DEPARTMENT OF COMPUTER SCIENCE

AUSTRALIAN NATIONAL UNIVERSITY

REFERENCES AND/OR EFFICIENCY IN HIGH-LEVEL LANGUAGES

BY

C.W. Johnson

TR-CS-78-02

REFERENCES AND/OR EFFICIENCY IN HIGH-LEVEL LANGUAGES

C.W.Johnson,
Department of Computer Science,
School of General Studies,
Australian National University,
P.O.Box 4, Canberra, A.C.T. 2600

The role of reference variables in producing efficient but insecure data structure spaghetti is similar to that played by GOTOs in un-Structured Programming. Secure programming language constructs are suggested to replace various identified uses of references. Efficiency of program implementation is maintained by allowing specifications of the desired representation method to be added to the abstract data structure definitions.

Key words: references, pointers, data abstractions, data representation specifiers, abstract data types, programming languages.

CR Categories: 4.12, 4.20, 4.22, 4.34.

Paper submitted to
Eighth Australian Computer Conference
Canberra August 1978

INTRODUCTION

Our concern is with the data structuring facilities in general purpose algorithmic languages, their impact on reliable programming, and the efficient implementation of the resulting programs. By high-level languages we mean the line of development of algorithmic languages including Algol 60, PL/I, Pascal, Algol 68, Simula 67, CLU, etc. We take an object-oriented view of computation, in which an object is an instance of a data type, that is an unstructured or structured variable, and a structured object has a number of discrete components that constitute its value, each one of which may be a structured or unstructured variable. We shall be considering abstract programs and their implementations on general-purpose computers, referred to as "machines": the abstract program is an algorithmic specification (in a language for which an acceptor does not necessarily exist) of some computation, its implementation an equivalent program at machine level. The abstract program uses abstract data structures with no necessary relation to actual machines, while the implementation of the program consists of representations of its abstract data structures and algorithm in machine-level data and control structures. The abstract notion of an object described as a list or tree is a single object, though it may be represented by a number of linked objects at machine level. The efficiency of an implementation is a measure combining machine space and time taken to run the machine program, i.e. to perform the computation specified by both machine and abstract programs. Reliability of representation is the extent to which the machine program which purports to represent the abstract program actually does so.

The reference, or pointer, in high-level languages is a very powerful feature. It extends the power to manipulate the values of simple and composite variables to the manipulation of objects of variable and complex forms. It is, however, almost unchanged from the machine-level concept of a stored address and bringing it into a high-level language has some grave disadvantages:

- the very power and generality of references (inherited from machine addresses) makes them a very dangerous tool. Compile-time undetectable errors in using them can break the type-security system that the high-level language otherwise provides. Other mistakes in pointer manipulation can have far-reaching effect and unobvious program origin (e.g. accidentally establishing cycles in a structure where none is intended, no error being detected until a procedure that assumes no cycles exist attempts to traverse the structure) because there is little clear correspondence between the forms in the program text and the abstractions of the intended computation.

- a large collection of pointer-using techniques is known, but only very informally. Each programmer must be taught these or re-invent them to be able to implement a range of data structures. References fill many different roles in these implementations: the appearance of a reference in a program does not carry with it any information about its intended function. It s therefore not easily seen from the

program which abstract structure is being implemented, nor whether it is being implemented correctly.

Some of these problems have been reduced by the restriction of reference variables to point to objects of a single specified type, and the (expensive) removal of any explicit release of the storage of a referenced object, which together prevent any violations of the type security system and make program correctness proofs possible. References still model the machine's structures and not any abstract notion of computation, and the use of references in a high-level language leads to violations of the "structured" principles of clarity and safety of programs which high-level languages seek to maintain. The problem has recently been addressed by several workers, e.g. Hoare (1975), Berry et al (1976), Kieburtz (1976), Dembinski and Schwartz (1977).

References are necessary at some level in the implementation of high-level languages because desirable abstract data spaces cannot be modelled on conventional machines without them, but this alone does not justify their inclusion in high-level languages where they are an inappropriate concept for the programmer to work with. Their presence is supported on the grounds of efficiency of implementation (Berry et al): by placing references where he/she desires, or explicitly choosing among alternative data structures while describing an abstract program in the high-level language (e.g. copying objects that would otherwise be shared; providing expansion space in a vector for a size-changing sequence or regular tree; placing notionally distinct but related objects in adjacent locations), the programmer can achieve an efficient realization of the abstract program.

However, the reference concept is poorly integrated into higher-level languages and may introduce some possibly major inefficiencies beyond the programmer's control:

- the general use of the accompanying dynamic storage allocator (for all structures in Simula 67, all those with pointers to them in Pascal and derivatives, HEAP store in Algol 68) is very expensive, either in time for garbage collection or sufficient space to avoid it. The alternatives to this are explicit de-allocation of objects, with associated dangers to type security, and/or the programmer providing specialized storage allocators, which is expensive and unreliable.
- the use of a stored machine address to encode structure makes communication of structured data between programs impossible without re-encoding the address for the particular storage device used (to avoid this, Algol 68 transput, Pascal and Simula 67 i/o handle nothing but groups of elementary values).
- the programmer cannot control placement of references with respect to machine page boundaries, thus risking page thrashing and/or pipeline/ cache under- utilization.

The notion of record AREAs has been suggested to reduce the paging and garbage collection problem (Baecker 1975) (and to increase clarity, as shown by ease of correctness proof: e.g. Euclid's collections and zones) but has no effect on pipelining, nor on paging large collections of objects.

We first consider the uses of references and why they arise, and then suggest the elements of an abstract data structure manipulation language with a separate representation specifying facility.

THE USES OF REFERENCES

The general power provided by freely manipulable references is that of modelling dynamic binding of objects to names and as components of other objects. Linear uniform
to names and as components of other objects. Linear uniform
storage may be used for representing objects and their bindings where these have invariant form and are static. This requires mapping the objects onto machine addresses (actual or virtual), an address space which is quite inflexible, permitting no squeezing in or removal of elements. The mapping from program text to address is non-dynamic. Hence to bind, share and unbind dynamically a dynamic re-mapping of binding is needed, which is provided by addresses as the values of reference variables.

The uses of references in implementing and within high-level languages can be grouped as follows.

(a) References are one method of implementing structures that are wider or deeper than the storage provides directly, or have variable width (size) or depth (levels of structure). There are the alternatives of mapping excess (invariant) depth of structure onto elements at the one level (e.g. the standard representation of the fields of a Cobol record); and the provision of extra space for limited expansion within a fixed containing object (e.g. the standard size-limited stack representation). But the choice of a stored reference implementation is necessary for more general variability of size in structures, which if invariant could be represented directly, because of the inflexibility of the storage medium.

(b) They allow the dynamically variable binding of names to objects. This makes block-structured environments with logically separate multiple instances of the one local environment possible, and also the traversal of composite structured objects by reference variables ranging dynamically over their elements. (FLEX arrays and "variable length" arrays fall into this category and not, as their name suggests,
arrays fall into this category and not, as their name suggests,
into (a) above: they allow the binding of one name to logically distinct objects of different sizes, but not any variation in size of the particular object so bound).

(c) They enable the lack of any binding of a name as a

special case of variable binding to be represented by using a particular reference value, usually denoted by NIL.

(d) References allow the shared binding of more than one name to an object at the one time, while allowing these bindings to be separately established and broken. Structures with shared substructures, cyclic structures, "by reference" parameters, traversing variables ((b), above) are instances of this.

(e) The general modelling of dynamically variable directed graphs is done with objects as the nodes and references to represent the arcs between them. The relations represented by the arcs can then be dynamically variable, many to one (shareable) and non-existent (by NILs).

DEALING WITH REFERENCES

The abstractions of variable binding, form and size can be well defined without recourse to a particular implementation description using pointers. This tells us little about how to integrate these abstractions into a programming language because at the same time we cannot afford to ignore the economics of storage space occupied and time taken by the running of an implemented program: the efficiency of the concrete representation matters. The specification of abstract program and of its implementation must however be kept as separate as possible. This allows the program to express the programmer's actual intentions without the distortion of implementation peculiarities. Changing representation choices, or tuning the program for specific data values or machines, are easy without major changes to its fabric.

One stream of thought (e.g. Schwartz 1975) takes the separation of programming language from machine structure as far as providing only abstract structures, possible representations being only discovered by quite sophisticated analysis of the program. One objection to Very High Level Languages is that the abstractions provided are inappropriate to describe the abstractions of object- oriented programming; another is the difficulty of obtaining an efficient implementation. At almost the opposite end of the spectrum it is argued that hiding either GOTOs (Knuth 1974) or pointers (Berry et al 1976) from the programmer leads to presently unavoidable inefficiencies of implementation. The suggested solution for pointers (Berry et al 1976, Shaw 1976) is to restrict their use to inside protectively sheathed modules each of which describes both the meaning and the implementation of a single data structure abstraction (CLU clusters, Alphard forms). Berry et al claim that this provides the implementation efficiency desired along with the reliability implied by the ability to then do formal proofs: the proofs are demonstrated but the efficiencies assumed. In particular, the modules describing data structures ("clusters": the base language is CLU) may be freely shared (the standard assignment mechanism is by binding, alias "reference assignment" in Simula 67's terms): the hiding

of pointers within clusters may help with implementing the internal low-level detail of separate data abstractions but not with the various binding abstractions basic to the language. A cluster is both the definition and implementation of a data structure abstraction; hence while a representation may easily be altered because its details are all within the one cluster, there is no way to protect the meaning of the cluster (i.e. the abstraction it defines) when changing its representation.

We hope to demonstrate that separating the various notions of binding and variability from each other and from the implementation- level reference concept allows more transparent programming, less tied to the machine. This requires refining the notion of data type as applied to variables to include an indication of the kind of binding the variable may have to objects of "its" type. Efficient representation methods for the objects and bindings specified in the abstract program can then be chosen and specified without affecting its clarity.

THE REFERENCELESS LANGUAGE

Consider a mini-language in which a program characterizes an abstract machine, by defining its data structures and its instructions (as procedures). The language expresses abstract data types as distinct capsules in terms of other abstract and elementary types using the basic structuring methods of named parts, simple ordering, and regular recursion. This "clump" encapsulation follows CLU, Alphard, Euclid etc., and hence we believe powerfully and manageably expresses the concept of "data type" for algorithmic languages. The data type constructors provide objects with more flexibility than do these languages, matched with declarative forms to specify where this flexibility is made use of. Assignment is by binding and by copying, and the abolition of bindings is possible. CLUMPs describe data objects by their internal state and the operations that may be applied to that state. Object instances may be bound to variables of the same type, or into other objects as components (of the same type also), and may be explicitly or implicitly created. Neither procedures nor particular instances of them may be bound dynamically to names or as object components. There are no restrictions on data types being recursive.

A caveat: the range of data structuring facilities described in the mini- language and accompanying representations is limited by the space available here. Specific forms for regular recursive types (e.g. binary trees) and their representation other than as reference- linked discrete records therefore are not described.

An example portion of a program follows, concerned with describing a simple model of persons, their spouses, cars and favourite drinks.

```
    PROGRAM EXAMPLE
 1: CLUMP automobile = (
 2:        STATE: STRUCT(year: int; value: real;
 3:                      POSSIBLY motor: posint ) ;
 4:       hidecapacity = PROC: BEGIN motor <- NONEX END ;
 5:       ) ;
 6: CLUMP person = (
 7:       STATE: STRUCT ( bankac: real;
 8:                      POSSIBLY prefdrink: drink ;
 9:                      POSSIBLY spouse : person ;
10:                      car : automobile ) ;
11:       buycar =   PROC ( newie: automobile) ;
12:           BEGIN
13:           bankac := bankac + car . value ;
14:           car <- newie ;
15:           bankac := bankac - car . value ;
16:           WITH hw = spouse
17:             WHEN EXISTS DO
18:               hw . changecar (car) ;
19:           END ;
20:       changecar = PROC ( newie: automobile ) ;
21:            BEGIN car <- newie END ;
22:        findpref = PROC ( pub: hotel ) ;
23:            BEGIN
24:            WITH glass = prefdrink
25:              WHEN EXISTS DO (*nothing*)
26:              OTHERWISE
27:                FOR try : OVER pub.drinks DO
28:                  WITH bestyet = prefdrink
29:                    WHEN EXISTS DO
30:                      IF try.taste > bestyet.taste THEN
31:                            prefdrink := try
32:                    OTHERWISE prefdrink <- COPY ( try );
33:            END
34:       ) ;
35: CLUMP drink = (STATE: ANYNUMBER OF ingredient ORDERED;
36:        taste = FUNC: int; BEGIN ... END;
37:        addone = PROC ( newingr : ingredient );
38:            BEGIN
39:            GROW  AT 1 WITH COPY(newingr)
40:            END  ) ;
41:   SHAREABLE ( person.spouse, electoroll.ALL) ;
42:   SHAREABLE ( person.car, person.buycar.newie
43:                 person.changecar.newie ) ;
```

A CLUMP (line 1) encapsulates an abstract data type, by the possible states of an object and the applicable operations. A STATE (lines 2-3) is either a STRUCT, of parts with names and types each of which may be marked as POSSIBLY unbound (line 3, 8, 9) (and is otherwise always bound to some object); or an ordered collection of either fixed, limited or unlimited size (<integer> OF <type>; UPTO <integer> OF <type>; ANYNUMBER OF <type>) of objects of the one type (line 35). The operations follow the STATE declaration as procedure (PROC) and function (FUNC) declarations. They are called remotely by dot notation (line 18) or locally by their

simple name (no example here). Calling by dot gives the procedure access to the state of the particular object specified ahead of it in the dot expression (as in Simula 67). Objects' states may not be remotely updated, but may be remotely accessed (line 13).

Two assignment operators are used: one (:=) to copy a value (simple or structured) into an existing object (line 13, 31), the other (<-) to rebind the left hand name or component to a different object or to none (line 4, 14). The corresponding comparators are = (equals) and <-> (sameas). NONEX denotes non- existence, or no binding: the construct "WITH <name> = <object> WHEN EXISTS DO <statement> {OTHERWISE <statement>}" (line 16, 28) is a discriminator on the existence of a binding with obvious meaning.

New objects are explicitly created by the object expressions "NEW <type> (<component list>)" (no example here) and "COPY (<expression>)" (line 32). An ordered object may be changed in size by shrinkage or growth (line 39) within the continuity of current elements. "GROW AT n WITH <object>" makes that object become the nth component of the owner (the containing object's state), the previous nth, if any, becoming the (n+1)th and lower- ordered components than n (if any) being unaffected.

The declarations of component, variable or function names that may be bound to shared objects are written as type and operation names with dots followed by component or parameter name (lines 41 & 42) in a SHAREABLE declaration. The name "ALL" is used here to denote the components of an ordered structure. The various instances of a single sharelist name may be bound to common objects (e.g. person.car, line 42), as well

as the other names in the list. SHAREABLE lists must all be disjoint; the names in any one list must be of the same type, but there may be more than one list of the same type. Parameter passing and component binding on creation of an object are done by the same binding mechanism as the rebinds operator. Parameters that might be passed by reference in a simpler language are shared with the actual parameter; if not shared, an actual parameter expression must be "COPY...".

A traverser variable (line 27) is restricted to traversing over the elements of objects denoted by a single identifier of ordered type. It need not appear on a sharelist for those elements: it is part of the articulation mechanism of the ordering, rather than simply a variable. Once initialized, the traverser is confined to a single object until re- initialized. The operations on a traverser are to initialize it to the start of a particular object, step it on to the next element from its current position, check whether any elements remain to be traversed, grow and shrink the ordered object at the traverser's current position, and access the current element (by using the traverser variable in a context where an object expression is expected - line 30). Some of these operations, and the declaration of the traverser variable, are collapsed into the form

"FOR <traverser> : OVER <object> DO <statement>"

with the obvious meaning. Indexing ordered structures may be used as well as traversing.

Recursive types are made little use of here (e.g. person). They have provision for expressing whether cycles or convergences (sharing of branches) are possible, and their own class of traverser variables, with a wider range of operations than those for simple ordered structures.

The specific uses for reference variables in high-level languages listed above are here expressed by:

(a) variable width: allowing ordered objects to be of possibly unlimited size, and to grow and shrink at any place at will.

(b) dynamic rebinding (which provides variable depth): the rebinds operator (<-).

(c) non-existent binding: marking a name with POSSIBLY to indicate that there is only possibly a bound object, while allowing the apparent re- binding to NONEX to denote the loss of any binding, and the existence discriminator "WITH..." for protected computing with possibly non- existent bindings.

(d) shared binding: defining SHAREABLE lists of names and components that may simultaneously be bound to shared objects, by using rebinds assignment, and the provision of distinct traverser variables which are constrained to operate in the composite object corresponding to a single abstract object.

(e) general directed graphs: the STRUCT and ORDERED state mechanisms, which are unrestricted in their recursive abilities (but POSSIBLY should be used for STRUCTs), are able to define the element data structures of graphs, with sharing and variability of form as above (b, c, and d).

These notions expand the vocabulary of programming and hence enable programs to be constructed more easily and reliably, and to be obvious in intent. Such a language proposal is however only a small advance unless the implementation of a program can be better than the pessimistic automatic method of merely representing each shareable, possible or re- binding by a reference variable, and variable ordered structures by reference- linked linear lists. To implement efficiently, we must consider the alternative representations available: we shall define names to specify the representation methods as we go.

For the notion of "possibly no binding" (marked by POSSIBLY) the alternative to substituting a reference whose range of values includes a NIL value is to extend the range of the possible component itself, to include a value that will represent nonexistence of the binding. Elementary types (including reference) may have some value(s) from their range

taken to mean non- existence. This value extension is denoted by "EXTEND ELEMENTARY". A non- elementary type can have its range of values extended only by introducing an extra Boolean component whose value by itself indicates the existence of any binding to its owner. For each binding to a shared object to have separate indication of existence, there must be a separate existénce component for each possible owner. "EXTEND" denotes this form of extension which may also be applied to components of elementary types.

The use of a reference for any purpose, including the representation of possible bindings, sharing, or dynamic bindings, we denote by "REF OUT" - because the reference displaces the object it refers to. The alternative to references for dynamically sharing a common object is to maintain separate copies of the object, denoted by "MULTICOPY". This representation is only reasonably simple when no selective changes are applied to the copied object, since these must be reflected in every copy. For dynamic rebinding we provide no representation other than "REF OUT" and "MULTICOPY",but note that in limited cases it may be implemented by value copying.

The default representation of ordered and part- named structures of elementary components is by direct mapping onto adjacent machine storage locations. "Storage location" normally means "smallest addressable unit" (i.e. word or byte) - adjacent bit-fields (loose packing) can also be specified, at some cost. This representation is the only one that may be specified separately from the notion of type in any of the Algol-based languages - Pascal. We also denote it by "PACKED". A non- elementary component may either use "REFOUT", or integrate the components of that component with those of the owner. This destroys the separate identity of the component, and hence this method (denoted by "INCORPORATE") is forbidden when that component may be shared.

An ordered structure may be represented by:

- a simple vector, which is allowed only for a fixed- size structure (default).
- a fixed- size record consisting of a fixed vector of the structure's maximum size plus one or two integer (subrange) index variables, which is the conventional representation of a limited-size stack or queue. This can only apply if the structure has a known maximum size. It is denoted by "CONTAIN"; the "used" part of the containing vector may be either at the "top" or "bottom" (needing only one index variable: e.g. conventional stack) or be anywhere within the "wrapped around" vector, using two indices for its start and finish (e.g. queue). These are denoted by "CONTAIN AT TOP", "...AT BOTTOM", and "...WITHIN".

  (Note that changes of size of the contained object away from the "ends" do not require these representations, but non-end changes are more expensive);
- a reference- linked list of records, each with an element of the ordered structure and a reference to the next such record (NIL for the last) - denoted by "REFLINK".

The representation of a corresponding traverser variable and its operations differ for each of these. For "REFLINK" it is a reference variable, including NIL in its range, otherwise being an index variable ranging over the vector.

These representation specifiers are to be added to a program text to turn it from the specification of a purely abstract program to that of a concrete, machine-acceptable one. Working in the program text domain (and wishing to avoid interpreting representations at runtime), we cannot specify the representation of individual objects. However, we wish to avoid having a single representation for all objects of one type: the representation often exaggerates small differences of behaviour ignored when abstracting a data type. Program textual correspondence requires that each owner of a shared object see it represented identically, but allows change of representation when object values are copied. Representation specifiers are therefore attached to component (and variable) type specifiers, whether applying to the binding (REFOUT, MULTICOPY, EXTEND - which have no effect on the component's representation - and INCORPORATE and EXTEND ELEMENTARY, which do affect it), or the objects bound (CONTAIN, PACKED, and REFLINK). Two convenient shorthands: all instances of a type may be represented alike, by specifying it at the clump's declaration; and the single representation of a shared object may be specified at the sharelist.

A non- structural specifier "MAX <integer>" allows an implementation limit to be imposed on a clump with state "ANYNUMBER OF <type> ORDERED". In itself this implies no particular representation, but allows a choice between REFLINK and CONTAIN to be made for ANYNUMBER in the abstract.

EXAMPLES OF REPRESENTATION SPECIFICATION

(a) The representation of an "automobile", and in particular its binding to the "motor" component.

```
     CLUMP automobile = (STATE:STRUCT
           (...POSSIBLY motor:posint...)...)
```

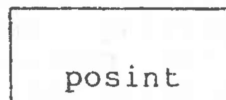
POSSIBLY cannot be represented by the default implementation, which requires that a component be permanently present. The usual representations are

- to make a separate object for the "posint" which may be bound here, and represent the binding by a component of reference type extended to include NIL to denote the lack of any posint.

```
(1)    ...POSSIBLY motor:posint REFOUT & EXTEND ELEMENTARY...
```

(an automobile)

OR



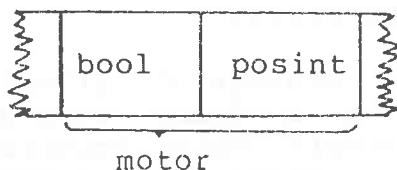
- to make a separate object which has two components: one indicating the logical presence of the binding, the other being a posint variable that is only looked at when the first component is true.

(2) ...POSSIBLY motor:posint EXTEND...

There are two ways of representing the resulting 2-level structure: either to suck up both components among those of automobile

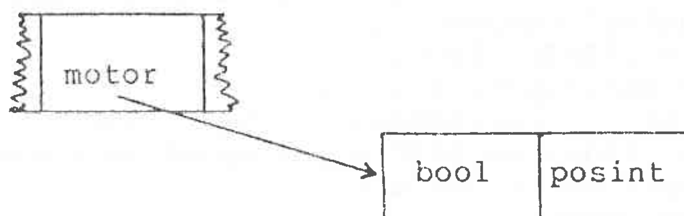
(3) ...POSSIBLY motor:posint EXTEND & INCORPORATE...

(an automobile)



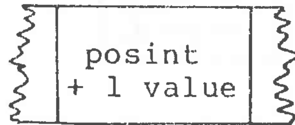
or to refer to a distinct object that consists of the two

(4) ...POSSIBLY motor:posint EXTEND & REFOUT...

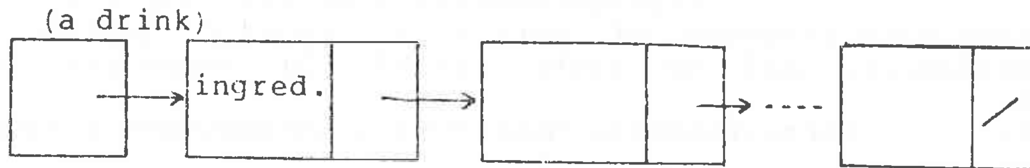


In this case the reference need not be extended with a NIL value because the possibility of logical non-existence has been absorbed by the extension with an extra component.

- to extend the range of values of a motor component in place, assuming it is an elementary type
(5)   ...POSSIBLY motor:posint EXTEND ELEMENTARY...



(b) The representation of every instance of a drink.
CLUMP drink = (STATE:ANYNUMBER OF ingredient ORDERED;...
Without any limit on size, the only representation that can be used is a linked list of dynamically allocated elements
(1)   STATE:ANYNUMBER OF ingredient ORDERED REFLINK;

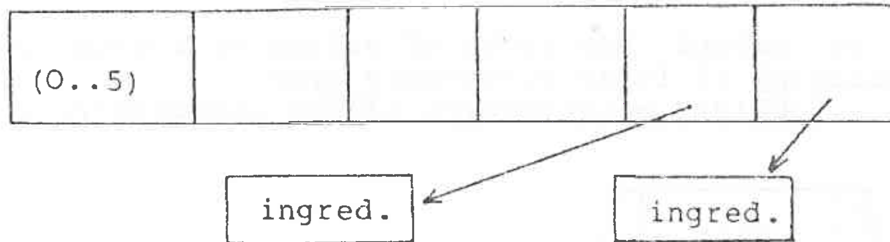


(The empty drink must be distinguished from the non- existent - hence the "header" element). All references in the reflinked representation are EXTENDED ELEMENTARYs with NIL in their range of values).
(The representation of the binding to the component ingredients will not be pursued here.)

Once a limit is placed on the extent of the sequence, other representations become possible, but REFLINK is not excluded.
(2)   STATE:ANYNUMBER OF ingredient ORDERED MAX 5;
may be represented by containment
(3)   STATE:ANYNUMBER OF ingredient ORDERED MAX 5 & CONTAIN AT TOP;
This is strictly incomplete, since there are now 2 components, an index variable and a vector of 5 ingredients. By default the elements of the vector will be incorporated into the representation of the drink, giving us

which represents a drink with 2 ingredients.

A vector of references to ingredients, rather than the ingredients themselves, may be specified by
(4)   STATE:ANYNUMBER OF ingredient REFOUT ORDERED MAX 5 & CONTAIN AT TOP



(c) All instances of automobile may be shared, and so any representation specification at one of the owners must also be at each of the others. The representation of the shared binding may differ at each owner, however, and if copying is used to represent shared binding by MULTICOPY then the copies so obtained may be differently represented.

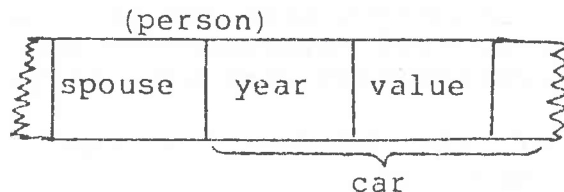
(1) SHAREABLE (person.car, person.buycar.newie,
person.changecar.newie) REFOUT;

represents all these bindings by a reference variable.

(2) CLUMP person = (STATE:STRUCT
(...car:automobile MULTICOPY...)...)

gives each instance of person a separate instance of an automobile, and we might extend this representation further to

(3) car:automobile MULTICOPY & INCORPORATE & PACKED



DISCUSSION

The representation specifiers described here can be applied in ordered combination to produce a wide range of data structure representations. They are the elements of a sub- language for data representation, used applicatively on the data definitions of the abstract program and on the results of previous applications to specify structurally related data types with equivalent behaviour to the originals'. They are only a partial solution to the problem (of which implementing particular programs on particular machines is itself only a part) of describing programs of all kinds powerfully enough to capture their equivalences and differences, i.e. a Theory of Programs. These data structure representation specifiers have a structural basis which makes them a contribution to the theory, although they are limited to programs which make clear the somewhat arbitrary distinction between data and algorithm components, and further restricted to applying only to individuals among the data structure definitions and not at all to the control structures. Given these limitations, specifying representations by adding specifiers to abstract program text

can be judged on four grounds: the ability to describe a range of representations; the extent to which implementation efficiency can be improved over a default representation; the ease with which the representation can be developed from the abstract program; and the reliability of the development.

While a careful, clever programmer starting from the same abstract program could produce a possibly better implementation, the result produced by specifiers is more reliable, being certainly equivalent to the abstract program. Because clump definition and representation are expressed separately, representation choices can be made (and altered) with no risk that the meaning of the program might be changed by the process of representing it. This method of applying representation specifiers to the data structure abstractions of a high-level language is therefore a tool that advances our abilities to do high-level programming at low cost.

The only storage structure considered here is a very generalized one, of linear coordinates and fixed size (word/ byte) elements. A more complete treatment would require a general descriptive method for the "target" storage medium (or media), but handling vari- or richly- structured targets is much more difficult than representing even very rich abstract objects on a simple target machine. Representations involving major structural changes perform adequately, however, with this simplified storage model. Non- uniform timing characteristics of storage have also been ignored. While the storage medium is uniform space/ time tradeoffs can be left to the representer's own management, but in multi-level stores placement of data and timing tradeoffs because of placement of references become more critically interdependent. Simple, independent data representation specifiers may not be adequate to describe the desired implementation, and are of little help in handling the complexities of the process of designing it.

Apart from coping with such refined models of the machine, nearly all the textbook representations can be described with the addition of a few more representation specifiers. More efficient representation can be had by extending the range of representations within the same framework. Some examples for future consideration are:

- with extensions of these methods we would expect to be able to specify local storage allocators for particular data types or particular variables. The "CONTAIN" representation is only one example of a representation including storage allocation and de-allocation that is tailored to particular patterns of storage usage and hence may be much more efficient than making use of the global allocator and garbage collector for certain objects. The range of similar methods is unknown.

- the independence of representation specifications from each other and from all but one data type in the abstract

program is an advantage in constructing the program by allowing attention to be narrowly focussed, but it means that some quite desirable implementations of the whole program are unreachable. For instance, reorganizing the algorithms used (e.g. from repeated "production" and "consumption" of single data items to "produce all; consume all") and changing the intermediate data structures to match may be a better implementation for the abstract program. The initial factoring of program into data and algorithmic components makes such notions difficult to pursue.

The specifiers are not easy to use for program development. Some of the examples above need three or four specifiers to describe the representation of a single clump, and after even only two or three specifiers it is often not clear (even to the practised user) what further specifications must be made to achieve a program that is acceptably close to machine structures. The effects on the algorithms that operate on the altered data structures can only be imagined with difficulty, and no assistance is given towards designing an economical space/ time tradeoff. While a program that includes representation specifiers is a good textual description of an implementation of an abstract program, the representation specifiers are not a good tool for deriving it. The difficulty arises because although each specifier captures a structural equivalence of data structure or binding, the expression of this equivalence in the program surface text is not simple.

In this form, representation specifiers are limited. However, they suggest a basis for a Program Construction system, or mechanical programming assistant, that will aid in the process of developing programs from abstractions to concretions. The specifiers may be seen as applying transformations to the textual definition of the program, the effect of each being localized to the state and operations of one data type. Several successive transformations of a data type definition may be needed, but only the end result need be close enough to machine structures to be a "representation" of the data structure. (See Loveman (1977) and Kibler et al (1977) for an initial look at mechanically assisted program transformation). The text of the program and estimates of relative time and space requirements could be held and produced by the programming assistant system for the programmer to develop a "best" implementation, the effect of each data structure transformation being reflected in the current program text. The original text remains the abstract program definition, with well- defined semantics; the final result of all transformations is only of interest as input to an automatic compiler, to take it down to actual object code. The transforming operators, their points and orders of application to the original text give a notation for specifying representations allowing notions of representations to be discussed clearly and precisely in the abstract.

BIBLIOGRAPHY

ALPHARD: Wulf,W.A., London,R.L., Shaw,M. (1976): "Abstraction Verification in Alphard: Introduction to Language and Methodology",
University of Southern California, Information Sciences Institute, 1976.

BAECKER,H.D. (1975): "Areas and Record-classes",
Comp.J., vol 18, no 3 (Aug 1975) pp 223-226.

BERRY,D.M., ERLICH,Z., LUCENA,C.J. (1976): "Correctness of Data Representations: Pointers in High Level languages",
Proc. Conf. on Data: Abstraction, Definition and Structure (SIGPLAN Notices , vol 11 (1976), special issue) pp 115-119.

CLU: Liskov,B., Snyder,A., Atkinson,R. (1977): "Abstraction Methods in CLU",
C.ACM, vol 20, no 8 (Aug 1977) pp 564-576.

DEMBINSKI,P., SCHWARTZ,R. (1977): "The Taming of the Pointer",
SIGPLAN Notices, vol 12, no 7 (July 1977) pp 60-74.

EUCLID: Lampson,B.W., Horning,J.J., London,R.L., Mitchell,J.G., Popek,G.L. (1977): "Report on The Programming Language EUCLID",
SIGPLAN Notices, vol 12, no 2 (Feb 1977) complete issue.

HOARE,C.A.R. (1972): "Proof of Correctness of Data Representations",
Acta Informatica, vol 1, no 4 (1972) pp 271-281.

HOARE,C.A.R. (1975): "Recursive Data Structures",
Internat. J. Computer and Info. Sciences, vol 4, no 2 (1975) pp 105-132.

KIBLER,D.F., NEIGHBORS,J.M., STANDISH,T.A. (1977): "Program Manipulation via an Efficient Production System",
Proc. Symp. on AI and PLs (SIGPLAN Notices, vol 12, no 8, Aug 1977) pp 163-173.

KIEBURTZ,R.B. (1976): "Programming Without Pointer Variables",
Proc. Conf. on Data: Abstraction, Definition and Structure (SIGPLAN Notices, vol 11 (1976), special issue) pp 95-107.

KNUTH,D.E. (1974): "Structured Programming with GOTO Statements",
Comp. Surv., vol 6, no 4 (Dec 1974) pp 261-301.

LOVEMAN,D.B. (1977): "Program Improvement by Source-to-Source Transformation",
J.ACM, vol 24, no 1 (Jan 1977) pp 121-145.

SCHWARTZ,J.T. (1975): "Automatic Data Structure Choice in a Language of Very High Level",
C.ACM, vol 18, no 12 (Dec 1975) pp 722-728.

SHAW,M. (1976): "Research Directions in Abstract Data Structures",
Proc. Conf. on Data: Abstraction, Definition and Structure (SIGPLAN Notices, vol 11 (1976), special issue) pp 66-68.