Hybrid Data Structures Defined by Indirection

N. E. GOLLER

Operational Research Executive, NCB, Coal House, Lyon Road, Harrow, Middlesex, UK; now at Faculty of the Built Environment, Polytechnic of the South Bank, Wandsworth Road, London SW8

In a typeless language such as BCPL, data structures may easily be defined by means of an indirection operator. This useful way of defining data structures is here extended so as to apply to a type-rich language.

Each structure is defined by its structure graph, which is built at compile-time according to certain rules that are explained. A structure built in this way is called a hybrid data structure. Examples are given to show how these structures may be used in programming.

1. INTRODUCTION

In this paper I shall describe what is meant by a hybrid data structure and examine how such structures can be used for programming. The syntax used here is not the most compact that could be developed, but is correspondingly easier to define and more self-explanatory. It is an example of what might be done, not a prescription of what should be.

In a similar non-prescriptive spirit, I shall describe a simple implementation on a conceptual bit-addressable machine, while ignoring modifications that would improve efficiency or that would be needed to deal with alignment problems on a word-addressed machine.

To illustrate the nature of hybrid data structures, consider a data structure with two types of component: a series of *vector components* all of the same type, and a series of *list components* of diverse types. Such a structure may be used for example to represent a string (Fig. 1).

int
char
char
char

Figure 1. A string that knows its length.

Here the vector components represent the characters of the string and are held in consecutive 1-byte cells. There is here just one list component, an integer representing the length of the string, held in (say) a 2-byte cell.

The components of such a structure S will be accessed by the following notation. Each vector component will be denoted by an expression S:*n*, where *n* is an integer. Each list component will be denoted by an expression S.*id*, where *id* is an identifier that has been associated with this component. Now S itself will denote a cell containing a memory address; and the colon in S:*n* and the period in S.*id* are *indirection operators*, to be evaluated by the following algorithm:

```
examine structure graph of S to determine offset
    associated with n or with id;
add offset to value held in S;
result is address of component.
```

The structure S, accessed by indirection operators in this way, is now a simple kind of hybrid data structure. The way that a hybrid data structure is defined by a directed graph, its structure graph, is detailed in sections 2 and 3 of this paper. Section 2 deals with a particular kind of structure whose structure graph is always a tree; section 3 deals with the general case.

In so far as data structures are defined by indirection, the method of data structuring described here is a generalization of that found in the language BCPL (see Ref. 1). In BCPL, an untyped language, there is no need to examine structure graphs to evaluate the indirection operator, since all objects appear to the compiler to have the same structure. By using structure graphs we shall instead be able to define indirection operators appropriate to a type-rich language.

The construction described here is one that allows all type-checking to be done at compile-time, as is discussed in section 4. The way that hybrid data structures are passed as arguments to procedures is discussed in section 5: this leads to a consideration of procedural classes and data abstraction.

2. SCALARS AND TREE STRUCTURES

The core of our conceptual machine is a sequence of addressable locations each containing one bit. Data storage in this bit-addressable memory is in cells, where a *cell* contains a certain number of consecutive bits. The *type* of a scalar object says what size of cell it uses (e.g. 32 or 16 bits) and how the bit-pattern in such a cell is to be interpreted by the programming language (e.g. as a real number or as an integer). I shall assume that a sufficient set of elementary scalar types has already been defined; for simplicity I shall assume these types are **int**(eger), **real**, **char**(acter), and **bool**(ean), each with their defined representation conventions and cell sizes (measured in bits) *intsize*, *realsize*, *charsize*, and *boolsize*.

Now to implement indirection we need a type whose value (after its bit-pattern has been interpreted) is a memory address. This type will be called **loc**, with size *locsize*. For reasons that will become clear, **loc** is an internally used type, not encountered by the programmer. Instead he will encounter two types **link** and **ref** which are held in cells of size *locsize*. The use of type **ref** is postponed to the following section, therefore we shall work now with the scalar types **int**, **real**, **char** and **bool**, and the non-scalar type **link**.

The following syntax is used to declare scalar variables:

identifier **is** *type*

where *type* is **int**, **real**, **char** or **bool**. Following such a declaration the compiler arranges for one cell of the appropriate *typesize* to be allocated to store the variable. Assignments to the variable will then change the contents of this cell.

The declaration for one-level tree structure is in two stages. First we declare the identifier to be of type **link**. Next we declare what it points to, by a declaration taking the form:

identifier **pointsto** (*componentlist*)

where *componentlist* contains zero or one items of the form **vec** *i* **thru** *j type* (in which *i,j* are integer constants representing the subscript bounds), followed by zero or more items of the form *identifier* **is** *type*. Fig. 2*a* shows such a declaration for the tree structure STRING to hold a variable-length string up to 80 characters long. The syntactic effect of this declaration is that we can subsequently use STRING:*n* for suitable *n* to denote the *n*th character of the string, and STRING.LENGTH to denote an integer that will (presumably) be used by the programmer to hold the string length.

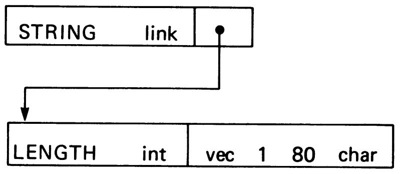
To implement this declaration for STRING the compiler will build the *structure graph* shown in Fig. 2*b* and will organise the allocation of **loc**, **int**, and **char** cells, and initialization of the **loc** cell, to give the run-time structure of Fig. 2*c*.

(*a*) Declaration

STRING **is link**

STRING **pointsto (vec** 1 **thru** 80 **char**, LENGTH **is int)**

(*b*) Structure graph



(*c*) Run-time storage

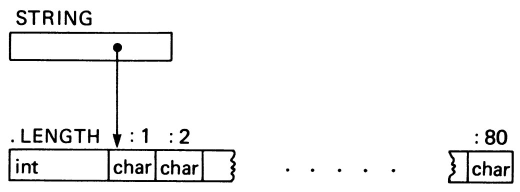


Figure 2. The hybrid data structure: STRING.

The declaration of multi-level structures follows a similar principle. For each non-scalar component one specifies first that it is a **link** and next what it points to. Fig. 3 illustrates this for a structure called BUFFER and Fig. 4 for a rectangular real array MATRIX. Since all the components MATRIX:1,...,MATRIX:100 point to a similar structure, the keyword **any** is used in the declaration to denote an arbitrary integer within the required range.

At run-time the **loc** cells MATRIX:1,..., MATRIX:100 will in effect form an Iliffe vector (see Ref. 2, p. 141) through which the scalar components MATRIX:*i*:*j* are accessed. In the absence of bracketing, the BCPL convention is used of evaluating the leftmost indirection first: thus MATRIX:*i*:*j* is interpreted as (MATRIX:*i*):*j* and BUFFER.FILENAME:3 as (BUFFER.FILENAME):3.

Now to declarations for variables one should attach (explicitly or implicitly) a specification of the scope and lifetime of the variable. The details will depend on other features of the language. We can ignore this matter since it does not affect the shape of the structure graph.

Structure graphs will be shown here as being built from an abbreviated version of the character-strings used in the declaration. With each **link** node is associated a pointer to the appropriate list of structural components. The items of this list appear in the reverse order to their occurrence in the declaration. The **link** pointer points to the leftmost (last-declared) item. In the absence of a vector component the keyword **vecnull** marks the end of the list.

The following terminology is used. A *node* is a run-time or compile-time object denoted by an expression legitimately formed from a variable by indirection operators. Formally the set of run-time nodes is the smallest set such that:

(i) If X is the declared identifier for a variable then X is a node.
(ii) If X is a node of type **loc** that points to one or more vector components, then X:I is a node whenever I is an expression of type **int** whose value is in the declared subscript range.
(iii) If X is a node of type **loc** that points to one or more list components, then X.*id* is a node whenever *id* is the declared identifier for such a component.

Here type **loc** includes types **link** and **ref**.

For example BUFFER, BUFFER.FILENAME, BUFFER.FILENAME:1 and BUFFER.FILENAME:(I+J) are run-time nodes – assuming I+J evaluates to an integer between 1 and 32 – whereas I+J is not a node. These run-time nodes map on to the compile-time nodes BUFFER, BUFFER.FILENAME, and BUFFER.FILENAME:**any** of the structure graph.

Any run-time node is also a cell, but a cell need not be a node – for example intermediate results from the evaluation of an expression may be held in appropriately sized cells but these cells cannot be accessed by the programmer using indirection expressions.

A compile-time node of **loc** type is said to be *resolved* if its associated pointer points to another compile-time node that has already been built. After the declarations above the broken line in Fig. 3*a* have been processed by the compiler, that part of the structure graph above the broken line in Fig. 3*b* is present. The compiler will detect that the declaration is so far incomplete by the presence of unresolved **link** nodes. One requires that when a new **link** or **ref** node in the structure graph is built, its pointer is initialized to a recognizable 'nil' value that does not point to anything.

The details of evaluating indirections are as follows. Let X be a run-time **loc** node. The value held in the **loc** cell denoted by X will be the address of the first bit of the first vector component that X points to. Given the declaration:

X **pointsto (vec** *i* **thru** *j typev*,
id1 **is** *type1*,..., *idr* **is** *typer*)

(*a*) Declaration

```
BUFFER is link

BUFFER pointsto (vec 0 thru 1023 char, LENGTH is int,
                 FILENAME is link, STATUS is link)
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
BUFFER . FILENAME pointsto (vec 1 thru 32 char, LENGTH is int)

BUFFER . STATUS pointsto (EOF is bool, ERR is bool)
```

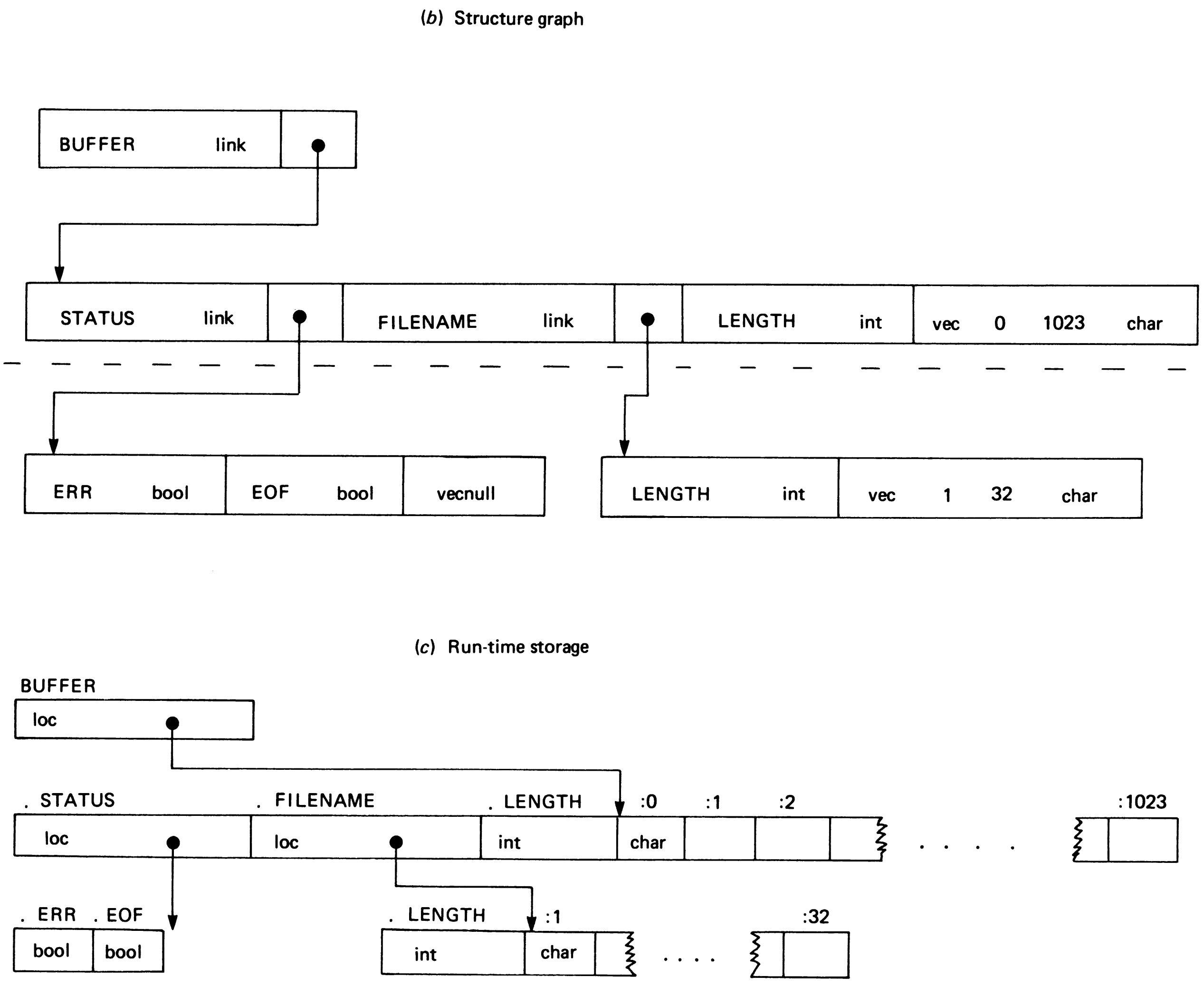


Figure 3. The structure BUFFER.

then X:n has address (X+*offsetv*) where *offsetv* = $(n-i)$**typevsize*; and X.*idk* has address (X+*offsetk*) where *offsetk* = −(*type1size*+*type2size*+…+*typeksize*). The offset associated with a list component is negative and the list components are stored at run-time in the reverse sequence to that occurring in the declaration.

When a tree structure is created, it is intended that the structure remain in force throughout the lifetime of the variable. Therefore the only cells of a tree structure that may be altered after allocation are the scalar components of the structure, corresponding to the leaf nodes of the structure graph. Consequently we shall insist that the left-hand side of an assignment statement must be a node of scalar type. (A less rigid rule could be introduced, but is not discussed here, in which assignment to a tree structure is interpreted as multiple assignment to its leaf nodes.)

Once the **link** nodes for a structure have been resolved, the compiler can proceed to arrange allocation for the whole structure, including code to initialize the contents of **link** cells. Each run-time **link** cell will point to the first vector location in a block of consecutive storage containing list and vector components. The constraints on consecutivity of storage are precisely those that arise from the method just given for evaluating indirections.

A tree structure, namely any structure that may be built

(*a*) Declaration

```
MATRIX is link

MATRIX pointsto (vec 1 thru 100 link)

MATRIX : any pointsto (vec 0 thru 4 real)
```

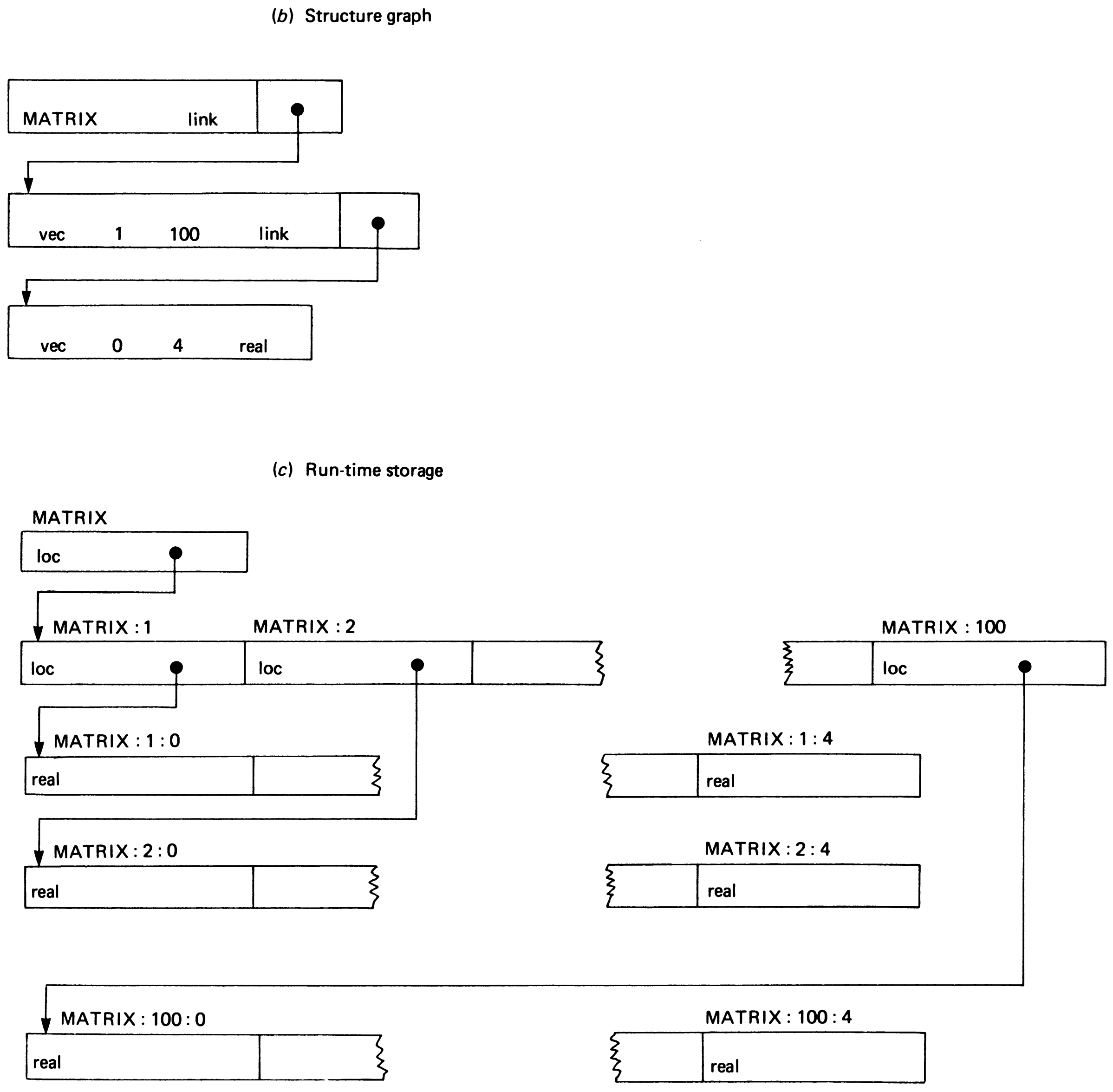


Figure 4. The structure MATRIX.

as described in this section using **link** and **pointsto** declarations, is seen to have the following properties:

(1) the structure graph is necessarily a tree;

(2) the values held in **loc** cells of the structure cannot be changed after allocation;

(3) space for an instance of the structure is allocated as a result of its declaration.

The next section explains how structures may be built that do not have these properties.

3. REFERENTIAL STRUCTURES

To declare a referential structure we shall use **ref** in place of **link** in its declaration. A node of type **ref** is regarded as a scalar, and its value may be altered by assignment. In other respects its function is similar to that of a **link** node. The resulting mechanism has clear similarities to the indirection mechanisms in BCPL, of which it is essentially a generalization; but differs markedly from the way that **ref** is used in Algol-like languages. In particular, coercions (referencing and dereferencing) do not occur. (See Ref. 3, p. 36).

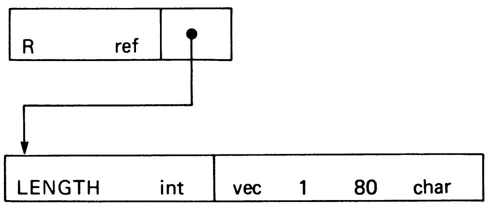
Fig. 5*a* shows the declaration for a referenced string R. The syntactic effect of this declaration is as follows. Suppose STRING is an actual string for which space has been allocated as in Fig. 2. Then the assignment R = STRING causes R to point to the same space, by altering the value held in the **loc** cell R. Following this,

(*a*) Declaration

R **is ref**

R **pointsto** (**vec** 1 **thru** 80 **char**, LENGTH **is int**)

(*b*) Structure graph



(*c*) Run-time storage

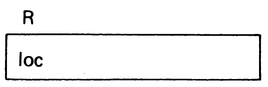


Figure 5. A referenced string.

(*a*) Declaration

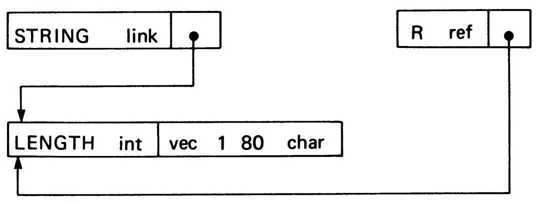
STRING **is link**

STRING **pointsto** (**vec** 1 **thru** 80 **char**, LENGTH **is int**)

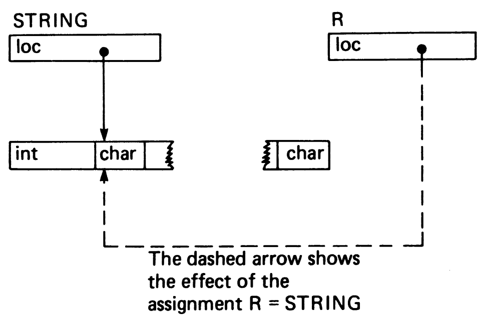
R **is ref**

R **pointslike** STRING

(*b*) Structure graph



(*c*) Run-time storage



STRING
loc
R
loc
int
char
char
The dashed arrow shows the effect of the assignment R = STRING

Figure 6. Alternative declaration for referenced string.

assignments such as R:12 = 'Z' or R.LENGTH = R.LENGTH+1 will cause the corresponding components of STRING to be altered. This follows automatically because the structure graph of R, from which the offsets are computed, is identical to that of STRING. The space allocated for the structure R is just the **loc** cell needed for the **ref** node itself.

Since a variable that references an object is useless unless an actual object of that type exists, it will often be convenient to declare the latter first and to use the abbreviation **pointslike** as shown in Fig. 6.

The way that reference declarations are compiled is as follows. The compiler sets up a structure graph just as it would for a tree structure. A **pointsto** declaration causes the next level of the graph to be built, as before, whereas a **pointslike** declaration shortcircuits this process by making the **ref** node point to a part of the same or different structure that has already been built.

Now any structure that can be completely declared using **link**, **ref**, **pointsto**, and **pointslike** in the manner described in this section is a *hybrid data structure*. The declaration of a hybrid data structure is complete once all **link** and **ref** nodes in the structure graph have been resolved.

Let X be a hybrid data structure, or more generally any compile-time node. The *reduced structure tree* of X is the tree with X as root obtained by resolving all **link** nodes but not resolving any **ref** nodes. Each leaf of the reduced structure tree is of a scalar type. A tree structure is one containing no **ref** node, and in this case the reduced structure tree and the structure graph coincide. The space allocated for a hybrid data structure is only that needed by the run-time nodes that are represented in its reduced structure tree.

A **ref** node may occur at any level of a hybrid data structure as a list or vector component.

The declaration X **pointsto** (*componentlist*) is equally valid whether X is a **ref** node or a **link** node.

The declaration X **pointslike** Y is legal provided that:

Either **Case 1**

X is a node of type **ref**;

Y is a node of type **ref** or **link**;

Node Y has been resolved.

(Nodes below Y do not need to have been resolved; node Y itself must have been resolved since X is to be made to point to whatever Y points to. Later additions to Y's structure graph will apply to X automatically, and vice versa.)

Or **Case 2**

X is a node of type **link**;

Y is a node of type **link**;

Y and all **link** nodes below it have been resolved, i.e. Y's reduced structure tree has been built.

(The purpose of including Case 2 is merely to allow the convenient and efficient abbreviation. The restriction on Y ensures that any structure built using **link** and **pointslike** is equivalent to a tree that could instead have been built using **pointsto**.)

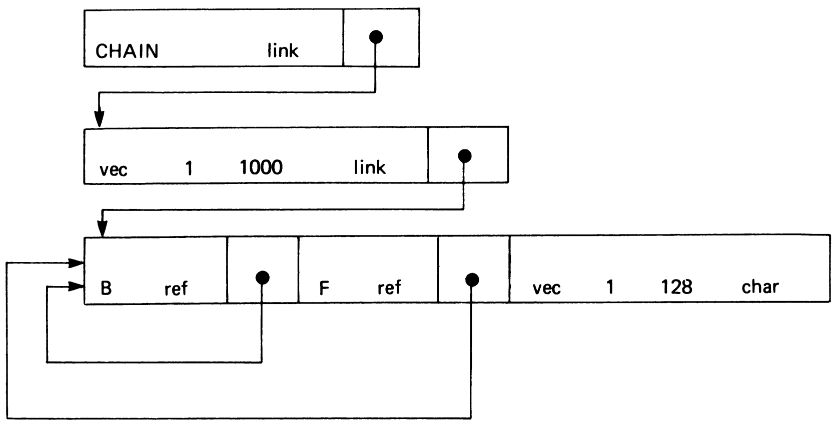
If X is a node of type **ref** and Y a node of type either **ref** or **link**, then the assignment X = Y is legal.

A hybrid data structure can be self-referential. Fig. 7 illustrates the use of **ref** to create a self-referential structure graph for a data chain with forward and backward pointers. The use of this structure is exemplified in Fig. 8, a program that unlinks element X

(*a*) Declaration

```
CHAIN is link

CHAIN pointsto (vec 1 thru 1000 link)

CHAIN : any pointsto (vec 1 thru 128 char, F is ref, B is ref)

CHAIN : any . F pointslike CHAIN : any

CHAIN : any . B pointslike CHAIN : any
```

(*b*) Structure graph



(*c*) Run-time storage

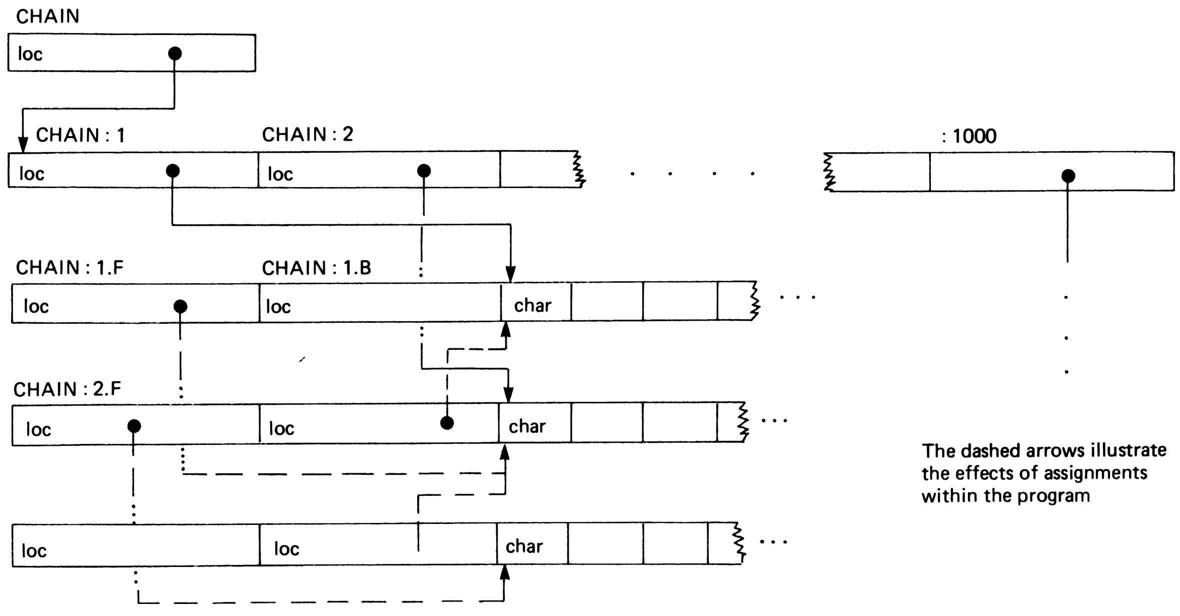


Figure 7. A data chain with forward and backward pointers.

```
X is ref
Y is ref
X pointslike CHAIN:any
Y pointslike CHAIN:any
I is int; J is int
    {comments will be shown in curly brackets}
X = CHAIN:I; Y = CHAIN:J
    {Now we shall unlink X from chain and reinsert it before Y}
X.B.F = X.F; X.F.B = X.B
    {...close gap that will be left by X}
X.B = Y.B; Y.B.F = X
    {...link X to precedessor of Y}
Y.F = Y; Y.B = X
    {...link X to Y}
```

Figure 8. Using a self-referential structure

from the chain and reinserts it before Y. A similar BCPL program is given in Ref. 4. More subversive uses of **ref** are discussed in the next section.

The absence of coercion implies that if X is a variable of scalar type (as distinct from a node of scalar type within a structure) then there is no way to create a reference to X or to alias X legitimately by another node. The proof of this is simply that no **loc** node has been allocated to hold X's address. If the programmer wishes to manipulate references to a single scalar value, he must hold this value from the outset not in a scalar but in a structure – say a vector of length 1. This is no hardship; the restriction here (as compared with Algol-like languages) is a syntactic rather than a practical limitation.

4. COMPILER-TYPING

The language developed here is a *compiler-typed* language, or a language with the *compiler-typing* property, defined as follows:

The type of any legal expression can be determined at compile-time.

For example, after the CHAIN declarations of Fig. 7 the compiler is able to deduce, by tracing through the indirections on the structure graph, that the expression CHAIN:4.F.B.B.F.B:3 is of type **char**. What makes this always possible is the condition given for the completeness of a declaration, namely that all **link** and **ref** nodes of the structure graph have been resolved.

Since the compiler *can* determine types at compile-time we are at liberty to insist that it *must* do so. A compiler satisfying this requirement will be called a *strong* compiler. Such a compiler will determine the offsets and cell sizes needed to evaluate indirections, and will determine which type-dependent code to load for arithmetic operations – in short, the structure graphs are not needed at run-time and can be discarded once compilation is complete.

If a strong compiler is not used, then the structure graphs have to be preserved at run-time and inspected each time it is necessary to determine the type of an expression. This is likely to cause a severe drop in run-time efficiency. (Nevertheless one may sometimes have good reason to prefer such an implementation, for example to do interactive debugging.)

The property of being compiler-typed is so valuable that I shall wish to preserve it when making an extension to the language. The discipline that this imposes on procedure declarations is the subject of the next section.

Two caveats are in order. Firstly, suppose that E **pointsto** (**vec** 1 **thru** 50 **real**) and that I is a variable of type **int**. On encountering the expression E:I the compiler deduces: *if* this expression is legal then it is of type **real**. The legality of the expression cannot be decided until run-time, since only then will it be known whether I lies between 1 and 50.

It is possible, given its access to the structure graph, for the compiler to generate code for run-time range checks whenever a subscripting operation occurs. Novice programmers and their teachers usually desire such an option to be available; experienced programmers and their paymasters usually do not desire this option to be compulsory.

```
9a: legal
    E is link
    E pointsto (I is link, R is link)
    E.I pointsto (vec 1 thru 50 int)
    E.R pointsto (vec 1 thru 50 real)
9b: illegal
    E is link
    E pointsto (vec 1 thru 50 int, vec 51 thru 100 real)
```

Figure 9. Legal and illegal ways of forming a structure with real and integer components

Anyway, at compile-time the best that can be done is to classify the expression as 'legal provided subscript is in range'. As far as type-determination is concerned the expression behaves just like one that is unconditionally legal.

Suppose we wish to have a structure E composed of 50 integer values and 50 real values. There is no difficulty in defining such a structure, as shown in Fig. 9*a*. Suppose instead that the principle of hybrid data structuring had been extended to allow the structure of Fig. 9*b*, so that E:1,...,E:50 are of one type while E:51,...,E:100 are of another. In this case the compiler encountering the expression E:I would not only fail to know whether it was legal but would also fail to determine its type. This illustrates the fact that the compiler-typing property is not trivial, but is sensitive to the exact definition of what data structures are allowed.

The second caveat is to do with reference structures. According to the rule already given, any assignments of the type **ref** = **ref** or **ref** = **link** are legal. This rule deliberately makes no mention of the structures that are pointed to. It is expected that all uses of **ref** will be restricted to experienced programmers. Once self-referential structures are allowed, it is difficult to design a general algorithm for detecting structural compatibility. Therefore it has been left to the programmer, rather than the compiler, to ensure that his particular use of **ref** is appropriate. A second reason for allowing licence in the use of **ref** is to enable storage to be reallocated at run-time, which in some circumstances is a useful technique. The simplest use of the technique just causes two vectors to occupy the same space, regardless of their types, by means of the assignment **ref** = **link** as in Fig. 10. For multilevel structures this goes wrong since the **loc** cells of the new structure will hold incorrect values. The technique becomes more general if address calculations can be done

```
A is link; A pointsto (vec 1 thru 100 int)
B is ref; B pointsto (vec 1 thru 50 real)
B = A
```

Figure 10. Overlaying two vectors

```
R is ref; R pointsto (vec 1 thru m char, LENGTH is int)
    {The value of m is irrelevant. The declaration could be omitted
    if we are willing to let the compiler deduce it from the allocation
    assignment which occurs below.}
M is int
:
    {Assume now that M has been assigned a value, and we wish to
    allocate run-time storage for a string of maximum length M.}
R = alloc (vec 1 thru M char, LENGTH is int)
    {This is the allocation assignment, which the compiler will
    translate into the following sequence of run-time operations:
    STACKPTR = STACKPTR + intsize
    R = STACKPTR
    STACKPTR = STACKPTR + M*charsize
    if (STACKPTR > MEMORYLIMIT) deal with error fi}
```

Figure 11. Run-time allocation of a structure.

explicitly in the programming language, or alternatively if suitable kinds of allocation operators are made available. Fig. 11 illustrates an extension to the language that enables objects to be allocated at run-time on an upward-growing stack. (It is easy to arrange for deallocation to be done automatically on exit from a procedure; or instead of this, for deallocation to be done under the control of the programmer, in so far as this is consistent with the stack discipline. Much more difficult would be the provision of a method for garbage collection.)

5. PROCEDURES AND CLASSES

As usual, a *procedure* is a block of code that acts on zero or more arguments stored outside the procedure itself; a procedure may be either a *subroutine* or a *function*, the difference between these being essentially a matter of syntax.

Suppose a structure is passed as argument to a procedure. We must ensure that the declaration for this structure is equally available to the called and the calling procedure. One way of doing this is to pass a description of the structure as well as its address. This implies that the structure graphs must be preserved and inspected at run-time, and violates the compiler-typing property.

The strategy to be described here preserves compiler-typing as well as having other useful features. In part it resembles the strategy adopted by the language Ada (see Ref. 5).

The first thing to do is to place declarations for the called procedure and its arguments in a part of the program that is within scope of both the called and the calling procedures. Several different kinds of scope rule occur in programming languages, but since the detailed rules are irrelevant here I shall make the following simple-minded assumption: there is a part of the program text, known as a *globunit*, which at compile-time is within scope for all other parts of the program. In the globunit we shall place declarations of the form:

identifier **is subroutine**
identifier **actson** (*arglist*)

or

identifier **is function**
identifier **actson** (*arglist*) **&returns** *returnarg*

Here *arglist* is a list of arguments whose structures have already been declared in the globunit. The *returnarg* of a function is required to be of a scalar type, in line with the restriction that only a scalar may appear on the left-hand side of an assignment.

```
12a: Square-root (SQRT) function
     X is real &noalloc
     SQRT is function
     SQRT actson X &returns X
12b: CHAINSHIFT subroutine
     L is link &noalloc
     L pointsto (F is ref, B is ref)
     L.F pointslike L
     L.B pointslike L
     CHAINSHIFT is subroutine
     CHAINSHIFT actson (L, L)
```

Figure 12. Procedure declarations

```
subroutine CHAINSHIFT (X, Y)
    X.B.F = X.F; X.F.B = X.B
    X.B = Y.B; Y.B.F = X
    X.F = Y; Y.B = X
```

Figure 13. Subroutine CHAINSHIFT.

The details are illustrated in Fig. 12. Note the indicator **&noalloc** appended to argument declarations: this instructs the compiler to build the structure graph but not to allocate any storage.

We can next write the subroutine shown in Fig. 13. Since the globunit is in scope, the arguments do not need to be declared anew within the subroutine body. The compiler will complain if it finds any inconsistency between the body of a procedure and its declaration. The arrangements needed for separate compilation of procedures are discussed in Appendix 1.

The argument-passing method most natural with hybrid data structures is the following. If the argument is of scalar type, including **ref**, then its value is passed to the called procedure, and the possibility altered value resulting from execution of the called procedure is returned to the calling procedure. If the argument is of type **link**, then the value held in the **link** cell is similarly passed; but since it cannot legally be altered does not need to be returned. (This mechanism, for scalar types, can be described unambiguously as 'call by value/result'; for **link** type, the usual terminology becomes ambiguous, and the mechanism could with equal justification be described as 'call by value' or 'call by reference'.)

The next thing to investigate is what compatibility is required between the procedure declaration and a procedure call. (As for the compatibility between scalar arguments certain questions arise, viz. type-conversion and passing of expressions, but since these issues are peripheral in the present context they will be ignored and the discussion will centre on the passing of arguments of type **ref** and **link**.) The simplest and most stringent requirement is to insist that the arguments passed are identical in their structure to the arguments declared. This stringency is unnecessary and a different policy will be found advantageous.

Let C, D be two nodes. I shall say that C is a *refinement* of D, in symbols $C \succ D$, if the following condition holds. (The definition is by induction on the height of the reduced structure tree for D.)

Either **Case 1**
C and D are both scalar types other than **ref**, and the type of C is identical to the type of D.
Or **Case 2**
C is of type **ref** and D of type either **ref** or **link**.
Or **Case 3**
C and D are both of type **link**. In this case we require furthermore:

4-2

If D has vector components, so does C, and these components satisfy

(C:**any**) $\succ$ (D:**any**).

If D has $s > 0$ list components, then C has $r \geqslant s$ list components. The identifiers *id1*, ..., *ids* that name the list components of D are identical to, and occur in the same order as, the identifiers that name the first *s* list components of C. For each list component *id* of D, the relation

(C.*id*) $\succ$ (D.*id*)

holds with the corresponding component of C.

Now let (DEC1, ..., DEC*n*) be the *arglist* occurring in a procedure declaration, and let (CAL1, ..., CAL*n*) be the corresponding list of arguments used in a procedure call. We shall require firstly that the same number of arguments occur in both lists. (A relaxation of this requirement is possible but will not be discussed here.) The remaining condition for legality of the call is that each call argument CAL*k* is a refinement of the corresponding declared argument DEC*k*. Compiler typing is preserved because the refinement relation can be checked at compile-time; therefore the compiler can issue warnings about erroneous procedure calls.

The formal requirements stated above can be understood as follows. For CAL to be a refinement of DEC it is necessary that CAL contain the reduced structure tree of DEC as a substructure. If CAL is passed as an argument to a procedure, the procedure may well have been declared to use only part of this structure, namely the substructure DEC. The refinement relation CALL $\succ$ DEC is necessary and sufficient for the correct interpretation by the procedure of the substructure that it uses – subject to the usual licences regarding subscript ranges and **ref**. These licences, analagous to those discussed in section 4, arise as follows. Case 2 requires no compatibility between the structures pointed to. Case 3 requires no compatibility between subscript ranges. Therefore the programmer, who is assumed to know what he is doing, can write a subroutine that operates on a vector of unknown size, the actual size being determined by other arguments passed (or by other components of the same argument). Note that a **link** value must not be passed to a procedure that uses it as a **ref**, since a **ref** value may be altered while a **link** value may not. This has been taken care of in the definition of the refinement relation.

Now BUFFER declared in Fig. 3 is a refinement of STRING declared in Fig. 2. We may have a procedure COPY whose declared arguments are (STRING, STRING); then we may call COPY (STRING, BUFFER), and this call will operate on the 'string' substructure within BUFFER while ignoring the other bits and pieces that are attached to it.

If X is any structure, let the *dataclass* of X be the class of all structures that are refinements of X. Procedures can be considered to act on dataclasses rather than on structure types. For example CHAINSHIFT (declared in Fig. 12 and defined in Fig. 13) operates on the dataclass L (declared in Fig. 12) which is the class of 'abstract' chain elements with forward and backward pointers. We may issue the calling statement CHAINSHIFT (CHAIN: I, CHAIN:J) where CHAIN:I and CHAIN:J as in Fig. 8 are 'concrete' chain elements that contain data as well as pointers. The subroutine shifts the pointers while ignoring the data, just as it should do.

I shall define a *procedural class* to consist of a dataclass, or collection of dataclasses, together with a collection of procedures that act on these dataclasses. The declarations present in a globunit reflect just such an organization. Indeed, if we are now allowed to split our globunit into several globunits, each containing the argument and procedure declarations appropriate to a particular procedural class, then the notion of procedural class is reflected in the language exactly. This is the sense in which hybrid data structures lead naturally to a class concept.

One deficiency in this concept should be noted. The refinement relation $\succ$ is not a lattice ordering. If X and Y are two structures, it is in general not possible to find a structure Z that is a refinement of both X and Y.

I shall comment now on what happens if we wish to pass a procedure PCAL as argument to another procedure Q. Suppose PDEC is the procedure that occurs as the corresponding argument in the declaration for Q. Suppose PCAL is declared with arguments (CAL1, ..., CAL*n*) and PDEC with arguments (DEC1, ..., DEC*n*). The condition for legality of the call Q(..., PCAL, ...), as far as this argument is concerned, will still be written as PCAL $\succ$ PDEC; this holds if and only if DEC1 $\succ$ CAL1, ..., DEC*n* $\succ$ CAL*n*. The reversal of the refinement relation that occurs here should be carefully noted. I do not give the full proof of this result, but the method should be clear from Appendix 2, which demonstrates the result in a particular case. (Behind this phenomenon of refinement reversal can be discerned the fact that procedures and data structures are not 'similar' objects but are 'dual' objects in a certain mathematical sense.) Similar considerations are needed if one wishes to extend the language to include procedural variables, i.e. variables whose values are procedures.

One final point about procedures. The language described here is basically one in which storage requirements are determined at compile-time. Nevertheless, it is easy to arrange a stack discipline for the allocation of procedural data frames. Therefore procedures could be used recursively, and data storage would then be dynamic in so far as it is controlled by procedure calls.

6. CONCLUSIONS

This paper describes how hybrid data structures are defined by indirection and gives examples of their use. Hybrid data structures are implemented by the use of structure graphs, which are built by the compiler and discarded at run-time. This allows indirection operators to be defined which are appropriate to a type-rich language.

The mechanism can be extended to deal straightforwardly with procedures and procedural classes.

A elegant and practical syntax can be developed; the actual syntax could be more concise than that used here. The concept appears well suited to a small but powerful general-purpose language.

APPENDIX 1

Separate compilation of procedures

Suppose *file1* contains the source text for a main program and *file2* contains the source text for some procedures that will be called from *file1*. The programmer must place copies of the relevant globunits in both files. Each file is compiled, and the checking of passed against declared arguments is done in the usual way.

It is not until the files are brought together that any check can be made that the globunit declarations in the two files are in fact consistent with one another. It is desirable to make this check – particularly since the files may have been written by different people – otherwise annoying errors will occur. Now loaders are commonly designed to be independent of the language from which the compiled code was produced: this is a right and proper modularization. Therefore it is not reasonable to expect the loader to check the consistency of globunit declarations.

So we need to introduce a language-dependent 'vetting' step in the sequence compile–vet–load which performs this check. The structure graphs produced by the compiler are used by the vet step, and are then thrown away.

APPENDIX 2

Demonstration of the reversed refinement relation for procedural arguments

The proof is given here for the particular case of a subroutine APPLY (S, X) in which S is a subroutine and X is a data structure.

Given the following declarations:

```
SA is (data structure)
TA is (data structure)
X is (data structure)
S is subroutine
S actson (SA)
T is subroutine
T actson (TA)
APPLY is subroutine
APPLY actson (S, X)
```

and the following subroutine definition:

```
subroutine APPLY (S, X)
  S(X)
end
```

and the following call in the main program:

```
APPLY (T, Y)
```

then the situation is as follows.

Firstly, the necessary and sufficient condition for APPLY to compile successfully is that the call S(X) is valid, i.e. $X \succ SA$.

Now we seek the condition denoted by $T \succ S$ which is the weakest condition such that:

APPLY (T, Y) is a valid call whenever ($T \succ S$ and $Y \succ X$ and APPLY has compiled successfully).

Equivalently:

T(Y) is valid whenever ($T \succ S$ and $Y \succ X$ and $X \succ SA$);

i.e. ($T \succ S$ and $Y \succ SA$) implies $Y \succ TA$;
i.e. $T \succ S$ implies ($Y \succ SA$ implies $Y \succ TA$);
i.e. $T \succ S$ implies $SA \succ TA$;
and since the weakest condition is sought, this gives

$T \succ S$ if and only if $SA \succ TA$

completing the demonstration.

The general proof needs to deal with procedures having arbitrarily many arguments. However, since the present demonstration has established the necessity of $SA \succ TA$ in one case, it remains only to show that $SA \succ TA$ is sufficient in every case.

Acknowledgement

The work described in this paper was carried out while the author was employed in the Operational Research Executive of the National Coal Board. The author is grateful to the National Coal Board for permission to publish the paper.

REFERENCES

1. M. Richards and C. Whitby-Stevens, *BCPL – the Language and its Compiler*. Cambridge University Press, Cambridge (1979).
2. R. Bornat, *Understanding and writing Compilers*. Macmillan, London (1979).
3. D. W. Barron, *An Introduction to the Study of Programming Languages*. Cambridge University Press, Cambridge (1977).
4. M. Richards, Programming structure, style, and efficiency. *Infotech State of the Art Report on Structured Programming*, Pergamon, Oxford (1976), pp. 337–345.
5. J. G. P. Barnes, *Programming in Ada*. Addison-Wesley, London (1982).