CORRECTNESS OF DATA REPRESENTATIONS: POINTERS IN HIGH LEVEL LANGUAGES

(Extended Abstract)

by

D.M. Berry[†*]
Z. Erlich[*]
C.J. Lucena[ζ§]

1. INTRODUCTION

At present, there is considerable debate whether, in light of what is being learned about construction of reliable software, pointers are desirable in high level programming languages. One side [Hoa73, Hoa75] maintains that

1) Pointers are like the goto in that they are an invitation to create spaghetti in one's program [Hoa75].
2) In some languages, e.g., PL/1 [Wlk71], the use of pointers can lead to serious type violations, for example, the compiler believes that a given pointer will be pointing to an integer when in fact, it will be pointing to a real.
3) The indiscriminate use of pointers may confound the attempts of hardware pipelining and use of cache memory to speed up computations [Hoa73, Hoa75].
4) A pointer may be left dangling, that is, a pointer may point to a variable or other datum which has been deallocated [Bry71, CDMPS73, Bry74].

The other side [BEL74, Luc75] feels that

1) Problems 2 and 4 can be solved by insisting that pointer types carry the type of the object pointed at [vWn75, Wir72] and that pointers point only to explicitly allocated cells which remain allocated until they are no longer accessible [Wgb70, Wir72].
2) Pointers are needed in an extensible data type construction facility to build so-called recursive types and types with an unbounded extent.

*Computer Science Dept., UCLA, Los Angeles, California, 90024.

§Depto. de Informática, Pontifícia Universidade Católica, Rio de Janeiro, RJ, BRASIL

†Supported [in part] by the U.S. Energy Research and Development Administration, Contract No. E(04-3)-34, PA214, and [in part] by the National Science Foundation, Grant No. DCR75-08659.

ζSupported by the U.S. Energy Research and Development Administration, Contract No. E(04-3)-34, PA 214.

The solution offered by the first group to the very real need for pointers is to have the use of pointers deduced by the compiler from recursive type definitions [Hoa73].

However, the other side points out that in such a scheme:

1) One uses disguised terms which really describe pointer behavior anyway, e.g., assignment by sharing vs. assignment by copy [Lis74] and identity vs. atomic objects [Ear73]. Why not call a spade a "spade"?
2) The programmer may not have sufficient control over the placement of pointers in his data structures to get the most efficient behavior.
3) The implicit pointers may be just as confounding to the pipelining and cacheing mechanisms provided by the hardware as are the explicit pointers provided by the programmer.

This paper proposes a compromise solution taking advantage of the cluster or class [LZ74, DMN70, Hoa72] abstract type construction facility.

A cluster [LZ74] defines an abstract data type in terms of another data structure serving as the representation of the abstract type and a set of operations, e.g., a stack may be defined as:

1) an array with a top of stack index plus the operations create, push, pop, top, and empty

or 2) a linked list with the operations create, push (cons), pop (cdr), top (car), and empty (null).

Only the operation names are accessible to the user of the cluster. The representation and the bodies of the operations, being hidden from the user, are the concern only of the implementor of the cluster. Ideally, he builds a cluster presenting the operations of the abstract type while choosing some representation that allows an efficient, correct implementation of the operations of the abstract type. The implementor may use any representation so long as he can prove that the operation bodies operating on the representation have the desired behavior as expressed, say, by axioms for the abstract type.

The user of a cluster implementing a particular abstract type, in showing that his program is correct, needs only to use the axioms describing

the operations of the abstract type.

The key point in our compromise is that pointers are permitted for building representations of abstract types and not permitted as abstractions themselves. It is thus proposed that pointers be provided under the following constraints:

1) They are fully typed, i.e., the type of a pointer carries the type of the datum pointed at.
2) They may point only to explicitly allocated storage which remains allocated until it is no longer accessible, and not to storage for variables.
3) They and their operations may be used only in a cluster to build representations.

The proposed scheme eliminates objections 2 and 4 (objection 3, as we have seen, is not really valid) and at the same time uses the natural hiding properties of a cluster to control the complexity of pointer use, thus at least partially alleviating objection 1. Presumably, clusters are "small" (at least in comparison to the whole programs that use them) and the use of a pointer is thus restricted to a "smaller" portion of the code than it might have been used in. It is thus easier to see what is happening. In addition, by the natural shielding of a cluster, one can be certain that code outside the cluster cannot affect or be affected by the pointers in the cluster except in ways explicitly permitted by the operations.

If our proposal is to have any merit, it must be that proving the correctness of a cluster involving the use of pointers is not so terribly difficult. In the report of which this paper is an extended abstract, we consider a particular pair of abstract types[Bry75],

sequences
and elements

which are such that

1) each sequence is an ordered list of zero or more elements,
2) an element is in no more than one sequence, and only once in that sequence,
3) an element contains an updatable value which can be changed without changing the element's membership and position in any sequence, and
4) an element can be inserted into and removed from a sequence at any position in the sequence without changing its value.

This abstract type is typical of systems' queues and the SIMULA 67 SIMSET class [DMN70].

The cluster defining these abstract types defines a sequence as a doubly linked list (for easy insertion and removal anywhere) and an element as a cell containing two possibly nil link pointers and an independently updatable value part.

We prove the correctness of the cluster by use of a slight modification of the technique for proving correctness of classes developed by Hoare [Hoa72, BEL74, Luc75, Lav75].

2. OVERVIEW OF BODY OF FULL REPORT

In the report, we first give axioms and rules of inference for the use of pointers and related data structures. We then describe an extension of Liskov and Zilles' clusters which permits the definition of more than one abstract type at once. Then we state the method of proving such a cluster correct. With the necessary groundwork laid, axioms are given for the abstract types and a cluster is given which implements them. Finally, the necessary lemmas for carrying out the proof are set up and a few representatives of these lemmas are proved.

3. OBSERVATIONS REGARDING DIFFICULTY OF PROOF

In carrying out the extended example, we found no particular difficulties in doing the proof that seemed to be due to the use of pointers. The large size of the proof seems to be a direct result of the large number of operators defined in the cluster. The major difficulty was in the mutual discovery of the invariant of the representation and the mapping from the representation to the abstraction. These two must be delicately balanced against each other, for not enough invariant implies too much mapping and not enough mapping implies too much invariant.

In yet another extended example [LSB75] in which no pointers were used, we found exactly the same areas of difficulty. The size of this proof appears to be the same function of the number of operators defined in the cluster.

It is our opinion that the difficulty in proving the correctness of a cluster stems more from the "distance" between the representation and the abstraction rather than from the use of any particular type as the representation.

4. PROPOSALS

4.1 Types vs. Constructors

As a prelude to our proposal, we must distinguish carefully between types and constructors. Each type represents a set of values all of which are operated on by a particular set of operations. Each constructor represents a set of types characterized by a common organization and set of operation schema; a constructor takes one or more types and/or values as parameters and yields a particular type or another constructor.

Basic Types and Constructors

A language will generally have a set of basic types and basic constructors provided as primitives in the language. The basic types include integer, real, boolean and character. Associated with each of these is a set of operations such as arithmetic operations, logical operations, and character operations. Also provided are operations on some of these types to others, such as comparison operations.

The basic constructors of almost all languages include the array constructor. Also appearing in many languages are the record (or structure), the union, and the pointer constructors. The parameters to these constructors and some of the usual operation schema associated with these constructors are listed in Figure 1 at the end of the paper. The operation schema are referred to as such

because they become bona-fide operations on n-tuples of types to a type only when the operation is provided with the parameters of the constructors used to construct the types operated on. For example, associated with the array constructor is the subscripting operation scheme. Subscripting can be applied to any array of any dimensionality and element type. However, given dimensionality n and element type m, the subscripting operation scheme becomes an operation on n-dimensional-arrays-of-m's by n-tuples of integers to m's.

$$\text{subscript}_{n,m}: \underline{\text{array}}\ (n,m) \times \underline{\text{int}}^n \rightarrow m$$

Type and Constructor Clusters

If a language provides clusters, then two kinds of clusters can be identified, type clusters and constructor clusters. A type cluster is a cluster with no parameters at all and a constructor cluster is a cluster with one or more type and/or value parameters. As an example, the stack cluster

```
cluster stack (element-type: type) is create,
        push, pop, top, empty;
        .
        .
        .
end stack;
```

is a constructor cluster because it takes an element type as a parameter. On the other hand, the cluster

```
cluster stack_of_int is create, push, pop,
        top, empty
   rep = stack (int)
   .
   .
   .
end stack_of_ints;
```

is a type cluster because it has no parameters at all.

In the case of the stack cluster, create, push, pop, top, and empty defined in the cluster are but operation schema which become operations when applied to specific stacks with specific element types. In the case of the stack_of_ints cluster, create, push, pop, top and empty are bona-fide operations.

4.2 Basic and Cluster Types and Constructors

We propose a language in which basic constructors and constructor clusters may be used only to define the implementation or rep of other clusters and in which the operation schema associated with these constructors may be used only within cluster bodies. Outside cluster bodies only basic types and type clusters may be used to declare variables and only operations defined for these types may be used in operations involving these variables.

Thus, there may be identified two levels of language within the language we propose. One is a high level outside-of-cluster language permitting use only of types, basic as well as cluster, and their associated operations. The other level, containing the high level language as a sublanguage, is a lower level in-cluster implementation language, permitting also the use of basic constructors and constructor clusters and their associated operation schema.

The basic types of the language should include at least the following:

1. integer	4. character
2. real	5. string
3. boolean	6. void

as in ALGOL 68 [vWn75]. The first four are obviously useful. The fifth, string, is useful as a basic type implementing unbounded length character strings because it is hard to fit its constants into the framework of other types or constructors (e.g., flexible arrays of characters do not directly support the usual character string constant surrounded by quotes). The last, void, is useful for maintaining a consistent type algebra for compile time type checking.

The usual set of operations should be provided along with a set of axioms describing the behavior of these operations.

We suggest that a large variety of basic constructors be provided to give the programmer many well-known implementation techniques for building his own constructor and type clusters efficiently. Specifically, at least the following should be included:

1. Fixed arrays	as in ALGOL 68
2. Flexible arrays	as in ALGOL 68
3. Structures or records	as in ALGOL 68 or PASCAL
4. Pointers	as in PASCAL
5. Unions	as in ALGOL 68
6. Sets	as in PASCAL
7. Files	as in PASCAL
8. Subranges	modified from PASCAL*

Associated with each of these are the usual set of operation schema:

1. subscripting, trimming, bound checks
2. subscripting, trimming, bound checks, concatenation
3. selection
4. indirection
5. uniting, type checking
6. union, intersection, difference, element of, etc.
7. in, out
8. type check, empty conversion to the type that the subrange is a subset of.

*It is suggested that the subrange constructor have as its only parameter, the type the subrange is a subrange of. The lower and upper bounds should be part of the value of a variable of the subrange type so that the check that a value is within the range can be done at run time.

Also associated with each of these are axiom schema describing the behavior of the operations on values of the types derivable from these constructors.

With the language suggested in this proposal:

1. The programmer cannot circumvent the abstract type mechanism. All types used in variable declarations other than basic types must be defined in type clusters.
2. A good variety of even "dangerous" constructors are available inside clusters for the purpose of building efficient implementations of the abstract data types. Presumably, it is safe to permit the dangerous types and their operations in the controlled environment in which operations are being defined.

The user of a type cluster need only be concerned with the axioms describing the behavior of the associated operations on the elements of the type. The implementor of a type need only prove that his implementation of the type behaves according to the axioms. This should be possible even if "dangerous" constructors are used because

1. There are axioms for these constructors [HW72].
2. The use of these constructors is well controlled.
3. The size of a cluster is usually small enough to deal with even messy axioms.

4.3 Compile Time Type Checking

With the exception of two selected constructors, union and subrange, to permit delaying type checking until run time, it is desirable that all checking be performed at compile time; compile time checks permit more efficient code to be generated. To maintain* compile time type checkability in the presence of clusters, it is suggested that all arguments to basic constructors and constructor clusters be either compile time checkable types and/or constant (compile time computable) values. For example, the array constructor of ALGOL 68 takes an element type parameter and an integer dimensionality parameter. Under this suggestion, to preserve* compile time checkability of types constructed with the array constructor, it is necessary that the element type be a compile time checkable type and the dimensionality be an integer constant.

Suppose a constructor has a value parameter which cannot reasonably be restricted to being compile time computable, e.g., it is desired that the bounds of arrays and subranges be computable from expressions at block entry (allocation) time (thereafter the bounds of that allocation do not change). It is suggested that these parameters be made part of the value of an element of the type rather than part of the type. This strategy is used for arrays in ALGOL 68, SNOBOL and Oregano [vWn75, GPP74, Bry74].

It should, therefore, be possible to include parameters in the create routine which are not provided by the type, e.g., as in the type cluster below:

```
cluster real_square_matrix is add, mult, invert,
                              rank,
    rep = array [,] real; ¢two dimensional array
                          of reals¢
    create = op (bound:int) cvt;
                return ([1: bound, 1: bound]
                        real);
             po;

    rank = op (m:cvt) int;
               return (upb(m,1)) ¢array operatio
                 returning 1st upper bound of m¢
           po;

    .
    .
    .
end real_square_matrix
```

Notice that the bounds of a real_square_matrix is made part of the array value that implements it.

5. CONCLUSIONS

In the current reexamination of the act of programming, the pointer has come under a bit of fire. The main objections to it are:

1) One may not be able to guarantee the type of the object pointed at.
2) One may not be able to guarantee the existence of the object pointed at.
3) The use of pointers can make a program unmanageable and difficult to comprehend.
4) The use of pointers can make a program difficult to prove correct.

Yet, because pointers are needed to be able to build realistically efficient implementation of abstractions, they cannot really be thrown out. Instead of trying to let them be implied by use of recursive types, we try to alleviate the problems directly.

The first two objections are easily taken care of by insisting on fully-typed pointers which point only to explicitly allocated heap cells (disjoint from variable cells) which remain allocated until they are no longer accessible. The other two objections are dealt with by providing a linguistic framework to properly control the use of pointers, to increase both manageability and provability.

The Liskov and Zilles cluster concept provides such a framework. We restrict pointers to being used only inside clusters for the purpose of building abstractions (rather than as abstractions themselves). Clusters shield the users of a cluster from using the pointers except in ways which are explicitly allowed by the operations and which are, thus, meaningful to the abstraction. In addition, the cluster collects all the code for building a single abstraction into one "small" module. Only the implementor of the cluster need

*Maintenance of compile time checkability of types means that if the types and the constructors used in the representation of a cluster are compile time checkable or maintain compile time checkability, then so will the type or constructor built by the cluster.

be concerned with the inner workings and pointer shuffling. These restrictions should make pointers more manageable, if only by insuring that all code involving pointers is in one place. The shielding property of clusters insures that anything proved about the insides of a cluster cannot affect or be affected by the outside world.

To demonstrate our claim of manageability and provability, in the full report, we define a pair of non-trivial abstract types, sequence(t) and element(t) and give a cluster using doubly linked lists with pointers to implement the types. There, we lay sufficient theoretical groundwork to perform a proof of correctness of the cluster. With this support for our claims, a more complete proposal is given for a language providing pointers and other "dangerous" types as tools for building abstractions correctly and efficiently.

6. BIBLIOGRAPHY

[Bry71] Berry, D.M., "Block Structure: Retention vs. Deletion," Proc. Third Annual ACM Symposium on Theory of Computing, (1971).

[Bry74] Berry, D.M., "On the Design and Specification of the Programming Language Oregano," Technical Report UCLA-ENG-7388, Computer Science Department, UCLA, (1974).

[BEL74] Berry, D.M., Z. Erlich and C.J. Lucena, "Structured Data Representation: Proposed Modifications to the Concept of Clusters," Modeling and Measurement Note #27, Computer Science Department, UCLA, (1974).

[CDMPS73] Chirica, L.M., T.A. Dreisbach, D.F. Martin, J.G. Peetz and A. Sorkin, "Two PARALLEL EULER Run Time Models: The Dangling Reference, Imposter Environment and Label Problems," Proceedings of the ACM Symposium on High Level Language Computer Architecture, SIGPLAN Notices 8:11, (1973).

[DMN70] Dahl, O.J., D. Myhrhaug, and K. Nygaard, Common Base Language, NCC Publication S-22, (1970).

[Ear73] Earley, J., "Relational Level Data Structures for Programming Languages," Acta Informatica 2, (1973).

[GPP74] Griswold, R.E., J.F. Poage and J.P. Polonsky, The SNOBOL 4 Programming Language, Prentice-Hall, (1974).

[Hoa72] Hoare, C.A.R., "Proof of Correctness of Data Representations," Acta Informatica 1, (1972).

[Hoa73] Hoare, C.A.R., "Recursive Data Structures," Stanford A.I. Lab Memo AIM-233, Stanford University, (1973).

[Hoa75] Hoare, C.A.R., "Data Reliability," Proceedings of the International Conference on Reliable Software, (1975).

[HW72] Hoare, C.A.R. and N. Wirth, An Axiomatic Definition of the Programming Language PASCAL, Eidg. Technisch Hochschule, Zürich, (1972).

[Lav75] Laventhal, M.S., "Verifying Programs which Operate on Data Structures," Proceedings of the International Conference on Reliable Software, (1975).

[Lis74] Liskov, B., "A Note on CLU," Computation Structures Group Memo 112, Project MAC, M.I.T., (1974).

[LZ74] Liskov, B., and S. Zilles, "Programming with Abstract Data Types," SIGPLAN Symposium on Very High Level Languages, SIGPLAN Notices 9:5, (1974).

[Luc75] Lucena, C.J., "On the Synthesis of Reliable Programs," Tech. Report UCLA-ENG-7505, Computer Science Dept., UCLA, (1975).

[LSB75] Lucena, C.J., D. Schwabe, and D.M. Berry, "Issues in Data Type Contruction," Tech. Rpt. Pontificia Universidade Católica, Depto. de Informática, (1975).

[vWn75] van Wijngaarden, A., B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintzoff, C.H. Lindsey, L.G.L.T. Meertens, and R.G. Fisker, "Revised Report on the Algorithmic Language ALGOL 68," Acta Informatica (to appear in 1975).

[Wlk71] Walk, K., "Modeling of Storage Properties of Higher Level Languages," Proc. ACM Symposium on Data Structures in Programming Languages, SIGPLAN Notices 6:2, (1971).

[Wgb70] Wegbreit, B., "Studies in Extensible Languages," Ph.D. Thesis, Harvard University, (1970).

[Wir72] Wirth, N., The Programming Language PASCAL (Revised Report), Eidg. Technische Hochschule, Zürich, (1972).

[Bry75] Berry, D. M., "Correctness of Data Representations: Pointers," Internal Memo 144, Computer Science Dept., UCLA (1975).

Figure 1

Constructor	Parameters	Operation Schema
Array	Integer dimensionality, element type	Subscripting
Structure	Component types & selectors	Selection
Union	Alternate types	Uniting, type checking
Pointer	Pointed-to type	Indirection