attribute being defined. The *descriptor template* defines the set of values that a descriptor can assume by defining how many and what kind of fields the descriptor has. We have chosen a field partitioning which allows of a simpler algorithm for name translation, as will be shown in Section 3.5.

3.3 *Descriptors for Pascal data types*

A Pascal variable can be defined by its address and type so a Pascal variable descriptor would have two fields, *type* and *address*, the former usually being a complex field. It will be shown that finding descriptors for Pascal variables can be reduced to the problem of finding descriptors for Pascal types. It is also useful to consider type descriptors as objects in themselves. Since types can be shared by variables, the same descriptor can be used in the definition of several variables.

In order to describe all possible type declarations we need at least one descriptor template for each data type. All non-recursive types, i.e. simple types and sets, can be described by a fixed format descriptor template. Arrays and records on the other hand, if one tries to put into their descriptor the entire semantic specification, cannot have a fixed descriptor representation. Fortunately, Pascal restricts the type of operations on structured types. The only operation allowed is assignment of equal type structures which does not depend on any attribute of the type apart from its size. As an example, the semantic data needed for an array x and its element $x[i]$ are different. For the first case, only its size, whilst for the second information about bounds and element type is necessary. There is an exact parallel in the case of records.

This fact gives us the key to solving the problem of the recursive nature of these types. The descriptor template for arrays and records has, apart from its tag, only one field to hold the array or record physical size. Separate templates are defined for array elements and record items.

We show below the associated descriptor template for each Pascal data type (Jensen and Wirth, 1978). The mnemonic *bit* [*n*] denotes an implementation dependent field size.

Primitive types . Primitive types, being predefined and static, have no need for any semantic parameter in their definition. They are defined uniquely by their tag.

```
primitive-type-template =   record
                              tag : (int,char,bool,real)
                            end
```

For example, a declaration like

type *T* = *integer*;

would create a descriptor for *T* denoted by *d*(*T*) as

d(*T*)=(*int*)

Note that from now on we omit the field identifier in front of the descriptor field value, e.g.

d(*T*)=(*tag*=*sca*,*card*=3) becomes *d*(*T*)=(*sca*,3)

Scalar types . A scalar type is defined by a set of constant identifiers over which the Pascal standard functions *pred*(*x*) and *succ*(*x*) are defined. They can be implemented by mapping the constants on to a subset of integers 0 . . *i*, so their semantic description needs only the number of elements in the set. Their template is

scalar-type-template = **record**
tag : (*sca*);
card: *bit* [*n*]
end

For example

type *T*=(*white*,*grey*,*black*)

would generate

d(*T*)=(*sca*,3)

Subrange types . The subrange type is defined by a pair of constants marking an interval over an already defined scalar type. Its template can be defined as

subrange-type-template = **record**
tag : (*subr*);
lcon, *ucon* : *bit* [*n*]
end

Set types . The set type can be semantically identified by

set-type-template = **record**
tag : (*set*);
card: *bit* [*n*]
end

where the field *card* defines the number of elements in the type over which the set is defined.

Array types . The semantic definition of the type array involves two templates. The first is a descriptor for the whole array

array-type-template = **record**
tag : (*arr*);
size : *bit* [*n*]
end

where *size* is a field to hold the array physical size, e.g. in bytes. We have a second template for the array elements:

array-element-template = **record**
tag : (*arl*);
index : *simpletypetemplate*;
element : *typetemplate*
end

where *index* is a template of an allowed simple type and *element* is that of any type.

Record types . As in the array case, there are two templates defined for records.

record-type-template = **record**
tag : (*rec*);
size : *bit* [*n*]
end

record-item-template = **record**
tag : (*fld*);
item : *typetemplate*;
offset : *bit* [*n*]
end

where *offset* is a field holding the physical distance of the item from the beginning of the record.

Pointer types . Since pointers are defined over an already defined type, their semantic specification does not need any semantic fields (they are already in the pointed type).

pointer-type-template = **record**
tag : (*ptr*)
end

We will now show how descriptors for those types are used to form the descriptors of more complex types. In the left column there is a Pascal declaration, and in the corresponding right column we find its descriptor. In the complex type descriptors, parentheses are used to indicate fields which are themselves descriptors.

In following example we use the symbols:

d(⟨*id*⟩) — for the descriptor of ⟨*id*⟩
d(⟨*id*⟩–*e*) — for the descriptor of an element of array ⟨*id*⟩
d(⟨*id*⟩–*p*) — for the descriptor of the type to which the variable ⟨*id*⟩, a pointer, is bound.

type	
alfa = 1. .10;	*d*(*alfa*)=(*subr*,1,10)
beta = **set of** 1. .6;	*d*(*beta*)=(*set*,6)
gama = **array** [*alfa*] **of** *char*;	*d*(*gama–e*)=(*arl*,(*subr*,1,10), *char*) *d*(*gama*)=(*arr*,10)
delta = **record**	
x : *alfa* ;	*d*(*x*)=(*fld*,(*subr*,1,10),0)
y : *beta* ;	*d*(*y*)=(*fld*,(*set*,6),1)
z : *gama* ;	*d*(*z*)=(*fld*,(*arr*,10),2)
u : ↑*delta*	*d*(*u*)=(*fld*,*ptr*,12)
end;	*d*(*delta*)=(*rec*,16)
epsilon = **array** [*alfa*] **of** *delta*;	
	d(*epsilon–e*)=(*arl*,(*subr*,1,10),(*rec*,16))
	d(*epsilon*)=(*arr*,160)

3.4 *Descriptors for Pascal variables*

Given a set of descriptor templates, one for each data type, we can generate descriptors for any Pascal variable. The descriptor for a variable is defined by two fields, a data attribute field which is the type descriptor to which the variable is bound and an address field.

The format of any variable descriptor can be defined as:

variable-descriptor = **record**
attribute : *type template*;
address : *bit* [*n*]
end

For example suppose a declaration like

var *sigma* : *epsilon*; *d*(*sigma*)=(*d*(*epsilon*),*address*)

but

d(*epsilon*)=(*arr*,160)

and if *sigma* is bound to location 300 in store then

d(*sigma*)=((*arr*,160),300)

3.5 *Descriptor operators*

Given this semantic description of a data structure we can get the descriptor of one of its elements by using specific *descriptor operators*. The ICL 2900 addressing via descriptor and modifier is in essence a descriptor operator which evaluates the array element descriptor from the array descriptor and an index.

When a single element which is part of a data structure is referenced, it is denoted by a series of selectors applied to the highest hierarchic name in the data structure. One way of thinking about a cascade of selectors is as constituting a series of operators applied on data types.

Our main constraint in the design of the descriptor operators was the need for a simple translation algorithm to minimise the work done by the compiler when generating code for a Pascal name. The second restriction is *one-symbol-look-ahead*

which implies that the analysis of names must be done in a single scan from left to right. Additionally, during evaluation the system should use the normal data stack without any special features. At the end of the evaluation process the resulting descriptor should be at the top of the data stack. These conditions allow a very simple and structured technique for evaluating Pascal names since all evaluations, both of expressions and descriptors, are made on the same stack.

The simplest way of fulfilling the above conditions is by a simple one to one replacement of the Pascal '[' the array selector, '.' the record item selector and '↑' the pointer selector by three descriptor operators which we call **bracket, dot** and **arrow**.

For example, a name like *sigma*[*y*].*z* would be converted by the compiler into the reverse Polish string

d(*sigma*) *d*(*sigma–e*) *y* **bracket** *d*(*z*) **dot**

where *d*(⟨*id*⟩) implies 'load the descriptor of ⟨*id*⟩ to the stack'. In this case **bracket** would operate on *d*(*sigma*), *d*(*sigma–e*) and the value of the expression *y* to produce the descriptor of *sigma*[*y*], which combined with *d*(*z*) by the operator **dot** gives as result the address and type of *sigma*[*y*].*z*.

This means a transfer of the operations made by the compiler when generating code for *sigma*[*y*].*z* to run time. The Appendix shows the name translation procedure. The descriptor operators assume a resulting descriptor with the format

```
result =        record
                  type: typetemplate;
                  address: bit [n]
                end
```

We now assume the following functions

length(*x*)—is a function that when applied to the descriptor argument *x* returns the size (in bytes) of the element described by *x*.

value(*x*)—is a function which returns the value of the object described by the descriptor *x*.

lbound(*x*)—the argument is a simple type descriptor and the function returns the value of the lower bound of the type specified by the descriptor.

The **bracket** operator, given an array descriptor *da* an array element descriptor *de* and the evaluated index *i*, generates a descriptor for the array element variable. Its operation can be defined by the following procedure (operands being assumed to be global)

```
procedure bracket;
  {generate a variable descriptor for the array element}
 begin
  result.type := de.element;
  result.address := (i - lbound(de.index))
                    *length(de.element)
                    + da.address
 end
```

This means the generation of a variable descriptor whose type is the element field of the array element descriptor and whose absolute address is the sum of the base address of the array and the product of the relative index and the array element size.

The **dot** operator gives as result the semantic characteristics of the item being selected inside a record. If *dr* is the record descriptor and *dri* is the record item descriptor then its operation can be defined as:

```
procedure dot;
  {generate the descriptor for the record item}
 begin
  result.type := dri.item;
  result.address := dr.address + dri.offset
 end
```

This means the generation of a variable descriptor whose type is the same as the item descriptor and has as address the sum of the base address of the record and the item offset.

The **arrow** operator gives as result the semantic description of a pointer selected variable. As the pointer variable descriptor *dp* describes a pointer variable whose contents point to a variable of type *t*, the result is the creation of a variable descriptor of the same type *t*, having as address the contents of the pointer variable. This can be defined as

```
procedure arrow;
  {generate the descriptor of a pointed variable}
 begin
  result.type := t;
  result.address := value(dp)
 end
```

3.6 *Examples*

We want to show that, given a name in its textual form with all the descriptors associated with it, we can form descriptors for its elements. In the following examples we assume the variable *sigma* bound to memory location 300 and that descriptor evaluation is taking place on the same stack as for expression evaluation.

Using the same type definitions as in Section 3.3 and the beginning of Section 3.4, valid Pascal names defined over *sigma* are:

sigma
sigma[2]
sigma[2].*x*
sigma[2].*z*
sigma[2].*z*[3]
sigma[2].*u*↑.*z*

Case 1 . The name is *sigma*. The descriptor of *sigma* is
d(*sigma*) = (*d*(*epsilon*),300) = ((*arr*,160),300)
which means that *sigma* is an array of size 160, starting at location 300. Note that no other semantic information is needed, since Pascal operations on data structures are limited to assignment.

Case 2 . The name is *sigma*[2]. Its descriptor is derived from the reverse Polish expression
d(*sigma*) *d*(*sigma–e*) 2 **bracket**
By definition of **bracket**
d(*sigma*[2]) = ((*rec*,16),316)

Case 3 . The name is *sigma*[2].*x*. Its descriptor is defined by
d(*sigma*[2]) *d*(*x*) **dot**
which is in detail
((*rec*,16),316) (*fld*,(*subr*,1,10),0) **dot**
By definition of **dot**
d(*sigma*[2].*x*) = ((*subr*,1,10),316)

Case 4 . The name is *sigma*[2].*z*. Its descriptor is defined by
d(*sigma*[2]) *d*(*z*) **dot**
The result is an array descriptor with the attributes of type *gama* and with address 318.
d(*sigma*[2].*z*) = ((*arr*,10),318)

Case 5 . The name is *sigma*[2].*z*[3], with descriptor defined by
d(*sigma*) *d*(*sigma–e*) 2 **bracket** *d*(*z*) **dot** *d*(*z–e*) 3 **bracket**
which expression, when evaluated from left to right, gives the descriptor of a variable of type *char* at address 320.
d(*sigma*[2].*z*[3]) = (*char*,320).

Case 6 . The name is *sigma*[2].*u*↑.*z*, Suppose that a Pascal instruction NEW(*sigma*[2].*u*) was issued before, allocating a record of type *delta* at position 1000 in memory. Evaluate
d(*sigma*[2]) *d*(*u*) **dot** *d*(*u–p*) **arrow** *d*(*z*) **dot**
from left to right, giving the descriptor of an array.
d(*sigma*[2].*u*↑.*z*) = ((*arr*,10),1002)

4. Conclusions

The solution offered by the ICL 2900 descriptor mechanism for the implementation of Pascal data structures is very space consuming. Also no simple solution for assignment of data structures or creation of dynamic data objects is offered. While our alternative method may not have been economic when the ICL 2900 series was designed, the rapidly falling cost of hardware leads us to believe that it is becoming practical.

To sum up, this paper suggests the following.

To be used efficiently as a technique for implementation of data structures, descriptors must be devised in several formats and with a matched set of machine primitives.

Since they are the expression of the primitives used in data definition and access, these requirements tend to be language dependent.

They imply a machine able to work with variable size operands (descriptors) having an implementation of the descriptor operators either implicit or in the machine instruction set.

Since it uses the normal data stack in the evaluation of semantic attributes made at run time, the compiler will be smaller and faster. The code generated will be more compact. giving gains at execution time.

Appendix—The name conversion algorithm

The process of name conversion can be described by a Pascal algorithm. The global variable *symbol* contains the next symbol to be analysed. Fetching of the next symbol is omitted from the text to simplify its readability.

```
procedure convert;
  {convert a Pascal name to a semantic expression}
 begin
  generate ('descriptor of variable');
  if (symbol in ['[', '.', '↑'])
  then case symbol
       of
       '[': begin
            generate('descriptor of array element');
            expression;
            generate( bracket)
           end;
       '.': begin
            generate('descriptor of record item');
            generate(dot)
            end;
       '↑': begin
            generate('descriptor of pointed type');
            generate(arrow)
           end
       end {case}
 end {convert}
```


References

BUCKLE, J. K. (1978). *The ICL 2900 Series*, Macmillan Computer Science Series. Macmillan, London.

COLEMAN, D. (1978). *A Structured Programming Approach to Data*, Macmillan Computer Science Series. Macmillan, London.

ICL (1976). *ICL 2900 Primitive Level Interface*, Product Specification Document 2.5.1. International Computers Ltd, Bracknell, UK.

JENSEN, K. and WIRTH, N. (1978). *Pascal User Manual and Report*, 2nd Edn (corrected). Springer, New York.

REES, M. J., GOODSON, J. I., REYNOLDS, J. J. and ZELL, H. J. (1978). Pascal on an advanced architecture, in *Pascal—The Language and its Implementation*, edited by D. W. Barron. Wiley, Chichester.

SCHMITZ, E. A. (1979). A study of static data type usage in Pascal, Research report no. 79/17. Department of Computing, Imperial College, London.

Book review

Project Auditing Methodology, by W. S. Turner III, 1980; 454 pages. (*North-Holland*, $46·25)

The book is a detailed review of procedures developed within the author's organisation for auditing EDP projects and said to be applicable also to other types of development project. It is crammed with information on all facets of a project. A substantial chapter is devoted to the audit report and the importance of its content and presentation. Other features of particular value are the list of problem areas to look for in each aspect of the project, the extensive audit check lists and evaluation forms, and the treatment of contract details and relationships between contractor and client. There is also a wide-ranging annotated bibliography.

The author's approach involves three levels of audit: first an overview audit to obtain a preliminary assessment of the state of a project; second, an administrative audit to cover all general aspects in detail; third, one or more technical quality audits where required to assess the quality of work done. If every project area was audited in the detail described by the author, the auditor would risk achieving notoriety for high audit costs and excessive interference with the project. Clearly this is not intended. Nevertheless, it would have been helpful to have some indication of audit areas to concentrate on in the more usual circumstances of restricted time and access.

The scope of work to be done goes beyond traditional audit activities to areas where the prime intention is to provide information for management decisions. In my opinion detailed forecasting of the future course of a project is more properly a project management responsibility and the auditor's function should be restricted to checking that forecasting procedures are adequate. The same type of comment applies to some other topics, which the author deals with at length in the context of providing a service to management, such as valuation of work done, quality of work done and contractor's profit expectations. In my view management should not have to rely on the audit function for essential information of this type; this risks destroying the auditor's independence. In contrast, there are some important areas in a project audit which are curiously omitted, for example the evaluation of project management controls over the work of the contractor; indeed, it appears that the audit itself is to be the primary control tool. There is no mention of timewriting procedures and related controls which form an essential feature of a reimbursable contract. There is no discussion of checks on controls being built into the system to preserve security, reliability and accuracy. Little is said of the auditor's involvement in system testing; this is an essential activity if the audit group is to remain associated with the project in the operation phase.

While in critical mode I must take issue with the author over his belief that recommendations should be excluded from the audit report, or at best be relegated to a separate document. His views do not stand up to close scrutiny. I believe that carefully presented recommendations assist project management to a fuller understanding of the significance and relative importance of weaknesses identified in the findings.

Despite these criticisms there is so much of value in the book that it must be a useful addition to the library of any auditor, project manager or specialist with the experience to select procedures of most benefit to his own environment.

J. R. JONES (London)