A Survey of Some Issues Concerning Abstract Data Types

Lawrence Flon

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pa. 15213

September 1974

This work was sponsored in part by the Advanced Research Projects Agency of the Office of the Secretary of Defense under contract F44620-73-c-0074 and in part by the National Science Foundation under grant GJ 32259.

Abstract

We attempt in this survey to define the notion of 'abstract data type' and discuss several important issues related to their use in high level languages, including encapsulation, exportation, parameterization, pointers, extent, coercion, assignment, initialization, and literals. The programming languages Algol 68, Simula 67, and PASCAL are chosen as a representative set of type-extendable languages and are compared and contrasted as to their treatment of these issues. Proposals for new languages or facilities made by Liskov, Wulf, and Morris are included in the discussion.

Introduction

ALGOL 60[9] provides as part of the language three so-called 'primitive types' for variables, namely **integer**, **real**, and **boolean**. In addition, the **array** concept is applicable to each of these types and is the only means of structuring groups of values. Also provided are a set of operations which can be used to manipulate primitive typed variables, among them +, **div**, and **not**. Looking back upon the design of ALGOL 60 there seems to be no reason to assume that these language-defined concepts form any sort of reasonable 'spanning set' for the space of desirable programs. In fact one finds that many ALGOL 60 implementations introduce further primitive types, the most common one being some sort of character valued type. Several implementations have seen fit to introduce further capabilities for structuring data, for example record classes[3].

It is interesting to note that while ALGOL 60 provides only a very limited mechanism for constructing non-primitive data structures, it provides a very powerful abstraction tool for the construction of complex operations, namely the **procedure**. Much of the effort which has been expended in the quest of a suitable successor for ALGOL 60 has been directed towards mechanisms for constructing complex data structures out of more primitive ones in an effort to develop data definition facilities which are as powerful as those for operator definition.

The purpose of this survey is three-fold. We have attempted to define the concept of abstract data type and motivate its use in programming. We have also tried to collect in one place many of the important issues concerning programming language design and view them in relation to abstract data types. These issues include encapsulation facilities, exportation, parameterization, pointers, extent, coercion, assignment, initialization, and literals. In addition, we discuss these issues as they are treated by some existing languages which provide facilities for defining abstract data types (ALGOL 68[12], SIMULA 67[1], and PASCAL[13]), and by some proposals for new languages or facilities (Liskov[7], Morris[8], and Wulf[14]).

What is a type?

Abstraction is one of the abilities we usually list when put to the task of defining or characterizing intelligence. What we mean is the ability to distinguish properties from the entities which possess them and the ability to group together entities possessing the same or similar properties while ignoring for the time being any differences among those entities which are not concerned with the properties we are interested in. The entities we are concerned with in programming languages are values. The properties of a set of values which cause us to think of them as belonging to the same type are sometimes rather arbitrary and problem dependent. Those that are not are usually provided by the language (such as **integer**, **real**, etc.), if not by definition then by some standard prelude.

It is not enough, however, to think of a type simply as a set of values, for in doing so we forget the reason we chose to group those values together at all. We need to manipulate the values and indeed the manipulations we are most likely to want are those which assume the common properties as given. Thus we associate with the set of values a set of operations which are applicable to such values. In ALGOL 60, the type **boolean** possesses the set of values {**false**, **true**} and the set of operations {**and**, **or**, **not**}. The types **integer** and **real** both possess arithmetic operations with the same names, but actually the operations for both are distinct (for example + for reals means 'ADD FLOATING POINT' while + for integers means 'ADD INTEGER'; they are two distinct machine operations).

Also associated with a type is a representation for values of that type, for without a representation we would have no means of describing the behavior of the operations we associate with that type. The representations of the primitive types of ALGOL 60 are implicit and we don't usually concern ourselves with them, but the machine operations are defined in terms of those representations. As we shall see, the specification of a representation also determines the set of values of a type since it is defined in terms of existing types. In those languages where we can specify a set of values explicitly, we usually leave the problem of representing those values to the translator.

Why types?

Types have been used in mathematics for quite a long time. They are used every time we write:

Let f be a function ...
Let S be the set of ...

We associate with each symbol (variable) some set of values and operations so that our equations satisfy the mathematical analogue of the physicists' dimensional analysis. Indeed, if it weren't for types we would be hard pressed to explain away Russell's paradox:

> Let S be the set of all sets which are not members of themselves.
>
> Is S a member of itself?

One sees immediately that if we accept the proposition about S then the question has no answer. If, however, we define the operation **member of** so that x **member of** y has meaning only if x is of type t and y is of type 'set of t', then the proposition

S = {x such that **not** (x **member of** x)}

has no meaning[5].

Here we see one of the most useful advantages of the use of types - that of type checking. Type checking in Russell's paradox prevents us from making a mistake and introducing a bug into a logical argument. In programming, type checking prevents us from introducing program bugs through the use of inappropriate operations. For example in an assembler language, or even in a high level language without type checking such as BLISS[15], we are not prevented from, say, adding two locations which contain numbers in character form, having forgotten that we must first convert them to the machine representation of numbers. This is one of the hardest type of bugs to find in a program - one whose effect may not be noticed until some time quite later in execution.

The intent here is not to promote languages with sophisticated type definition and checking facilities since this may introduce run time slowdown inappropriate to some applications. What is desirable is a programming design discipline which is used to write programs using concepts of type and then to successively refine the type definitions into efficient implementations on a given machine. Refinement in this way will force a programmer to do type checking and lead to fewer bugs. We would, of course, like to have a high level language with some typing facilities as our base machine, so that we do not have to concern ourselves with irrelevant details and to provide a further check on the validity of the operations we wish to perform.

Type definitions

In this section we are concerned with facilities available for user definition of new types. There are, in general, two possible ways to define a new type - either from scratch or in terms of previously defined types.

By defining a new type from scratch we really mean adding a 'scalar' type to the language. A scalar type is essentially an unstructured type, i.e. its values are denoted by literals (e.g. identifiers or numerals). Typical examples are **boolean** (whose values are **true** and **false**) and **integer** (whose values are 0, -1, +1, -2, +2, etc.). One particular subclass of scalar types is the class of 'enumeration' types. By enumeration type we mean a type whose values are characterized by listing a finite set of literals. The capability for defining scalar types is an extremely powerful abstraction tool, for it allows us to represent, for example, colors, days, compass directions, etc., without forcing us to impose some arbitrary mapping between (in the case of colors) the notions 'red', 'yellow', 'blue' and some language-defined scalar type such as **integer**. We are usually provided with several language-defined scalar types, the constants of those types being part of the syntax of the language. Primitive operations are also provided by the language for use with these types. Therefore to define a new scalar type consistent with the rest of the language we must specify the set of constants of that type and the associated primitive operations. With the exception of **boolean**, most language-defined scalar types have an ordering associated with them, so we need to at least have this capability for new scalar types. The only language under discussion which provides any capability for scalar type definition is PASCAL. The PASCAL statement

```
type color = (red,yellow,blue);
```

defines an enumeration type named 'color' as having the given ordered set of values. The language primitives **succ** and **pred** are functions which will map any scalar type (including those which are user defined) onto itself using the implicit ordering (e.g. **succ**(yellow) has the value blue). Each constant of a new enumeration type must be distinct from all other constants in the language (just as language-defined scalar constants are). PASCAL provides no direct facility for the definition of new primitive operations so one must rely on the general procedure mechanism if any are desired.

PASCAL syntax prevents the definition of non-enumeration scalar types, simply because each of the constants must be listed. There are at least two possible ways for a language to allow the definition of infinite cardinality scalar types. The first is in terms of existing infinite cardinality scalars, as in

```
type evenint = integer i such that i mod 2 = 0;
```

(the syntax is purely hypothetical). The second method would be the dynamic extension of the language syntax, as in

```
type time = <1..24> : <1..60> : <1..60>;
```

or

```
type unary = '1' orelse '1' <unary>;
```

Again, a language designer must consider the tradeoff between implementation efficiency and program clarity, but the above is certainly a valid technique for program design[14].

Defining a new type in terms of existing ones entails tasks similar to those discussed for enumeration types. The difference is that the range (set of possible values) is not specified explicitly, but rather a structure is described, the elements of which are values of known types. The rules for specifying a structure (i.e. the set of different structuring mechanisms and the ease of combining them) play an important part in the ability of a programmer to easily translate his abstractions into a finished program.

Consider what was probably the first data structure (and certainly the only data structure in ALGOL 60) the **array**. As defined in ALGOL 60, the array is a generator of mappings from some range of integers (or n-tuples) onto values of one of the scalar types. More generally, an array may be thought of as a mapping from any scalar type onto any type at all. For example, in PASCAL the declaration

A : **array**[**boolean**] **of** color;

creates a mapping A from {**false**,**true**} onto {red,yellow,blue}. As such there are two 'locations' (A[**false**] and A[**true**]) both of which can posses any of the color values.

Arrays in PASCAL must be of conceptually finite length. Therefore, if integers are used as indices they must belong to a specified 'subrange' of integers. Sometimes one might think it convenient to have a mapping from _every_ integer onto some type. It seems, however, that in those cases where the mapping is not purely a function of the index (i.e. we allow the values of the mapping to change with time as in normal arrays) what is really desired is some sort of associative memory (like the SNOBOL4 'table'). Also of interest is the fact that no **real** subranges are allowed in PASCAL, while one really has no trouble envisioning a fixed-point subrange of reals (e.g. 1.0 **to** 5.0 by 0.1).

Since the range may be any type, one can specify, for example, a 'vector of vectors', which may be a convenient way to think of a two dimensional array, i.e.

A : **array**[1..10] **of** **array**[1..5] **of** **real**;

Unfortunately, array bounds in PASCAL must be compile time constants, so there is no convenient way to represent, say, a vector of variable-length **real** vectors. In ALGOL 68, array bounds may be 'flexible', e.g.

```
flex [1:0] real A;
```

declares A to be an array of 0 elements (initially) whose lower bound is fixed at 1 but whose upper bound is variable. This allows the size of A to change, but only upon assignment of an entire array value, e.g.

```
A:=(4,7,6.34,-4.27,0);
```

We can now declare a vector of variable length **real** vectors, e.g.

```
[1:n] flex [1:0] real A;
```

which might be useful in representing triangular matrices.

A major disadvantage of ALGOL 68 arrays, however, is that unlike PASCAL the domain must be the integers. This is mostly a consequence of the inability to define new scalar types - all that is additionally ruled out is **bool** and **char**. However unfortunate it is that all indexing must be done with integers, it would take more than a trivial modification to the language to allow anything else. Advantages of ALGOL 68 arrays are the flexibility of bounds and the fact that they can be dynamically allocated.

The SIMULA 67 array is closest to the ALGOL 60 array, the only difference being the addition of some new scalar types to the base language which can be formed into array types, so that one is subject to the same problems of limited abstraction found in ALGOL 60.

Another commonly used data structure is the record, which is a means for grouping objects of different types just as an array is a means for grouping objects of the same type. One advantage of using records instead of several arrays is that they may be dynamically allocated, while the size of an array is, for practical purposes, fixed at declaration time. In PASCAL,

```
type person =  record   name:string;
                        age:integer;
                        sex:(male,female)
               end
```

represents values which consist of three fields. The first is of type string (where string is previously defined) and is called name. The second is an integer and the third is of an enumeration type having constants male and female. If John were a variable of type person, the fields would be referenced as John.name, John.age, and John.sex, the last of which might be assigned a value by a statement such as

```
John.sex := male;
```

The ALGOL 68 **struct** corresponds to the PASCAL **record**. For example, we might represent the above record as

```
mode person = struct(string name,int age,bool male);
```

Note that the lack of a scalar type definition facility forces an arbitrary choice of a mapping between the abstract concepts male and female, and some existing scalar type, in this case **bool**.

The fields of a structure are referenced similarly to PASCAL, with some different syntax, i.e. if John is of mode **person**, its fields are name **of** John, age **of** John, and male **of** John.

We might digress briefly here to discuss the various syntactic alternatives for accessing the fields of a record. If r is a record and f is a field contained in r, then some of the possible notations for f are:

```
r.f
f.r
f of r
r(f)
f(r)
f[r]
```

Knuth[6] feels that f(r) is the most natural notation, since it is read "f of r" as if f were a function and r a parameter. In fact, the analogy to a function call is valid, because f has a fixed significance while r is variable and the value of the 'function' is the attribute of r selected by f. The ALGOL 68 notation 'f **of** r' is close to this idea since it puts the emphasis on f rather than r. Multilevel qualification in ALGOL 68 loses some clarity to the functional notation, for example,

son **of** father **of** mother **of** John

vs.

son(father(mother(John)))

but it does serve to distinguish field accessing from an operation with possible side effects, making the interpretation a little more obvious to the reader.

The mechanism for type definition in SIMULA 67 is the **class**. A class is essentially an ALGOL block whose activation records are heap-allocated, and hence multiple copies may exist independently of one another. Identifiers (data and procedures) declared local to a class are then reallocated each time a class is instantiated. By virtue of the use of the block mechanism, the record is the standard data structure in SIMULA 67. The same record might be represented as

```
class person;
      begin
      text name;
      integer age;
      boolean male;
      end;
```

If John is **ref**(person) then the fields are indicated by John.name, etc. as in PASCAL.

In [5] Hoare describes another data type called the discriminated union. The value of a variable of a discriminated union type is any value of any of the types listed in the union, for example in Hoare's notation,

```
type number = (i:integer,r:real)
```

defines a new type called number. If x is of type number, then either x.i or x.r exists, but not both, and this can be changed dynamically.

In PASCAL, discriminated unions are implemented as part of the type **record**. A PASCAL record may possess what is called a 'variant' field (which incidentally must be the last field in the record, although nesting of records is allowed) which is best illustrated by this example taken from [13]:

```
type person =    record   name:string;
                          age:integer;

                          case sex:(male,female) of
                                male:(height,weight:integer);
                                female:(size:array[1..3] of integer)
                 end;
```

Each variable of type person has in common those fields called name, age, and sex. Those records for which sex has the value male have in addition two more fields, called height and weight respectively. Those records for which sex has the value female have instead one additional field which is an array of three integers called size. The fields are fixed once a given record is instantiated, so the structure of that record cannot change dynamically.

In ALGOL 68 the discriminated union is not part of a structure but rather a separate mode (this is much closer to Hoare's original idea). The above example could be written as

```
mode male = struct(int height,weight);
mode female = struct([1:3] int size);
mode sex = union(male,female);
mode person = struct(string name, int age, sex s);
```

In the structure person, s is a field which is either of mode **male** or mode **female**. The field sex can change dynamically (which seems to be becoming more fashionable).

We have seen that in the PASCAL record, we can explicitly access the tag field which indicates of what type the variant actually is. Here there is no available tag but we can discover this by means of a 'case conformity clause', for example if p is of mode **person**, then

```
case s of p in
      (male): . . .,
      (female): . . .
esac
```

enables one to invoke different operations depending upon the current structure of the field.

SIMULA 67 also provides a way to create type unions, through the class concatenation mechanism.

```
class person;
      begin
      text name;
      integer age;
      end;

person class male;
      begin
      integer height,weight;
      end;

person class female;
      begin
      integer array size[1:3];
      end;
```

Now, if p is of type ref(person) it may also reference any male or female objects. No dynamic changes among the members of the union are possible.

It seems equally easy in SIMULA 67 and PASCAL to create unions which are variants of records. It is also equally cumbersome in both to create unions of arbitrary types since we must essentially make those types into records first. ALGOL 68 has the simplest mechanism of the three for creating type unions.

PASCAL offers some additional language defined types which the other languages do not. These are the **file** (Hoare's sequence), the **set** (Hoare's powerset) and the subrange. A **file of** t is a type which can be thought of as a sequential tape file. There are operations **get** and **put** defined for files which read and write the current 'record' and operations for testing and resetting the 'read/write head'.

The type **set of** t represents values which are sets, each one a member of the powerset of the primitive type t. For example,

```
type primary = (red,yellow,blue);
type color = set of primary;
```

Some examples of possible values for variables of type color are [red,yellow] and [red,yellow,blue].

Another type, the subrange, consists of a contiguous subset of some previously defined primitive type. For example

```
type digit = 0..9;
type highfrequencycolor = red..yellow;
```

As previously mentioned, subranges of all scalar types except **real** are allowed. Subranges in PASCAL are regarded as representing a finite set of values, since they are potential indices for arrays. A more general use of subranges would be to merely give bounds on the possible values of a variable, which of course would allow subranges of reals. See [4] for a discussion of some problems associated with subrange types.

Operations

The way in which operations are associated with type definitions, and indeed the entire notion of 'encapsulation', is an important issue. Encapsulation refers to the collection of all information relevant to a particular implementation of an abstraction, in our case a particular type. Languages tend to fall into one of three categories in this respect - 1) operations are bound to the definition of a type, or 2) operations are bound to the instantiation of a type, or 3) operations are bound to neither but are part of the general procedure mechanism. Languages which fall into category three, of which PASCAL is one, seem to violate the idea of trying to keep the notion of type as abstract as possible. In PASCAL it takes programming discipline to keep all the notions pertaining to a given type in the same place in the source code. Also inherent to the notion of 'operation on a type' is the fact that these operations may produce values of that type. In PASCAL, functions can only return values of scalar types. Thus, one cannot write

```
A := MATMULT(B,C);
```

to mean multiply two matrices and assign the product to A. Instead one is forced to write

```
MATMULT(A,B,C);
```

which is poorly structured since it hides the fact that an assignment is being made to A.

SIMULA 67 provides a fairly convenient way to associate operations with type definitions, since procedures may be declared as part of a class in the same way as data. We can declare a class to represent complex numbers as follows:

```
class complex(re,im); real re,im;
    begin
    ref(complex) procedure add(x); ref(complex) x;
         begin
         <return sum of x and this complex
           as a new instance of complex>
         end;
    .
    . <other operations>
    .
    end
```

With this use of classes, SIMULA 67 falls into category two, since all attributes of a class are bound to a particular class instantiation, including procedures. The add procedure produces the sum of its parameter and the particular complex number it is bound to (a reference to which may be generated by the construct '**this** complex'). If x and y are both **ref**(complex), then x.add(y) and y.add(x) will both produce the desired result. The syntax almost makes it clear that the operation is binary, but this is a coincidence and is destroyed when multilevel referencing is used. This way to construct the class complex in SIMULA 67 suffers from the loss of the abstraction that '+' is an operation which is binary and applies to all complex numbers. What we have done is to define operations +a, +b, +c, ... which are unary and of which there is one for each instantiation of the class. We have been forced to refine our abstraction of complex numbers further than is intuitively necessary.

We can get around the problem by writing

```
class complex(re,im); real re,im;
     begin
     ref(complex) procedure add(x,y); ref(complex) x,y;
           begin
           <return sum of x and y as a
             new instance of complex>
           end
     .
     . <other operations>
     .
     end;
```

in which case x.add(x,y), y.add(x,y), and z.add(x,y) for any arbitrary z would all produce the same result. The arbitrariness of the qualifier for the operation is a direct result of having operations bound to instantiation instead of definition.

It is in fact possible, using the method of concatenation, to define a class in a way much closer to the previous description of type definitions, as in the following definition of the complex data type:

```
class cmplx;
     begin
     class complex(re,im); real re,im;
           comment the empty statement; ;
     ref(complex) procedure add(x,y); ref(complex) x,y;
           begin
           <return sum of x and y as a
             new instance of complex>
           end
     end of cmplx;
        .
        .
        .
cmplx begin
        .
        .
        .
     end;
```

In the block prefixed by cmplx, the procedure named add need not be qualified by a reference. New instances of the class complex may also be created in this block, since all declarations local to cmplx are also known in it. What we have done in this example is to distinguish two different uses of the class mechanism - one as a structuring mechanism (as in complex above), and the other as an encapsulation mechanism (as in cmplx above). With this method of type definitions, SIMULA 67 falls into category one described previously.

ALGOL 68 provides for the definition of new infix binary (or unary) operators. The operators are generic in the sense that their definitions may change depending upon the types of their operands. As such, ALGOL 68 almost falls into the category wherein operations are bound to type definitions. Unfortunately, there is no way to prevent an operation (or procedure) which is used by two distinct operations from being used outside those operations since the operations do not have their scope restricted by the mode definition.

Both the Liskov[7] and Wulf[14] proposals associate operations directly with type definitions. This seems more in keeping with the abstract nature of a type.

Exportation

One of the functions of a type definition should be to hide from the user the way in which that particular abstraction is implemented. For example, in the case of a stack, it should not be possible to alter the stack pointer in any way other than through the use of a push or pop operation. Names which are known from outside of a type definition are said to be 'exported' from the definition. The ability to control which names can be exported is an important one in view of the work of Parnas[10] on information hiding (i.e. the 'need to know' principle).

Suppose operations A and B, which are associated with type t, both require the services of an implementation dependent operation C whose improper use could invalidate the type t data structure. We would like the names A and B known to anyone using t but C should be known only to A and B. In PASCAL and ALGOL 68, there is no chance for accomplishing this since operation definitions are not made in a scope local to type definitions. In SIMULA 67, although operations on a class are defined local to the same block as the data, their names are known to anyone possessing a reference to that class.

Although none of the existing languages possess the desired features, several good proposals have been made. In the function clusters described by Liskov[7], a type (or cluster) consists of several parts, including a representation, operations, and a list of those operations whose names are to be known outside the scope of the cluster definition.

Morris[8] proposes (in the context of ALGOL 60) a bracket pair {**module**,**end**} which can contain entire programs and can appear as a declaration in another program. Any notions associated with a particular type are wholly contained within a particular module, but names declared within a module and prefixed with a '+' are also known (but not writable) in the scope of the block which declares the module. This is a more general mechanism for exportation than Liskov's wherein only operation names may be exported.

In the ALPHARD language proposed by Wulf[14], exportation is even more general. One may not only export specific names from a type definition, but may also specify the rights for accessing those names (whereas Morris fixes them at read/execute).

Types vs. Structures

One largely unsettled issue in the literature is the difference, if any, between the notions of type and structure. For example, it is usually asserted that an array is a structure, while a matrix is a type, and there is some amount of intuitive support for this idea even with the fuzzy definitions of type and structure. One attempt to make the difference more precise is to say that types have operations associated with them while structures do not. A counterexample is the stack, which is intuitively a structure and yet has the operations push and pop associated with it.

It seems that the major difference is that we regard a structure as a collection of some number of 'atomic' entities, in that the operations we provide do not particularly care about the types of those entities. There does not seem to be enough motivation, however, to distinguish between types and structures at the language level.

Parameterization

The ability to parameterize a type definition is a very important language feature. Since abstraction generally means the ignoring of irrelevant differences among a set of objects, parameterization gives us a mechanism for explicitly stating what kinds of differences we choose to ignore and then concentrating on the similarities.

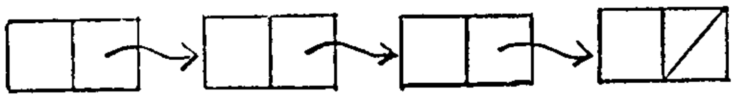
In PASCAL, for example, types cannot be parameterized, and the type definitions for a real stack and an integer stack would have to be distinct and would cause much repetition of code. Not even ordinary variables may be used as parameters, so it is impossible even to define a type 'n-vector'. Likewise in ALGOL 68, mode definitions cannot be parameterized. In SIMULA 67 classes may take parameters, in which case they represent initial values when the class is instantiated. They are only ALGOL 60 type parameters though, so types local to a class must be fixed. This at least allows for the definition of 'n-vector'.

Function clusters, as described by Liskov, may be parameterized in a general way allowing for both fixed type variables and those of type **type**. Liskov refers to parameterized types as type generators because they represent classes of types, each parameter representing some dimension of variability. The class of possible types is successively narrowed as each of the parameters becomes bound to a particular value, finally resulting in a single type. The mechanism is very powerful and an extremely useful abstraction tool. ALPHARD proposes a macro-like type definition mechanism called a 'form' which can take any kind of parameters in the same way as BLISS macro definitions.

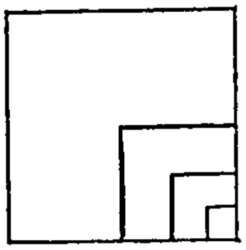
Pointers

There have always been two views as to the function of pointers in programming. There are those who believe that pointers are separate entities, whose values are locations or 'names' of other entities, a formalization of the hardware address concept. There are also those who believe that a pointer is merely an abstraction which enables one to think of a (possibly) more complex data structure as a single object. The difference is a conceptual one, and can be illustrated by the following two pictorial representations of a list:

(1)



(2)



In the first case, each list item possesses a field which indicates the location of the next item. In the second case, each list item possesses a field which actually is the next item (imagine a television camera focussed upon a screen which is receiving its image). The two representations are obviously equivalent. Consider, however, the definition of the second list as a recursive data structure:

```
type list(x) = union(null,listitem(x));
type listitem(x) = record item:x; nextitem:list(x) end;
```

Notice that the type of nextitem is fixed. In order to carry this over to the linear representation, we see that we must restrict pointers to reference only objects of a fixed type.

In some sense, the unrestricted use of pointers in a data structure is rather analagous to the unrestricted use of **go to**'s in a program. Just as a proliferation of **go to**'s can obscure the control flow of a program, too many pointers in a data structure will make the processing of that data structure arbitrarily complex and wordy. In addition, it should always be obvious (from the point of view of facilitating program proof) exactly where a **go to** will pass control. (Down with label variables!) For exactly the same reason, the type of the object referenced by a pointer should always be obvious. Languages like PL/I, which allow pointers to contain any arbitrary machine address, make it extremely easy to introduce bugs into a program. It is for this reason that most newer languages which provide pointer facilities, including ALGOL 68, PASCAL, and SIMULA 67, restrict them to a fixed type.

Pointers in PASCAL and SIMULA 67 are restricted to one level of indirection only. In SIMULA 67, one could have a type **ref**(complex) but **ref**(**ref** complex) is not allowed. This is because references must be made to classes. Multilevel referencing can only be done by imbedding references in the body of a class. In PASCAL, the type ↑t is valid for any type t but ↑↑t is not. Again, multilevel referencing must be done by imbedding pointers in the structure of t. There seems to be no need for explicit multilevel references. In ALGOL 68, unrestricted multilevel referencing is allowed, i.e. **ref real**, **ref ref real**, etc.

The mechanism by which pointers are transformed into values by a language is known as '**dereferencing**', a form of coercion (discussed later). Consider the statement

```
p := q;
```

Depending upon the types of p and q, there are several possible interpretations. Let '@' be a dereferencing operator (i.e. if p points to j, then @p is the same as j) and '#' be a referencing operation (i.e. if p points to j, then p is the same as #j). The following table indicates the possible actions a language might take to perform the assignment:

		type of p		
		t	ref t	ref ref t . . .
type of q	t	p←q	p←#q @p←q	p←##q @p←#q @@p←q
	ref t	p←@q	p←q @p←@q	p←#q @p←q @@p←@q
	ref ref t . . .	p←@@q	p←@q @p←@@q	p←q @p←@q @@p←@@q

In addition, there is the possibility of a syntax error in each case. It has been assumed that an assignment requires both sides to be of the same type.

The table indicates the numerous possibilities that exist for carrying out the assignment. Most languages take the view that the left hand side of an assignment, as an entity, acquires a new value. This would eliminate all possibilities which require dereferencing p first. This again brings us back to the two views of pointers, since if we regard a pointer as being the object it points to, we might well want to dereference it before assigning to it.

In PASCAL, there is no coercion of pointers at all. The only way to assign a value to a pointer is either by means of a generator (discussed a little further on) or by copying the value of another pointer of the same type (p←q in the above table, first two rows and columns only). Anything else is a syntax error.

In SIMULA 67, there is again no coercion of pointers, but there is the additional problem of assignment of pointers to different classes. This is legal only if either p's class includes q's class, or is a subclass of q's class and the value of q is a reference to a subclass of p's class.

In ALGOL 68, there is no implicit up-level referencing. The semantic rule used in an assignment is that q must be either one level dereferenced from p or must be dereferencable to that point (this is explained in the following paragraphs). The pointer coercion table for an assignment in ALGOL 68 is then:

		type of p		
		t	**ref** t	**ref ref** t . . .
type	t	**error**	p←q	**error**
of	**ref** t	**error**	p←@q	p←q
q	**ref ref** t	**error**	p←@@q	p←@q
	. . .			

An interesting question which arises is, why does ALGOL 68 allow multilevel referencing? One line of reasoning goes as follows:

First of all, a variable in ALGOL 68 is always of type **ref** x. For example, the declaration

```
real x;
```

is an abbreviation for the declaration

```
ref real x = loc real;
```

where **loc real** is a 'generator' which causes a location for containing reals to be created, and returns a reference to that location. The '=' causes the name x to be bound to the value of the expression on the right, making x a **ref real**. The reason for all this is to protect the declaration of constants. For example,

```
real pi = 3.14159;
```

declares the name 'pi' to be bound to the value 3.14159. Hence we cannot write

```
pi := 2.71828;
```

since both pi and 2.71828 are of mode **real** and under the rule for assignment this is illegal. We can write

```
x := 2.71828;
```

since x is **ref real** and 2.1828 is **real**.

Now consider the declaration

```
ref real q;
```

This is equivalent (as before) to

```
ref ref real q = loc ref real;
```

Therefore it is necessary to allow for at least **ref ref**'s. Since this is no easier than implementing **ref ref ref** etc., the obvious generalization was made. It seems, however, that there is no real need for indirection higher than level 2, just as there is no need for indirection higher than one level in PASCAL and SIMULA 67.

The generator **loc** causes allocation on the stack. As such the extent of the new storage is the lifetime of the block in which it is used. This is the same as ALGOL 60 local variables. Problems arise when pointers of more global scope refer to these variables. What happens when the storage goes away? When this happens it is most likely an error. In view of the work on information hiding it seems reasonable to prevent pointers from referencing the local storage of interior blocks. In PL/I, it is possible for a pointer to reference storage which no longer exists. ALGOL 68 enforces the rule that reference variables can never be assigned values which point to storage whose lifetime is potentially shorter than that of the reference variable itself. For example, the following is illegal in ALGOL 68:

```
begin
ref int r;
     begin
     int j = 2;
     r := j
     end
 .
 .
 .
end
```

While the above example is easily statically type checked by a compiler, consider the following:

```
begin
ref int ii;
      begin
      ref ref int jj =
             if (bool b := read(b); b) then ii
             else loc ref int;
      int i;
      jj := i
      end
  .
  .
  .
end
```

The validity of the assignment (jj:=i) is dependent upon the particular data value seen by the **read**. If this value is **false**, jj will be bound to a **loc ref int**, which has the same scope as i. If, however, the value read is **true**, jj will be bound to ii and the assignment becomes illegal. Therefore we see that in order to enforce the safe reference rule, and at the same time allow the full generality of execution dependent declarations, ALGOL 68 implementations must pay the price of run-time scope checking. This entails association of a variable and its scope (in practice, display level) and comparison of scopes before assignment. A smart compiler can easily keep the number of comparisons to a minimum, since most assignments could not possibly violate the rule.

In PASCAL, pointers may aquire new values only by means of a 'heap' generator, which allocates storage from a pool instead of a stack. The extent of this storage is until there are no more pointers referencing it (by means of garbage collection with reference counts). The PASCAL generator, incidentally, only applies to records. SIMULA 67 is very similar to PASCAL in that generation is from a heap and only applies to classes. ALGOL 68 has two types of generators - **loc** as previously described, and **heap**. The declaration

ref real x = **heap real**;

assigns to x the location of a real whose extent is determined by a reference count. This is of course not the same as the ALGOL 60 **own** concept because the ALGOL 68 declaration will be executed (and a new location created) every time it is encountered.

Coercion

Coercion is the term applied to the general task of converting one type into another, usually in relation to those conversions done implicitly by a language translator when it finds an object of a different type than it expects. The concept dates back to early FORTRAN when mixed mode arithmetic (i.e. integers and reals) was first allowed. PL/I is probably the definitive example of a language with default coercions between almost any pair of types (e.g. **char->real**, **bit->char**, etc.), most of which fall into two particular categories called 'widening' and (for the sake of symmetry) 'narrowing'. Widening refers to those coercions for which every value in the domain type has a corresponding value in the range type (e.g. **integer->real**). Narrowing refers to partial mappings (e.g. **real->integer**). PL/I syntactically allows both widening and narrowing, providing run-time error signals in the cases where there is no obvious range element. For example, narrowing a **real** to an **integer** is done by truncation (some languages perform rounding, of course), but some character strings (like '127') have obvious integer counterparts, while others (like '1a2bc') do not.

Most languages define coercions only for scalar types. It is unreasonable to expect a language to provide coercions between arbitrary types, for they may have complex structures. This type of coercion, since it represents non-trivial program actions, should be programmed explicitly by coding 'transfer' functions. The coercions PASCAL allows are:

1) widening of integers to reals
2) widening of subranges
3) narrowing of subranges

Only the third case must be checked at run-time. The view taken by PASCAL concerning not allowing narrowing (except (3)) is one taken by many lately (in sharp contrast to FORTRAN and PL/I). The reason is that narrowing not only forces run-time checking, but it also allows for many potential program bugs (e.g. the machine rounds instead of truncates and you get the wrong array element, or you find a statement that fails for only certain input). Unfortunately, PASCAL violates this idea somewhat by allowing the narrowing of a type to a subrange of that type, which it is forced to do as one of the consequences of the confusion over the treatment of subranges as types. If it did not allow this narrowing, then

```
n:1..10;

n:=5;
```

would be illegal, since 5 is of type **integer**.

SIMULA 67, since it is largely based on ALGOL 60, allows only the (**integer** <---> **real**) coercions of its ancestor, plus the coercion of references to different classes mentioned in the previous section.

ALGOL 68 treats coercion in a much more formal manner than any other existing language. It defines six different types of coercion. We have already discussed dereferencing in the previous section. ALGOL 68 allows widening but never implicit narrowing. Some examples of widening in ALGOL 68 are **integer->real, real->complex, char->string.** Another form of coercion, 'deproceduring', refers to the action taken when a procedure name is found where a value is expected, for example

```
x:=random;
```

where <u>random</u> is a **real**-delivering function and x is **real**. This is, of course, an old ALGOL 60 concept which applies to parameterless procedures, but in ALGOL 60 there are no procedure variables. In ALGOL 68 it is possible to have variables of type **proc**, hence the explicit definition of this type of coercion.

Coercions are applied not only in expressions but also in mapping actual parameters onto the formals. Interestingly enough, ALGOL 68 provides call by value, reference, and name, all through one mechanism. If a procedure expecting a **real** is passed a **real** value (or something coercible into a real value), then the actual parameter is copied into the location generated by the formal parameter declaration, so no assignment within the procedure body will affect the actual parameter (call by value). If a **real** variable is passed where a **ref real** is expected, then the <u>name</u> of the actual parameter is passed, which provides call by reference. If call by name is desired, a **proc real** routine text can be passed where a **proc real** is expected, and the actual parameter is deprocedured (in the context of the call site) when referenced, which is exactly the ALGOL 60 call by name mechanism.

The original ALGOL 68 report described a form of coercion called 'proceduring' which was used to form procedures out of program text being assigned to a **proc** variable. It was then possible to pass _an expression only_ as a parameter, and if the formal parameter was a procedure the expression would be procedured in the context of the call site. This was a particularly neat implementation of call by name. In the revised report, proceduring has beem eliminated due to some syntactic ambiguities caused by its general use outside of parameter binding. This has forced the method described above to now be used to achieve call by name.

Another ALGOL 68 coercion, called 'rowing', is provided in order to allow creation of a multiple value from a single element. The term applies to the coercion which transforms something of type x into something of type row-of-x. For example,

```
[ ] real A = 1.414;
```

This declaration specifies A as a 'row-of-reals' which is bound to the _row_ (of one element) 1.414. The upper and lower bounds are both set to one in this case. Rowing should not be confused with the array operations of, say, PL/I, where a constant added to a vector results in addition to every element of the vector. Two other coercions which are peculiar to ALGOL 68 are 'uniting' and 'voiding'. Uniting applies to the meta-coercion of a mode into a union of which that mode is a member, as in

```
mode intbool = union(int,bool);
intbool x;
x:=true;
```

The last coercion defined by ALGOL 68 is known as voiding, and refers to the discarding of the value of an expression. For example, in the following piece of program,

```
(real x; int i; . . . x:=i; . . .)
```

the assignment would be evaluated as follows:

1) dereference i
2) widen this value to real
3) assign this value to x
4) void the value

In addition to the implicit coercions which take place, the programmer can invoke coercions on his own by means of a 'cast'. A cast is a construct which serves to <u>force</u> a coercion in a context in which it would not normally take place. The syntax of a cast is

```
<mode> ( <value> )
```

For example,

```
real(2)
```

will produce the real value 2.0 from the integer 2. Suppose x and y are both of mode **ref ref real**, and we wish to decide if their values are the same. There is a boolean operator ':=:' which is the identity relation between two names. The expression

```
x :=: y
```

will not give us the desired result because the names x and y are not what we wish to compare. To compare the names which are <u>referenced</u> by x and y, we can write

```
(ref real(x)) :=: (ref real(y))
```

which will force dereferencing first.

<u>Literals</u>

The denotation of literals is an important issue with respect to abstract data types. If we have the capability to regard a variable as being of some abstract type, then it is important that we be able to assign to that variable a value at the same level of abstraction. If we are forced to construct a value through several program actions, we lose the abstraction of assignment to the variable as an entity. For example, in PASCAL, to assign a value to a record variable the values of the fields must be assigned one at a time:

```
type person =  record   name:string;
                        age:integer;
                        sex:(male,female)
               end;

p : person;

with p do
     name:='evelyn';
     age:=23;
     sex:=female
     end
```

If p is to represent an object which is a person, then we should be able to assign to it an object which is a 'constant' person, as in

```
p:=('evelyn',23,female);
```

In the latter case we see that the abstract assignment is represented as a single statement. The statement makes it clear that p is acquiring a new value, not just being selectively updated.

Both SIMULA 67 and PASCAL are greatly lacking in facilities for denoting literals. ALGOL 68 provides most of the desirable facilities for denoting literals of arbitrary modes. For example,

```
[1:5] int A;
A:=(7,12,-3,4,7);
```

Here A is assigned the vector as written. In most other languages it would be necessary to assign each element individually in separate statements, unless there was an algorithm for computing A[i+1] from A[i] in which case one would use a loop.

The lack of a good facility for denoting literals can cause much unnecessary coding and annoyance to the programmer.

Initialization

Initialization is another extremely important issue concerning abstract data types. The lack of initialization caused much grief concerning **own** variables in ALGOL 60, since the only way to assign a first-time-only value to an **own** variable is to use a global flag, which defeats the purpose of **own** variables entirely (Tennent[11] describes a mechanism for improving the **own** variable concept, both with regard to scope and to initialization). Initialization means more than a version of the FORTRAN **DATA** statement when abstract data types are concerned, since the creation of a new structure may require many actions.

PASCAL provides no initialization facilities at all which, in conjunction with its lack of capability for denoting literals, leads to programs whose structure is clouded by much too highly refined statements, as in the example on page 35. ALGOL 68, while providing good facilities for value initialization, lacks features for more complex actions.

The SIMULA 67 class provides the best existing mechanism for initialization. When a class is instantiated, it is executed. Recall that a class is similar to an ALGOL 60 block, and therefore contains not only declarations but also code. The code can initialize locals, build data structures, read and write information, and in fact do anything that one wishes to associate with initialization of that particular abstraction.

The Liskov proposal[7] for encapsulation provides, for each cluster definition, a section of code which is to be executed whenever an object of that cluster is created. It is admittedly based on the SIMULA 67 idea, and, for the purpose of initialization, is a little neater in that the SIMULA 67 class is really a coroutine instead of just a data type, and the way in which the body of a class is executed can be quite complicated.

The Wulf proposal[14] provides essentially the same facility and in addition recognizes two types of initialization, one which initializes things with respect to the entire class of instantiations of a particular type and one which provides local initialization. This seems to be a useful distinction which allows, for example, the initialization of a pool of storage from which all members of that type can draw. Wulf also recognizes a cleanup problem, and provides the same facilities upon destruction of an object or class of objects.

Assignment

Assignment is the means by which variables can take on new values. Classically this has meant that the statement

```
p:=q;
```

would cause the value of q to be copied and then given to p. Another possibility, however, is that the value possessed by q is shared with p. One is not highly motivated to use sharing when dealing with primitive types, since the copy operation in that case is extremely cheap. Consider, however, the following example. Suppose that the type matrix(m,n) is defined with operation MATMULT(A,B) defined as the matrix multiplication of A and B.

```
A,B,C:matrix(10,10);
read(A,B);
C:=MATMULT(A,B);
```

We will assume that MATMULT creates a new matrix to contain the value of A times B and returns this value. If assignment by copy is used, this 100-element value must be copied into C, even though the original reference to the new matrix is irretrievably lost once the assignment is performed.

The standard response to this example is "use pointers". Indeed, if we had MATMULT return a value of type **ref** matrix and if A, B, and C were also of type **ref** matrix, then assignment by copy would do just what we want. We have two reactions against this solution. First, the analogy between pointers and **go to**'s which has been established makes us wary of introducing pointers when there seems no neater way, just as being forced into using a **go to** makes us look for a better control construct. Second, if we do use pointers we are then forced to a level of refinement of abstraction which seems irrelevant. The abstract operation of matrix multiplication delivers a matrix as result, and to have to write this any other way seems unfortunate. Of course, we are not trying to make a case for assignment by sharing alone, since it too has many problems associated with it, most notably the fact that a selective update to a structure whose value is shared will be reflected in all references to that structure. Both forms of assignment have their uses and neither is sufficient, and therefore any language designed to facilitate structured programming should provide both forms to the programmer. For example,

```
p←q        means 'copy'
p:q        means 'share'
```

Assignment by sharing would be defined for all types, assignment by copy only for those types which provide an operation called **copy** in their definitions. It would also be necessary to define an equality operation to determine if two values of the same type (which are not identically the same) are equal.

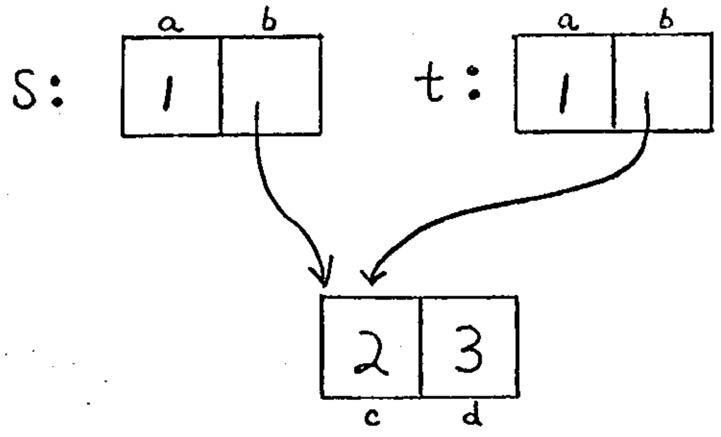
The default form of assignment in languages like PASCAL and ALGOL 68 is sometimes known as '1-level copy'. It implies that the top level of a data structure (i.e. some contiguous block of storage) is copied. For example, in PASCAL the statements

```
type pair = record c:integer; d:integer end;
s,t : record a:integer; b:pair end;

s.a:=1;
s.b.c:=2;
s.b.d:=3;

t:=s;
```

will create a structure like:



so that now the statement

```
t.a:=7;
```

will not affect s, while the statement

```
t.b.c:=7;
```

will. This type of assignment is a hybrid one, since it is partially by sharing and partially by copy. It is another 'dangerous' feature which, although not necessarily deserving of elimination, should at least not be the default.

Thus far we have discussed some of the more global issues concerning assignment. Let us now consider some of the local issues. Generally we regard an assignment as evaluating some expression on the right hand side and storing that result in the location specified by the left hand side. A simple generalization allows the left hand side to be a location-delivering expression. Languages which allow this generality include BLISS and ALGOL 68. For example, a legal sequence in ALGOL 68 is

```
int i,j,k;
bool b;
   .
   .
   .
(if b then i else j) := k;
```

or even

```
int i,j,k;
bool b;
proc ref int p =
     ref int : if b then i else j;
   .
   .
   .
p := k;
```

The fact that the left hand side of an assignment can be a function designator implies that the ALGOL 60 method for returning a value from a procedure (i.e. assigning a value to the procedure name) would cause ambiguity in ALGOL 68. This problem was cleaned up in a general way by defining the value of a block to be the value of the last statement, and the value of a procedure to be the value of the procedure statement. The generalized left hand side concept is by no means regarded as an unquestionably good one. One argument against it relates to the problem of unexpected side effects caused by the evaluation of the destination expression. Even this might have its usefulness, as in the case of a procedure which prints a message every time its 'own' variable is referenced, returning the name of that variable (this might be useful in program tracing).

Another argument against the generalized left hand side is that such a statement de-emphasizes the assignment itself, clouding it up in computation which could just as easily (and, perhaps, more clearly) be done in preceding statements.

Another important issue concerns the protection of variables. Often, variables are used to represent constant values and maintain understandable programs, as in

```
pi := 3.14159;
```

where pi is a variable likely to be known to much of the program, or even

```
sth := sin(theta);
```

where sth is local to a routine to which theta is a parameter, its value being constant throughout the execution of a particular call on that routine. The knowledge that a particular variable remains unchanged throughout some period of time is usually very helpful both to a programmer who is trying to prove some property of his program, and to an optimizing compiler which, for example, might avoid multiple copies of a constant parameter to a routine which calls itself recursively with that parameter.

Few languages provide reasonable facilities for the declaration of read-only constants. The PASCAL **constant** declaration provides only a very trivial mechanism which allows the definition of identifiers to represent literal numeric constants. This is too restrictive, as we would like constants to represent the results of arbitrarily complex computations. The BLISS **bind** mechanism is more useful in this respect, because it causes an identifier to be bound to an expression computed at block entry time. A language which provides type definition facilities, however, should also provide constant declaration facilities for arbitrary types. As we have already seen in our discussion of pointers, ALGOL 68 provides these facilities.

Summary

We have presented a view of abstract data types and discussed several programming issues with respect to data types as implemented or proposed in various languages. We have touched on many bases and hope we have stimulated the reader's interest in programming with abstract data types and provided him with a broad familiarity of many of the important issues in programming language design. Although some of the discussion was admittedly biased in certain directions, we have tried not to design a new language but rather to promote awareness of the complexity of such a task. Any negative remarks concerning existing languages could not detract from the value they have had in directing us toward discovering better programming facilities.

Acknowledgements

I am deeply indebted to Prof. A.N. Habermann for his direction and guidance during the months of preparation of this paper. I am grateful to Prof. P. Hibbard for his help with ALGOL 68 and to A. Lunde for his help with SIMULA 67. Thanks also to Prof.'s M. Shaw, A. Jones, and W. Wulf for their critical comments on the first draft.

Bibliography

1) Dahl, O.J. et al., Simula 67 Common Base Language, Norwegian Computing Center, Oslo (May 1968).

2) Dahl, O.J. and Hoare, C.A.R., Hierarchical Program Structures, in Structured Programming, Academic Press, London (1972).

3) Digital Equipment Corporation, PDP-10 ALGOL Reference Manual, Maynard, Mass. (1971).

4) Habermann, A.N., Critical Comments On the Programming Language PASCAL, Acta Informatica 3 (1973).

5) Hoare, C.A.R., Notes on Data Structuring, in Structured Programming, Academic Press, London (1972).

6) Knuth, D.E., The Art of Computer Programming, Vol. 1 (ch. 2), Addison-Wesley, Reading, Mass. (1968).

7) Liskov, B. and Zilles, S., An Approach to Abstraction, MIT Project MAC Computation Structures Group Memo 88, Cambridge, Mass. (Sept. 1973).

8) Morris, J.H., Types Are Not Sets, SIGPLAN Symposium on Principles of Programming Languages (Oct. 1973).

9) Naur, P. (ed.), Revised Report on the Algorithmic Language ALGOL 60, Comm. ACM 6, pp. 1-17 (1963).

10) Parnas, D.L., Information Distribution Aspects of Design Methodology, Proceedings of the IFIP Congress 71, Vol. 1 (1972).

11) Tennent, R.D., A Contribution to the Development of PASCAL-Like Languages, Department of Computing and Information Science, Queens University, Kingston, Ontario (April 1974).

12) van Wijngaarden, A. (ed.), Report on the Algorithmic Language ALGOL 68, Numerische Mathematik 14, pp. 79-218 (1969).

13) Wirth, N., The Programming Language PASCAL (Revised Report), Berichte der Fachgruppe Computer-Wissenschaften, Eidgenossische Technische Hochschule, Zurich (1972).

14) Wulf, W.A., ALPHARD: Towards a Language to Support Structured Programs, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa. (April 1974).

15) Wulf, W.A. et al., BLISS: A Language for Systems Programming, Comm. ACM 14, pp. 780-790 (1971).