PROGRAMMING WITHOUT POINTER VARIABLES*

by

Richard B. Kieburtz
Dept. of Computer Science
State University
of New York at Stony Brook

ABSTRACT

The presence of pointer variables in high level programming languages constitutes an artifact originally introduced to support the representation of recursive data structures. Programming practice has come to rely on pointer variables for their originally intended use, and for several others as well. Their use adds to the complexity of stating algorithms by forcing one to conceptualize data representations in which storage addressing is made manifest. In addition, the use of pointer variables allows one to refer to a common data object by a multiplicity of names, a phenomenon we call the alias variable problem. Alias variables make the verification of program behavior substanially more difficult, and frustrate the goals of modular decomposition.

In this paper, we consider alternatives to the use of pointer variables in programming. One of these alternatives is the inclusion of a class of data abstractions based on recursively defined data types. The representational power of the proposed data types is characterized formally, and some issues of implementation and efficiency of programs are discussed.

1. INTRODUCTION

Pointer, or reference variables have recently come under criticism [1] that parallels the criticism of the GOTO statement by Dijkstra [2]. Just as the unrestricted GOTO broadens the context from which a labeled statement can be executed, so the unrestricted pointer makes global the context from which a data object can be accessed. In fact, pointer variables offer more opportunities for magic and mischief than do GOTO's, as nearly any PL/1 programmer can tell you. Some of the worst abuses of unrestricted pointer variables can be avoided by binding each pointer variable to a specific data type when the pointer is declared. Two languages that employ this kind of restriction are PASCAL [3] and its predecessor, ALGOL W [5]. When strictly observed, this restriction prevents occurrences of the so-called "hanging pointer" (one that refers to an object that no longer exists, because of explicit deallocation or the end of its defined lifetime).

*This work was partially supported by the National Science Foundation under Grant No. JO 42203.

Even with restrictions to prevent hanging pointers, programming data structures with pointer variables still gives trouble to experienced programmers. To convince oneself, one need only try the exercise of interchanging a pair of successive elements in a one-way list linked by pointer variables. The operation can certainly be done, yet the program that does it is not nearly as intuitive as one that will exchange values of successive elements of an array, for instance.

Upon looking at the uses made of pointer variables in programs, four principal classes of use can be identified.

i) Arrays of pointer variables are used to index instances of structured types. These arrays are commonly referred to as "inverted files".

ii) Pointer variables are used to link together instances of structured types to form dynamic lists.

iii) Relations among data objects are often represented by the arcs of a data graph. A direct realization of a directed graph can be obtained if the nodes are defined to correspond to data values, and the ares are represented by pointer variables.

iv) One sometimes wishes to share information among a group of program modules or processes, without producing a fresh copy of the shared data base for each customer module. Access to a shared data base is commonly provided by passing an address constant, the value of a pointer variable, to each module.

In each of these classes of use, it is apparent that the values held by pointer variables, the bit strings that specify physical addresses to the storage unit of some actual computer, are not of primary interest to the applications programmer. Pointer variables are only a means to an end, and by employing them, the programmer is concerning himself with manipulation of the data representation in a machine. Pointer variables offer operations that are meaningful only at a very low level of conceptual abstraction, yet remain in the consciousness of the applications programmer even at the highest level of program implementation. We believe that the fundamental sin in the use of pointers is to make variables out of them, that is, to make them assignable. At the applications level, after all, the data structures programmer is concerned with access to data, not with its location in storage. Access should be definable, but it need not be assignable.

In the remainder of this paper, we shall address the problem of defining data types without the use of pointer variables, and shall argue by example as well as rhetoric that each of the applications mentioned above can be met without employing pointer variables. Finally, we shall show how computations involving address constants as data can be pushed down to a very low level of abstraction, and thereby standardized.

2. DEFINING COMPOUND DATA TYPES

The definition of data types provides a framework for the interpretation of abstract objects that can be described within a programming language. Defining types is part of the job of defining a typed language, and is the responsibility of the language designer. Declaration, on the other hand, is the means by which a programmer can define bindings of identifiers to particular data types, and can specify instances of certain compound types.

Since the name "data type" has been attached to several distinct notions by different people, we have an obligation to clarify what is the notion of type that will be used in this paper. We use data type to mean a formal notation to distinguish sets of values, and classes of mappings on those values. Data types are therefore defined in terms of sets, and to any definable data type there must correspond a set of values, or mappings, though these values may be compound, or structured, as well as elementary. A related notion, to which we shall refer later in this paper, is that of a _data abstraction_, which includes not only a data type, but a set of operations defined on that type. Data abstractions can therefore be formally defined in terms of algebras. Noteworthy examples of the incorporation of data abstractions into programming languages are to be found in SIMULA 67 [6], CLU [8], and ALPHARD [10].

The rules by which data types can be formed include:

i) _enumeration of values_. Any typed programming language must include one or more simple types, each of which is an enumerable set of scalar values. There may, in fact, be infinitely many simple types in the language, as is the case in PASCAL, in which any finite set of scalars declared by a programmer is a simple type. However, all such sets have been anticipated by the language designer, and hence all are defined in the programming language. In addition, it is convenient to define as a distinguished simple type a set that we shall designate by the symbol Ω, and which we call _undefined_. In some programming languages, this set contains arbitrarily many values, and it is not testable whether a value is a member of this set or of some other simple type. In other languages, the undefined type contains a single unique value, called Null.

ii) _composition of types_. Each of the rules that follows gives a form by which new types are composed from previously defined types.

iii) _cartesian product_. Any finite sequence of types can be named as a compound type. Values of the compound type are vectors whose elements take values from the type corresponding to the position of each element in the vector.

iv) _iterative replication_. This operation, analogous to the Kleene star operation in formal language theory, enables finite (but unbounded) vectors of values from a single type to constitute a compound type.

v) _recursive definition_. The ability to give names to types is assumed. A type definition in which the name of the type being defined appears as part of the definition is said to be recursively defined. This capability is required in order to be able to define as types lists, trees, and other graph-like objects of indeterminate extent.

vi) _mappings on types_. A compound type may be defined as a mapping from one type to another. Operator types are necessarily defined by mappings, but so are some data types such as arrays.

vii) _union of types_. If A and B are types, then $A \cup B$ can be defined to be a type. Since union types tend to erase the distinctions that are made by employing data types in the first place, unions of simple types are not used commonly. However, many operators commonly employed in programming languages are described by unions of operator types. Union types are also said to be _polymorphic_.

One other example of a union type deserves special mention. If t is any type, then the union $\S t = t \cup \Omega$, is the type we shall call _variable_ t. It will be the type we shall associate with program variables declared to be of type t.

viii) _reference types_. If t is any type, then _ref_ t is a type whose values are references to objects of type t. This is the type of pointer values, whereas pointer variables are of some type §_ref_ t. If pointers are not bound to some designated type by a declaration, then the target type t is the union of all defined types of the programming language!

The type formation rules given above deal with uninterpreted types; that is, the application of these formation rules is independent of the values actually held by specific objects of any type. This has several implications. One is that if types of identifiers are fully declared in a program, all type checking can be done by a compiler, since type correspondence is data-independent by these rules. Second, if the sets of values constituting the several simple types of a programming language are mutually disjoint, then any constant definable in the language belongs to a unique type.

However, since types defined by these rules are value-independent, subranges of enumeration types are not described as distinct types. Also, field identifiers that may be associated with a structured type, are not considered to be a part of that type. These restrictions should not be considered to be a deficiency of an uninterpreted type theory, but merely a demarcation between the notion of a data type, and the very useful extension to the notion of a data abstraction. This distinction, unforutnately, has not yet been made by many programming language designers.

It would be remiss not to mention an earlier exposition of type formation rules by Hoare [11] that strongly influenced the design of the programming language PASCAL. The rules given here differ in degree from those given there, particularly in being fully value-independent, and in distinguishing variable types. However, the author must acknowledge having spent considerable time in reflection upon the consequences of Hoare's type definitions.

In order to provide for the realization of type definitions, the language designer will find it necessary to define some standard procedures for the implementation of data accesses, assignment of values, and transformations on dynamically allocated typed objects. The applications programmer should never need to define procedures to access data types. We can state two seemingly fundamental principles concerning access mechanisms.

Principle of Encapsulation: For a variable of any data type that can be declared in a programming language, there shall be a standard access mechanism defined in the language.

Principle of Unique Reference: Each variable within a definitional context shall have a unique form of reference within that context.

A programming language that violates the Principle of Encapsulation might be said to be extensible, in the sense that it leaves to the user the task of specifying access mechanism. It is worthwhile to note that the principle of encapsulation has been adopted for data abstractions as well, as they have been incorporated into some of the newer, experimental programming languages [8,9,10].

Nearly every programming language in common use violates the Principle of Unique Reference; nevertheless, it is both desirable and achievable. The means of violation are: declared intent to violate by storage equivalencing, binding of multiple names as a consequence of a procedure call in which parameters are passed by name or reference, and assignment of a reference to more than one pointer variable. We refer to violations as alias variables, for they create alias names for the same object. We shall concentrate in this paper on eliminating pointer assignment as a means of aliasing.

3. RECURSIVE DATA TYPES

Just as it only became apparent that one could write programs without GOTO statements after some powerful control primitives had been incorporated into programming languages, one is not likely to convince programmers of data structures to give up the use of pointer variables until more powerful data types have been provided. In a recent paper [12] Hoare has made a clear distinction between the definition of a cell of a linked data structure, from which extensible linked structures can be programmed by cell allocation and pointer assignment, and the recursive definition of a linked data structure, for which extensibility is implicit, and the link pointers are managed by the system. Although Languages such as LISP that employ implicitly recursive structures have been available as long or longer than those that employ pointer variables, the advantage of using recursive definition to avoid the needless complexity of explicitly programming data structure maintenance seems not to have been fully appreciated by programming language designers over the past ten years.

We shall employ a slightly more elementary notation to define recursive data types than does Hoare, because we wish to stress the distinctions between pointer values, which must occur in linked structures but which need not be visible, and pointer variables which are visible and manipulable by a programmer. Hoare includes in his definition of a recrusive data type the names of type generators, which by the distinction we have drawn earlier, are a part of the definition of a data abstraction. The notation that follows is for the sake of exposition; we do not advocate it as preferable to any other notation for incorporation into an applications programming language.

Let us introduce the mapping type

cond (t) = Boolean → t

where t is any type that is not itself a program variable type. The interpretation given to cond is that it yields one or the other of a pair of objects of type t, depending on the evaluation of its Boolean argument. In a programming language supporting this type, one might declare a variable

Cond_C : cond (§char)

References to this variable could take the form Cond_C[true] or Cond_C[false], referring to each of its components, or Cond_C[X>0], referring to a component dependent on the value of the Boolean expression. This type is realized in PASCAL as a special case of the array type.

Next, let us define a pair of recursive types needed to allow the realization of dynamic lists, and the representation of relations on data in the form of data graphs. These are:

list (t) = (t, §list (t)), and

tree (t) = (cond (§tree(t))).

in which t is any defined type. The type formation rules employed in these definitions are composition, cartesian product (since the right side of each definition is an ordered pair of types), and recursive definition. The type §list (t) is not a pointer type, but takes values from the union $\Omega \cup$ list (t). Formally, the type is defined to be the least fixed-point of the defining equation.

In order to make these types usable as data abstractions, we must define a class of standard functions and operators on each type. It will be notationally convenient to use curly brackets to indicate selection of the first component of an object of either type, {}: (x, y) → x . To indicate selection of a second component of a list, we shall use a postfix operator

tail : (x, y) → t

Thus, given a definition of a list variable by

L : list (§char)

we specify by {L} the variable of type §char at the head of the list, and by {L.tail} the second §char variable. For convenience, multiple applications of tail can be indicated by appending a parenthesized positive integer constant; L.tail(2) is abbreviated notation for L.tail.tail.

To select one of the pair of objects that constitute the second element of a tree, we use as a postfix operator a Boolean expression enclosed in square brackets. As notational shorthand to avoid nested brackets, a list of Boolean expressions separated by commas will denote successive application, For example, declaring

T : tree (§char)

we could designate an object two levels deep in the tree by {T.[false,false]}.

A decision that cannot be avoided is how one can specify in an iterative program the traversal of a recursively defined data structure. In fact, one of the principal reasons for retaining the pointer variable in a programming language that supports iterative control in preference to recursion, is that pointer assignment within an iterated statement permits one to specify traversal of a linked data structure. Without this capability, one must resort to a recursive procedure to which a substructure is passed as an argument, and this seems to be an unreasonable price to pay for the elimination of pointer variables. However, there is an alternative short of revising our definition of recursive types, and it is the definition of a restricted assignment operator.

When a variable of a recursive type is declared, the name of the variable is bound to an object of that type, which is recursively defined data structure. We call the declared name of such a variable a selector name. It is possible, without creating aliases, to allow that same name to take a variable binding so that it can indicate any substructure of that same object, since no other variable name can refer to the object. Exactly how this is to be accomplished is up to the implementor, but logically, such a variable binding does not require merely the introduction of a ref type; it can be accomplished merely by defining a restricted assignment operator, selects. A statement such as

L selects L.tail

is defined to bind a selector variable to a selector expression that refers to a component of the same recursively defined object, by the semantics of textual substitution. This variable selector binding is all that is needed to allow structure traversal by an iterative program.

Since a selector name, once bound to a selector expression that identifies a proper sublist or subtree can never access the head of the list or the root of the tree by any further bindings to selector expressions, another standard procedure, reset is defined to restore the original binding.

The value of a selector expression is the list or tree to which it refers. It may also have an undefined value. The condition of being defined or undefined is presumed to be testable for any variable type, by a Boolean-valued function

Defined : §t → Boolean.

To complete the description of the data abstractions employing types list and tree, one must specify standard procedures for the allocation and release of elements. These are:

Extend (X, L) : (t, §list (t)) → §list (t)

This operator is similar to the cons operator of LISP, appending to the list designated by L a new initial element with value X. However, Extend is a procedure, not a function, and gives a new binding to the list selector ecpression passed as its second argument, so that it subsequently refers to the extended list. If L happens to refer to a proper sublist of a host list when Extend is invoked, then it will have the effect of inserting a new element with value X into the host, at the point selected by L.

Deletehead (L) : §list (t) → §list (t)

has the effect of shortening the list selected by L, deallocating its initial element and modifying the binding of the selector expression that is its argument. It can be used to delete an element from the interior of a host list by passing it the selector of the element to be deleted.

Newleaf (X, T) : (t, §tree (t)) → §tree (t)

This is the allocation procedure for type tree (t). Its second argument is a tree selector which must have an undefined value. A successful invocation of Newleaf adds a new node with value X to the tree at the branch designated by the selector expression, leaving the binding of the selector expression unchanged, although its value becomes defined.

Deletetree (t) : §tree (t) → §tree (t)

deallocates the entire tree or subtree that is the value of its selector argument, leaving the selector with an undefined value.

4. OVERLOADING THE ASSIGNMENT OPERATOR

When the definition of an elementary operator symbol is extended to include some compound operation on structured data types, this practice is called overloading. In several programming languages, the assignment operator is overloaded to allow it to specify assignment of arrays of comparable dimension. Given array variables

```
A, B : array [1..N] of §char
```

then the meaning of the assignment A := B is defined to be

```
A[i] := B[i]
```

where i is a pseudo-variable.

In the definitions of the recursive types list and tree we have gone out of our way to avoid the introduction of pointer variables, so that when a variable of type §list or §tree is the target of an assignment, we do not mean that the effect shall be pointer assignment. Instead, assignment to a variable of a recursive type is an overloading of the assigmment operator, implying that a fresh copy of the value on the right side of the assignment is to be made. Given two variables of recursive types,

```
L, Z : §list of §char
```

the assignment L := Z means

```
while Defined(Z) do
  begin
    if Defined(L) then
      {L}  := {Z}
    else
      Extend (L, {Z});
    Z selects Z.tail;
    L selects L.tail
  end;
while Defined(L) do
  Deletehead(L);
reset Z; reset L
```

The semantics of overloading assignment for type tree will be a little more complex, since traversal of a tree is required. There are no visible pointers that can be stored on a stack to provide a record of the path followed in reaching a particular node from the root. However, if we want to describe overloading in terms of an iterative algorithm, the chain of Boolean values that have been used to select branches in descending through the tree could be stacked instead.

As an alternative, we shall describe the meaning of assignment for trees by a recursive algorithm, in which the overloaded assignment operator is applied to subtrees. Given tree variables

```
T, Y : §tree (§char)
```

the semantics of assignment is given by

```
if Defined(Y) then
  begin
    if Defined(T) then
      {T}  := {Y}
    else
      Newleaf (t, {Y});
    for b := false to true do
      T.[b] := Y.[b]
  end
else Deletetree(T)
```

In the algorithm description above, false and true are constants of Boolean type, and therefore ordered, as defined in PASCAL.

The definitions of assignment overloading have been given in terms of algorithms to implement copying of values that are data structures. Since copying can be an expensive operation if the structures are large, one may be inclined to ask whether copying is always necessary, in order to implement the defined semantics. For instance, if one wished to exchange the values of the two branches of a node of a tree, performing the exchange by assignment to an intermediate temporary will require three tree values to be copied,

```
Temp_tree := T. [false];
T.[false] := T.[true];
T.[true] := Temp_tree
```

whereas had the tree been implemented by means of nodes containing pointer variables that serve as the links to branches of the tree, the exchange could have been performed by three pointer assignments, using exactly the same form for the assignment statements.

Some relief can be obtained by additional overloading of the assignment operator. The notion of simultaneous, multiple assignment in a single statement has been around for some time; it was included in the experimental programming language CPL [13]. The effect of simultaneous assignment is as if all expressions in the list on the right side of the operator are evaluated, and the addresses of all the variables on the left side are obtained, prior to performing the replacement of values. Thus, for instance, the two assignment statements

```
A [I], I := I, J.
```

and

```
I, A[I] := J, I
```

produce identical results, the same result as executing in sequence

```
A[I] := I; I := J
```

In particular, simultaneous multiple assignment allows one to perform an exchange of values by means of a single assignment,

```
I, J := J, I
```

The main objection voiced to the use of simultaneous assignment is the potential for ambiguity in the case of alias variables. For instance, the assignment,

```
A[I], A[J] := X, Y
```

is ambiguous in case the index variables I and J happen to take the same value. However, recalling that our original motivation for eliminating pointer variables was to eliminate the occurrence of alias variables, we may waive the objection to its use.

Returning to the example of exchanging values of two branches of a tree, suppose the exchange were programmed by a single statement,

```
T.[false], T.[true] := T.[true], T.[false]
```

A compiler with even the most limited capability for data flow analysis could determine that the variables named on the right side of this assignment each had their values redefined by the statement. Therefore, although the semantics of tree assignment dictate that a value must be copied, rather than to bind an additional name to the value, it would be impossible in this case to detect whether new copies were made of each tree, or whether the original copies were merely renamed. An implementor in search of efficiency would capitalize on such an opportunity, by performing the value exchange in the most efficient way consistent with the observable effect of the semantic definition.

Because each declared variable of a recursive type has only a single selector identifier, it is usually possible to determine from the program text when one selector expression designates a substructure of another. In such cases, as in the assignment

```
L, L.tail := L.tail, L
```

a correct implementation will require a certain amount of recopying, which is easily detectable by a compiler. However, there are some cases in which aliasing cannot be detected at compile time, as in the two tree selector expressions,

```
T.[F(x) = 0], T.[F(y) = 0]
```

In such cases, a runtime check for aliasing may be required, in order to determine what is a correct implementation.

5. PROGRAMMING WITHOUT POINTER VARIABLES

In this section, we shall attempt to convince the programmer of the advantages of recursive types over the use of structures linked by pointer variables, by considering some examples. This is risky business, because arguments for the introduction of new structures into programming languages have traditionally been made on the basis of demonstrating the power of the addition on some conjured-up example, without giving due consideration to other means of achieving an equivalent algorithm. The consequence can be the cluttering of a programming language with competing mechanisms that will not be understood in their entirety by any programmer.

However, what is proposed here is not simply the addition of recursive data types to a programming language, but their substitution for pointer variables as a means of managing and accessing dynamically allocatable data structures.

Our criticism of pointer variables has been that their use forces the programmer to conceptualize the use of a data type that has no direct correspondence to the data of his application, and which permits violation of the Principle of Unique Reference. In general, the use of pointer variables makes program verification more difficult, no matter what means of verification is employed.

The first example we shall consider is search in a binary tree, to determine whether or not a given value is present. If the tree were implemented in a language relying on pointer variables, a global declaration of a structured type to represent nodes of the tree is required:

```
type Tr = record (Val : §char;
                  Upper, Lower : pointer(Tr))
```

A recursive function for binary search in a tree built from nodes of type Tr is

```
function Tr_search (A : char; T : pointer (Tr))
       : Boolean;

  begin

    if T = Null then Tr_search := false
```

```
    else if T↑. Val = A then TR_search := true

    else if T↑. Val > A then
      Tr_search := Tr_search (A, T↑. Lower)

    else Tr_search := Tr_search (A, T↑. Upper)

end
```

Alternatively, to search a tree defined as a recursive type, one can use the function

```
function R_search (A : char; T : tree(char))
        : Boolean;

  begin

    if ¬ Defined(T) then R_search := false

    else if {T} = A then R_search := true

    else R_search := R_search (A, T.[{T} < A])

  end
```

The two functions are similar in structure; in fact, had the record type Tr used in the first function made use of the facility of PASCAL to define a pointer array indexed by Boolean values instead of the more conventional notation of separately named pointer variables, the two procedures could have been made identical. However, the function R_search accesses no global variables and requires only read access to the values passed through its argument list. This has been made explicit in the declarations of the formal parameters by not designating the parameter types as variable types. The function can be recognized by a compiler as one having no side effects. TR_search either requires global access to the structure composed of records Tr, as it has been written, or alternatively access to Tr might have been provided via a parameter passed by name. A compiler cannot verify that Tr_search is incapable of producing side effects unless it examines the code of its body.

A second difference is that Tr_search cannot be proved to terminate unless one is given as a hypothesis that the data structure formed by the nodes of Tr satisfies a partial ordering. Since the data type tree has this property by definition, a proof that R_search always terminates is trivial.

As a second example, consider the problem of reordering elements of a one-way linked list. With pointer variables, a list is formed of elements of the type

```
type Lst = (C : §char; Next : §pointer(Lst))
```

and a procedure to interchange elements if the value field of the first is greater in order than the value field of the second can be given as:

```
procedure Re_order (var Predecessor
                          : §pointer(Lst));

  var Placeholder : §pointer(Lst)

  begin

    if Predecessor ¬= Null
      and Predecessor.Next ¬= Null

      and Predecessor.C > Predecessor↑.
          Next↑.C then

  begin

    Placeholder := Predecessor;

    Predecessor := Predecessor↑.Next;

    Placeholder↑.Next := Predecessor↑.Next;

    Predecessor↑.Next := Placeholder

  end
```

It is very difficult to program this procedure or even to understand it without drawing a succession of diagrams tracing out the steps it executes. The actual parameter must be passed by name or reference, and furthermore it must be a link pointer of the structure in order for the procedure to succeed. If the actual parameter is not a link pointer of the structure, but is merely an external pointer into it, then the procedure will have the effect of removing an element from the structure, leaving the actual parameter as the only pointer (apart from other possible aliases) to the element removed. It is impossible to verify the correctness of this procedure without examining the circumstances of each call.

Contrast this with the following statement that accomplishes the reorder operation on a variable of type §list,

```
if Defined(L.tail) and {L} > {L.tail} then

   {L}, {L.tail} := {L.tail},{L}
```

Although the semantic definition of assignment for type §list is given in terms of a copy rule, the compiler is not obliged to implement copying of values to perform an interchange correctly, as was discussed in the preceding section. It may, depending on the implementation, generate exactly the same steps that were prescribed in the procedure Re_order, but it will not be the applications programmer's responsibility to work them out correctly. If the statement given above was embedded as a procedure body, and L became a formal parameter passed by name or reference, the correctness of the operation could still be verified independently of the circumstances of its call, since there can be no external pointers into an object of a list type.

As a third example, consider accessing an entry in a two-level hierarchical directory, to permit reading, updating, or replacement of a directory entry. In a language utilizing pointer variables, such a procedure would eventually return a pointer to the desired entry, if it were found. The pointer value can be stored, increasing the number of ways in which a component of the directory can be referenced and prolonging the lifetime of a single interrogation. This can lead to inconsistencies if the directory contents are subsequently altered, A directory access procedure, using a directory defined by means of the recursive type list can be allowed to return a value of the list type of a portain of the directory itself. In this example, we have followed custom in blurring the distinction between type and data abstraction by including in the statement of type definition, the names of field selector functions. The access procedure itself should also properly be defined as a member of the data abstraction associated with the directory. We shall suggest a notation and semantics to accomplish this in section 7. For the present example, the directory type is

```
type Directory = list(C : §char; Subcat
             : §list(§char))
```

That is, a directory is defined as a list of lists of char type variables. Each of the second level lists is identified by a tag which is itself a char variable. The access procedure returns a variable of a list type, rather than a value.

```
var function Lookup (Catalog : Directory;
                     Name1, Name2 : char)

                   : list(§char);

  begin

    while Defined(Catalog)
          and {Catalog}.C = Name1 do

       Catalog selects Catalog.tail

    if Defined(Catalog) then with {Catalog} do

       begin

          while Defined(Subcat)
                and {Subcat} = Name2 do

             Subcat selects Subcat.tail

          Lookup == Subcat

       end

  end
```

The designation of the final assignment by the operator symbol == is to denote explicit aliasing. As mentioned previously, we shall later improve our notation to eliminate the need for this.

The function returns a variable whose value is a part of the directory and may therefore be assigned (by the semantics of the overloaded assignment operator) tested, or modified. The result of the function call is in fact a temporary selector name for a portion of the directory, thus the function is properly a part of the data abstraction defining the directory. Because of the semantics given for list assignment, the variable cannot be preserved as a lingering reference to a portion of the directory; its contents can be copied by assignment to an external variable and stored, as can any value. This is in fact the operation that is frequently done when a portion of an off-line directory is to be processed by a program -- the desired portion is first copied into primary storage.

6. REPRESENTING RELATIONS ON DATA

In the preceding section we have shown by example that the recursive types list and tree permit certain list-processing algorithms to be stated more concisely and elegantly than would be possible if pointer variables were used. While such examples provide adequate reason to add new features to an extensible, application-oriented language, they do not justify the inclusion of a new control or data abstraction in a fully defined, general purpose programming language. A control abstraction is justified if it meets one or two criteria:

i) if its addition gives some definable subset of the programming language the power to compute the recursive functions, and that subset is considered a desirable programming language in light of other criteria that have guided the design of the host language, or

ii) if it makes explicit a control clause that can be stated awkwardly without its use, and that control clause is considered desirable by other criteria.

On the other hand, we have much less objective criteria by which to judge the necessity and desirability of proposed new data abstractions. The following criteria are hereby proposed:

I) a data type is justified if its addition to a programming language gives some definable subset of the language the power to represent recursive relations, and the addition of the type does not promote violations of Principles of Encapsulation or of Unique Reference;

II) a data abstraction is justified if it makes explicit a data access that can be done procedurally or by means of an encoding without its use, and the data access method is considered desirable by other criteria.

The first criterion is one of sufficiency, the second is one of desirability. Having stated criterion (I), we are now faced with the need to state, for at least one class of data structures, what it means to represent a recursive relation.

The class of data structures of interest here will be data graphs. Now it is certainly possible to obtain representations of recursive relations on other structures than data graphs, but the subset of a programming language that we have in mind is one that supports representation of relations on data graphs.

It is necessary to give some definitions. Let $G = \langle G, A \rangle$ be a directed graph, where G denotes a set of nodes (a recursively enumerable set) and $A \subseteq G^2$ a set of arcs. Denote by $\overset{*}{\rightarrow}$ the relation of reachability in G, reading $a \overset{*}{\rightarrow} b$ as 'b is reachable from a'. The reachability relation is the smallest set of ordered pairs from G^2 such that

i) $(\forall\, a \in G)\quad a \overset{*}{\rightarrow} a$, and

ii) $(\forall\, a,b,c \in G)\quad$ if $a \overset{*}{\rightarrow} b$ and $(b,c) \in A$ then $a \overset{*}{\rightarrow} c$.

Let D be a recursively enumerable set, and let R be a binary relation over D, $R \subseteq D^2$. Let ψ be a mapping from G to D. The set of ordered pairs $\{(\psi a, \psi b) \mid a *\!\rightarrow b\}$ is called the image of $*\!\rightarrow$ under the mapping ψ, and is denoted by $\psi(*\!\rightarrow)$. $\psi(*\!\rightarrow) \subseteq D^2$, and hence is a binary relation over D. If $(\forall a,b \in G)\ a *\!\rightarrow b \Longleftrightarrow \psi a\ R\ \psi b$, then ψ is said to be a <u>relational</u> <u>morphism</u>, and $\psi(*\!\rightarrow) \subseteq R$.

<u>Definition 1</u>: If $G = \langle G,A \rangle$ is a directed graph, $R \subseteq D^2$ is a binary recursive relation, ψ: $G \rightarrow D$ is a relational monomorphism (one-to-one) and if $\psi(*\!\rightarrow)$ is a cover for R, that is $R \subseteq \psi(*\!\rightarrow)$, then G is said to be an <u>exact</u> <u>representation</u> of R. ∎

<u>Definition 2</u>: Given the conditions of Definition 1 except that ψ is a relational homomorphism (many-to-one), then G is said to be a <u>direct representation</u> of R. ∎

Obviously, $\psi(*\!\rightarrow)$ is equal to R under conditions of exact or direct representation. The next task is to characterize the classes of relations that are exactly or directly representable by various classes of directly representable by various classes of directed graphs. The results are stated as theorems, but proofs (which are elementary) have been omitted for the sake of brevity.

<u>Theorem 1</u>:

a) R is reflexive and transitive $\Longleftrightarrow$ $\exists$G such that G exactly represents R;

b) R is reflexive, transitive and antisymmetric (hence, an ordering relation) $\Longleftrightarrow$ $\exists$G such that G is acyclic and G exactly represents R;

c) R is a lattice order with g.l.b. $\Longleftrightarrow$ $\exists$G such that G is a tree and G directly represents R;

d) $\langle D,R \rangle$ is a complete lattice and $((\forall a,b \in D)\ \neg(a\ R\ b)$ and $\neg(b\ R\ a)) \supset (\text{l.u.b.}(a,b) = \text{l.u.b.}(D))$ $\Longleftrightarrow$ $\exists$G such that G is a tree and G exactly represents $\langle D-\{\ell.\text{u.b.}(D)\},R \rangle$. ∎

The third notion of representation of relations that seems useful requires labeling of the arcs of a graph. Let a <u>T-F graph</u> be a directed graph each of whose arcs is labeled by T or F. The labels are to be used to define restricted reachability relations. Let $*\overset{F}{\rightarrow}$ be the smallest relation satisfying:

i) $(\forall a \in G)\ a *\overset{F}{\rightarrow} a$;

ii) $(\forall a,b,c, \in G)$ if $a *\overset{F}{\rightarrow} b$ and $(b,c) \in A$ and is labelled by F, then $a *\overset{F}{\rightarrow} c$.

Let $*\overset{T}{\rightarrow}$ be the smallest relation defined by:

iii) $(\forall a,b,c \in G)$ if $(a,b) \in A$ and is labelled by T and if $b *\overset{F}{\rightarrow} c$, then $a *\overset{T}{\rightarrow} c$.

The relation $*\overset{F}{\rightarrow}$ is reflexive and transitive; the relation $*\overset{T}{\rightarrow}$ is neither. Thus, $*\overset{T}{\rightarrow}$ is not an ordering relation, but rather a tool for describing a set of nodes in a bounded subgraph that is entirely reachable from a specific entry node.

<u>Definition 3</u>: If $G = \langle G,A \rangle$ is a T-F graph, $R \subseteq D^2$ is a binary recursive relation, $\psi : G \rightarrow D$ is a relational homomorphism with respect to $*\overset{T}{\rightarrow}$, and $\psi\ (*\overset{T}{\rightarrow})$ is a cover for R, then G is said to be an <u>indirect</u> <u>representation</u> of R. ∎

The idea of indirect representation is that a recursive relation can be represented as an enumerable set of ordered pairs. For each element <u>a</u> $\in \pi_1(R)$, where $\pi_1(R)$ is the pre-domain of R, the set $\{\underline{b} \mid \underline{a}\ R\ \underline{b}\}^1$ can be represented as a list, reachable in a T-F graph from the entry node <u>a</u>.

<u>Theorem 2</u>: Let R be a binary recursive relation over D^2. Then there is an effective construction of a T-F tree G which is an indirect representation of R.

<u>Proof</u>: The construction depends on the ability to enumerate elements of R. Only a binary tree is required. From each node will emanate at most one T-arc and at most one F-arc. An indirect representation tree has a special form. Beginning with the root node there will be a linear list of nodes linked by F-arcs; the image of this set of nodes will be $\pi_1(R)$. From each such node, x, will emanate a T-arc which provides a connection to a secondary list of nodes, having as its image $\{b \mid \psi(x) = a \text{ and } a\ R\ b\}$. The secondary lists will also be linked by F-arcs. The construction of such a tree from a recursive relation is obvious, and will not be given here. ∎

Returning to the original question posed in this section, the sufficiency of types <u>list</u> and <u>tree</u> for the representation of relations, a fairly precise answer can now be given. With type <u>list</u>-(t), one can give an exact representation, that is an enumeration, of an r.e. set of type t. With type <u>tree</u>(t), one can give exact representations of a restricted sub-class of complete lattices, the subclass specified in (d) of Theorem 1: one can give direct representations of the class of lattices with g.l.b.; and one can give indirect representations of recursive n-place relations. Since the data abstractions defined for types <u>list</u> and <u>tree</u> are compatible with the Principles of Encapsulation and Uniform Reference, their addition to programming languages intended for information processing applications on data bases of moderate size seems justified. Conversely, with the addition of these data abstractions, we see no justification for the continued use of pointer variables in such applications. The added representational power possible by the use of data structures created with pointer variables is of little practical importance, and necessitates violation of the Principles we have adopted.

7. <u>WHERE HAVE ALL THE POINTERS GONE?</u>

Throughout the development so far, and particularly in that of Sections 4 and 5, pointer variables <u>per</u> <u>se</u> have been avoided, although many of the capabilities afforded by their use have been provided through other, seemingly devious devices. The predominant means has been by binding of actual parameters in procedure calls, and by the ability to return values and even variables of compound types in function evaluations.

In the present section, we shall strip away the deviousness and discuss this crucially important mechanism. This section relies heavily on the concept of data abstractions and on some of the methods for their definition that have been proposed in SIMULA 67 [6] and in CLU [8].

In the examples of Section 5, data objects of types list and tree were passed as parameters to procedures without discussing the parameter binding rule that was to be used. If the purpose of a procedure is to effect a transformation on a structured variables passed as its argument, then obviously, the variable itself must be passed, rather than a copy of its value. On the other hand, if a parameter is passed by value, implying an assignment of the actual to the formal parameter, it is not so clear whether the assignment must be carried out by copying the value or whether the name of the formal parameter may be bound to the actual parameter. If the formal parameter is known not to be modified within the procedure body, and if its value cannot be modified by means of an alias reference, then in sequentially executed programs at least, the alternative modes of parameter binding cannot be distinguished. This suggests that an implementor may use his discretion to effect an efficient implementation, just as was suggested in discussing exchange assignments.

At the same time, we must acknowledge that the binding of parameters by reference is, in fact, the same operation as pointer assignment, although a formal by-reference parameter is not treated as a pointer variable within the body of a procedure. In the examples of the preceding sections, there is no magic that accomplishes list processing operations without the use of pointer variables; binding of variables has been used as a substitute for pointer assignment in some cases, selector assignment has been used in others. But in restricting the use of references to the interface between procedures and the environment from which they are called, we are afforded the opportunity to avoid the alias variable problem without crippling the power of the language to define operations on data structures, and without forcing implementation to become needlessly inefficient. The reason is that the boundary between procedure body and context of invocation provides a natural boundary of definitional context within a program; names known within a procedure body are not necessarily known without, and vice-versa.

If no object of a type that is to be passed to a procedure by reference is accessible by global definition within the procedure body, and if only a single object of that type is allowed as a by-reference parameter, then aliasing of variables within the procedure body cannot occur by means of parameter binding. In practice, it is entirely reasonable to impose an even more stringent restriction; a procedure or function may have at most one explicitly by-reference parameter, whose type may not coincide with the type of any global variable. Such a procedure usually has as its primary activity the role of an operator on the type of the by-reference parameter, and therefore should be defined as a part of a data abstraction encapsulating that type.

Some data abstractions are predefined in any programming language. Examples are arithmetics on integer or floating-point data types; character strings, built around an iteration type of a finite, scalar set of character values, and incorporating string processing operators; vector and matrix arithmetics, etc. We propose that a programming language to support list processing and relational data structures should also incorporate as standard data abstractions those described in Section 3 around the recursive types list and tree. In addition, a systems-oriented programming language should provide for data abstractions that are application-oriented and user defined. We shall discuss some aspects of data abstraction definition here, as it affects communication of parameters and results to and from procedures. We employ a class notation which is an extension of that originally adopted for SIMULA 67, but improved by giving explicit designation of the identifiers from the class definition that are to be made accessible outside the scope of the definition, as names of operators or selectors. This explicit designation is borrowed from CLU, and allows one to enforce the Principle of Encapsulation.

In its use in a program, a data abstraction definition is analogous to a type declaration; its appearance does not create a data object, and causes no storage allocation to take place, but it allows subsequent declarations of data objects belonging to the abstraction class. As in SIMULA, we advocate the use of postfix notation to designate the application of an operator or selector name defined for the abstraction class to a data object belonging to the class. Together with our previously mentioned convention on parameter binding, the postfix operator notation allows one to make an immediate visual distinction between by-reference parameter and the value parameters that may be passed to such an operator. For example, invocation of the list extension operator that was discussed in Section 3 would be applied to a list variable L as

```
L.Extend(X)
```

in preference to the notion previously used, Extend(L, X).

Another capability that is required is an operator to return access to a component of a data abstraction. This access may be in the form of a selector name for a variable. In fact, when we declare identifiers to be selector names for the components of a structured type in a programming language such as PASCAL, PL/1, or COBOL, it is done exactly to afford such access. It also happens that one may wish to describe a computed access to a component or substructure of a data abstraction. For this purpose, an operator can be defined to perform the access computation, returning a §-type. We call such an operator a selector function. The canonical example of a selector function is the mapping type that takes a value from a finite range of ordered scalars into §-type selected from an iterative replication type; this has been incorporated as part of the data abstraction called an array in many programming languages. Unlike a pointer-valued function that returns a ref type, the access returned by a selector function is not assignable, although it is usable.

Selector functions afford a means by which to communicate access to objects or components of objects that are shareable among several program modules, without simply declaring the objects to be globally accessible. Since the access computed by a selector function cannot itself be assigned, its introduction into a programming language does not by itself open the door to alias variables. However, if the access computation is dependent on the evaluation of parameter expressions whose formal equivalence is undecidable, or even difficult to decide, then multiple instances of invocation of the selector function itself may constitute alias references. This is the situation with array references whose index expressions may contain integer variables. Let us return to the example of the two-level directory discussed in Section 5, to illustrate the notation proposed for a data abstraction class.

```
class Directory = §list (char, §list (§char));

   access Extend, Deletehead, Defined, C, Lookup;

   selector C: {Directory} → char;

            Subcat : {Directory} → §list(§char);

   selector function Lookup (Name1, Name2 : char)
                    : §list(§char);

      begin

         while Defined and C ¬= Name1 do

            Directory selects Directory.tail;

         if Defined then

         begin

            while Subcat.Defined and {Subcat}
                  ¬= Name2 do

               Subcat selects Subcat.tail;

            return Subcat

         end

      end

endclass
```

The first line of the class definition defines the name of the data abstraction class and the type or data abstraction on which its data structure is to be based, in this case a list variable, each of whose elements is a structure or cartesian product type having a constant of type char and a §list (§char) variable as components. The second line gives the names of operators and selectors by which an object of the class may be accessed. Since a data abstraction for the underlying type includes several operators, any of these that are to be made applicable to the new abstraction must be listed, along with the names of any that are to be newly defined within the abstraction class definition. By their absence, the head ({ }) and tail selectors on §list types and the Subcat selector are not to be allowed as selectors on objects of the abstraction class Directory.

Next come the type declarations of a pair of selector names, C and Subcat. Since the second name has not been included in the access list, its definition is restricted to the body of the class definition. Since C is declared as the selector of an object that is not of a §-type, it cannot be assigned to. Subcat, on the other hand, selects an object having the type of a variable.

The selector function Lookup takes a pair of value parameters, neither of which may be assigned to in the body of the function because they have not been given a §-type in their declaration. It returns an object of type §list(§char), that is, a selector of such an object. Within the body of Lookup, any of the operators or selectors defined for the underlying data abstraction may be used, as well as the selectors C and Subcat. Since the function is to refer to any instance of the abstraction class, the abstraction class name is used in lieu of a selector name for an object of that class. Also, when an operator applies to the object of the abstraction class Directory, the argument of the operator may be implied, and the prefix notation for the by-reference parameter can be omitted for brevity. This convention aids the readability of the text of the function body considerably. Finally, a return statement is used to eliminate the need for an aliasing operator to bind the function name to the computed selector. If the return is never executed, the returned value of the selector function will be Null.

Although applying Lookup to a data abstraction variable binds that variable as the Directory parameter of Lookup, the selector name employed within the procedure body is treated as local. local. Thus, even though Lookup may apply the selector assignment operator repeatedly to Directory, its local selector name for the by-reference parameter, that does not cause repositioning of the selector name of the actual parameter. An invocation of Lookup applied to a directory variable called Catalog,

```
Catalog.Lookup(A, B)
```

therefore has no side effect in the context of the calling environment. This definition of locality of selector names obviously requires that selector values must be remembered as program execution passes through a nested calling sequence of data abstraction operators, and requires implementation support of a stack mechanism. However, within each definitional context of the program employing data abstractions, only a single value of the selector will be accessible.

There are several aspects of data abstraction definition that remain to be explored, but have not been discussed here. Among these are implementation; at what level in the description of a programming system does it become unavoidable to operate with address constants in order to define accesses to data? There is good reason to believe that this can be postponed, in a hierarchical, top-down definition of a programming language implementation, until the level of description of mapping of data types onto virtual memory, which is certainly a system-dependent level of description, if not actually machine-dependent.

In connection with hierarchical definition, the use of data abstractions affords a means of making precise the transitions between levels of conceptual abstraction. Although the examples given here have not been chosen to demonstrate this transition, the class notation has been designed to support it. A data abstraction is defined in terms of a more primitive data abstraction, its data structures and selection operators. Since a data abstraction may have type parameters, the rule of composition of types can be used to form a more specific abstraction from ones that are more general. Within the body of a data abstraction definition, all references to an object belonging to the abstraction are made to its representation in terms of the more primitive abstraction. In an applications program, references to an object of the defined abstraction cannot access its representation.

The idea of composition of abstractions has been explored in the design of the programming language CLU. We believe the class notation given here to be an improvement over that of CLU, but opinions as to notation are arguable. We also believe that the notion of selector function and the parameter binding conventions suggested here will meet the needs of data abstraction definitions, and have the advantages of simplicity, standardization, and prohibition of unwarranted side effects.

Finally, the use of types as parameters to data abstractions has been implicit in the definitions of list and tree, and can obviously be extended to other data abstractions. This affords no difficulty in the definition of data abstractions, insofar as no operations on the parametric types are utilized within the body of the abstraction definition. The only information about a parametric type that a compiler needs when a specific instance of a data abstraction is declared is the storage requirement, and this information is already in the data base of a compiler. Data abstractions, on the other hand, are not so easily made parameters of another data abstraction definition, since there is no easy way to make provision for unanticipated operations.

8. CONCLUSIONS

We have reviewed some of the objections to the use of pointer variables in high-level programming languages on grounds of mixing conceptual levels of abstractions, and interfering with the understanding of program semantics by creating alias names for variables. A formal description of un-interpreted data types was introduced as a basis for the subsequent development of recursively defined data types, and their incorporation into data abstractions. It was argued that data abstraction definition provides a top-down, modular approach to programming data structures for applications, and for specifying the operations and modes of access that are to be used on those data structures. By means of recursive types and data abstraction definition, pointer variables are to be eliminated from a high-level programming language, being suppressed to a level of abstraction at which bindings to virtual storage are programmed.

At the outset, four principal categories of use of pointer variables were outlined. Let's survey them again, and try to evaluate the likelihood that applications programmers could accept the elimination of pointer variables.

i) Inverted files. Although this application has not been discussed directly in this paper, if pointer variables were to be eliminated, an inverted file would identify objects by their directory names, rather than by their storage locations. Access would then be obtained by means of the selector function that accomplishes directory lookup and is defined as part of the data abstraction for the master file. This scheme has some advantages, for the directory name of an item does not change, even if the master file is reorganized in its physical storage area. If this mode of access is deemed too inefficient to use, then one of two corrective actions may still save the scheme. Since so much reliance is to be placed on directory lookup, this function should be optimized, and perhaps provided special hardware support to improve its speed. Another alternative would be to save pointers, along with directory names, in the inverted file, provided that some means of verifying a pointer access were provided, such as a unique tag for each item that must be matched by a tag associated with the pointer. Then, pointer access could be utilized so long as it could be validated, with directory access available as a backup. This scheme would obviously increase the space requirements of an inverted file.

ii) List processing. In most large list-processing applications, data abstractions informally defined through packages of subroutines are utilized to make programming feasible. We have addressed the use of the recursive data types list and tree to eliminate entirely the need for pointer variables. Given the ability to build more specialized data abstractions upon compositions of data types including list and tree, there is little question that their use would improve readability and reduce the incidence of programming errors, when compared to composing list processing programs by utilizing pointer variables. There is no reason to believe that the use of the data abstractions defined for these recursive types would compromise efficiency.

iii) Representation of relations. Underlying the use of data structures in all applications is the need to support the representation of relations among data items. In Section 6 we have tried to make precise the representational power of the data types list and tree. An alternate approach, that of sharing nodes of a recursive data structure when a relation is known to be a lattice order but is not necessarily a tree, has been thoroughly discussed by Hoare [9].

iv) Sharing of a data base. In Section 7 the selector function was suggested as a basic device that can be used to meet the need for a computed access to a shared data base. Although no explicit example was given, it is evident that the identity of a program module attempting to interrogate a shared data base could be passed to a selector function, enabling the use of discrimination in computing accesses. The selector function differs from the use of a pointer-valued function in that the access cannot be stored by the interrogating module. Thus it is forced to use the access immediately.

Efficiency need not be lost because of the need to invoke a selector function each time an access is required. If a compiler can assure itself that a subsequent access computation will duplicate an earlier one, then it can cause a pointer to be saved instead. This optimization is just a special case of redundant expression elimination, an activity already performed by many compilers. The possibility for this optimization gives a compelling reason to define selector functions so that they cannot have side effects, for otherwise reevaluation would be mandated in order to produce the side effect.

REFERENCES

[1] Hoare, C.A.R., Hints on Programming Language Design, ACM SIGACT/SIGPLAN Symposium of Principles of Programming Languages, Boston, Mass., Oct. 1-3, 1973.

[2] Dijkstra, E.W., GOTO Statement Considered Harmful, (letter) Comm. ACM, Vol. 11, No. 3, March, 1968.

[3] Wirth, N., The Programming Language PASCAL, Acta Informatica, Vol. 1, No. 1, pp. 35-63, 1971.

[4] Jensen, K. and N. Wirth, PASCAL User Manual and Report, Springer-Verlag, Berlin, Heidelberg & New York, 1974.

[5] Wirth, N. and C.A.R. Hoare, A Contribution to the Development of ALGOL, Comm. ACM, Vol. 9, No. 6, pp. 413-431, June, 1966.

[6] Dahl, O.-J., B. Myhrhaug and K. Nygaard, The SIMULA 67 Common Base Language, Norwegian Computing Center Forskningsveiven 1B, Oslo 3 Norway, 1968.

[7] Dahl, O.-J. and C.A.R. Hoare, Hierarchical Program Structures, in Dahl, Dijkstra and Hoare, Structured Programming, Academic Press, London and New York, pp. 175-220, 1972.

[8] Liskov, B. and S. Zilles, An Approach to Abstraction, Computation Structures Group Memo 88, Project MAC, M.I.T., Cambridge, Mass., Sept. 1973.

[9] Liskov, B., A Note on CLU, Computation Structures Group Memo 112, Project MAC, M.I.T., Cambridge, Mass., Nov., 1974.

[10] Wulf, W.A., ALPHARD: Toward a Language to Support Structured Programs, Carnegie-Mellon University, Computer Science Department, Pittsburgh, Pa., April, 1974.

[11] Hoare, C.A.R., Notes on Data Structuring, in Dahl, Dijkstra and Hoare, Structured Programming, Academic Press, London and New York, pp. 83-174, 1972.

[12] Hoare, C.A.R., Recursive Data Structures, Stanford University, Computer Science Department, Stanford, California, STAN-CS-73-400, October, 1973.

[13] Barron, Buxton, Hortley, Nixon and Strachey, The Main Features of CPL. Comp. J., Vol. 6, pp. 134-143, 1963.