Program Design With Abstract Data Types

Lawrence Flon

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

June 1975

This work was supported in part by the Defense Advance Research Projects Agency under contract F44620-73-C-0074, and in part by the National Science Foundation under grant DCR74-24573.

Abstract: This paper explores the use of abstract data types as a modularization and structuring technique in the design of programs. The concepts of type and type definition are discussed. Some data structuring mechanisms are generalized and several simple examples are presented. The examples increase in complexity and conclude with the design of a directory system, illustrating the power of data types as a design tool.

Keywords: Abstract data types, modularization, specification.

CR Categories: 4.10, 4.20, 4.30, 4.34

Contents

Introduction

This paper explores the use of abstract data types as a modularization and structuring technique in the design of programs, particularly larger programs including compilers and operating systems. The emphasis is on design. The particular language in which the program will be coded should be of relatively little importance in the design process. We are interested in a technique for constructing *specifications* which can be converted at another time (and possibly by others) into an *implementation*. Programming languages which provide data type facilities to varying degrees have been in existence since about 1967 when SIMULA-67 [1] appeared. Since then, there has been an ever increasing number of such languages, notably ALGOL-68 [7], PASCAL [8], and EL1 [6]. There are more advanced languages currently under development or proposed, including ALPHARD [9], CLU [3], and PASQUAL[5]. A useful bibliography of papers concerned with data types and programming languages may be found in [5].

Abstraction refers to the mental process which, when confronted with a set of objects, can distinguish some objects from others on the basis of what they have in common. What is "abstracted out" is a unifying theme, called the *type* of those objects. To pick a conventional example, the set { 7, 1.35, -6, 'A', 0.3E6 } contains three types of entities: integers, reals, and characters. These are types which just about every programming language provides. More abstractly, the set contains only two types: numbers and characters. Even more abstractly, it contains only one type - constant. When we prefix "data type" with "abstract" we mean an arbitrarily complex type which characterizes a certain kind of behavior.

The concept of using abstractions in programs to control complexity is certainly

not a new one. For very small programs it is sufficient to use a straightforward control flow through the program statements. Slightly larger to medium sized programs require the use of procedures to group individual statements in a logical manner, while a main program directs the flow of control among the various top-level procedures. When dealing with large programs it becomes necessary to group the individual procedures into modules. The process by which this should be done is not yet universally understood. This paper expresses the view that the use of data types as a modularization principle is the right approach.

We will begin by discussing what it means to define a new data type and will describe some basic data structuring mechanisms. Numerous examples are given to familiarize the reader with the notation, building up to the final example involving the design of a directory system.

1) Type Definitions

A type definition has two basic attributes:

1) functional specification: the behavior of a particular class of objects as seen by users of that class.
2) algorithmic specification: the means by which that behavior is accomplished.

The distinction between functional and algorithmic specification is most important. First, it allows programmers to construct implementations in parallel with one another without any more information in common than their mutual interfaces. Second, it ensures that a change in data structure or algorithm is localized as long as

the functional specifications remain unchanged. A type definition is really a module in the sense of the information boundaries discussed by Parnas [4]. It provides a clear design procedure for the modularization of a large program or system, where one has been previously lacking.

Language defined types such as **integer, real,** and **boolean** are called *scalar* types. The term scalar is intended to signify that values of such a type are considered to be indivisible. The scalar type **integer** characterizes single values taken from the range {-n,...,n}, in conjunction with a set of operations which may be applied to integers, such as addition, subtraction, multiplication, etc. As far as a user of type **integer** is concerned, he should not know about, and more importantly not rely on, the way in which integers are represented or the particular algorithms which implement integer arithmetic on his computer. Adhering to this principle makes program portability possible, since different machines may use different algorithms. We generalize these notions to abstract data types and say that a type definition consists of:

1) a representation for objects of the type, known only to the definition itself (algorithmic specification).

2) a set of operations which manipulate this representation, the names and calling sequences of which are known to users of the type (functional specification), but the algorithms of which are known only to the definition itself (algorithmic specification).

The binding of operations to particular types is vital to the description of behavior. It is not enough to present a picture of a control block as the description of a "process" in an operating system. What is necessary is an explanation of exactly

what can be done to a process, and specifically not how it is done. In fact control block formats should not be disseminated at all; rather only a description of valid operations which can be used on that data structure should be provided to users. One of the most important things gained from this is the "sanctity" of a data structure. Every data structure may be viewed as having an invariant relation which it is characterized by (e.g. the mathematical notions of "list" and "tree"). It is easier to verify the invariance of this relation if the relevant changes to the structure are made by a small number of "privileged" procedures. If you parcel out a data structure address and the right to modify arbitrary fields in arbitrary ways, it is possible (likely, in a brand new program) to find inconsistencies in the structure due to program bugs anywhere in the entire system. If all the information that is distributed is operation names, the scope of possible sources of error is greatly reduced to the implementation of those very operations.

The next sections present a set of tools useful for the description of data abstractions, and introduce some relatively intuitive syntax.

2) Enumeration Types

Often in the course of writing a program in a standard language, a programmer will find the need to express values of scalar types which are not provided by the language. For example, one might find an inventory program which, in order to identify the color of an object, uses integers to represent red=1, yellow=2, etc. When designing this program, it is not relevant to define the mapping between colors and integers, since this is merely an implementation detail. A mechanism is needed for defining a new type by exhibiting the set of constants which comprise its values. A

type so defined is called an *enumeration type*, the concept having been taken from PASCAL. An example of the definition of enumeration types might be:

type color = {red,yellow,blue,green} **end;**

type weekday = {monday,tuesday,wednesday,thursday,friday} **end;**

In order to declare variables as instances of particular types, we will use syntax of the form:

declare payday,dayoff:**weekday**, shade:**color;**

The declaration states that payday and dayoff are variables which may only possess values appropriate to type **weekday**, and that shade can only possess a value appropriate to type **color**. These variables behave as do more common scalar variables with respect to their appropriate constants. For example,

payday ← wednesday;

would be valid, but

payday ← red;

would be meaningless. Operations are defined for enumeration types as for example,

```
type color = {red,yellow,blue,green}

    op complement(c:color):color =
        case c of
            begin
            red:      green;
            yellow:   blue;
            green:    red;
            blue:     yellow
            end

    end;
```

The operations provided by a type are declared inside the scope of the definition in order to exhibit modularity. The brackets **type** and **end** delimit the definition. To use one of these operations in a program description, it is necessary to prefix the operation name with the type name in order to permit several types to have operations with the same name (and for readability). The complement operation takes a single parameter of type **color** and returns a value of the same type. For example, if the statements

```
shade←green;
shade←color.complement(shade);
```

were executed, shade would obtain the value red. Even though in most languages an implementation of these types will require the use of integers, nevertheless the irrelevance of the mapping may be preserved by declaring macros (or variables) for use in place of the integer constants.

3) Composite Types

Given the ability to define new scalar types, we must now be able to build upon those definitions to define more interesting types.

Consider as an example the definition of the type **complex**. We wish to represent numbers in the complex plane and provide arithmetic operations on them. We need to specify to users that the operations

```
add(w,z:complex):complex
mul(w,z:complex):complex
create(a,b:real):complex
```

will be available. Add will take two **complex** values and produce the **complex** sum. Mul will behave analogously. Create will form a new **complex** value equal to a+bi. In terms of algorithmic specification, we can represent a **complex** value by its real and imaginary parts. Consider the following definition:

```
type complex =
    declare r,i:real;

    op add(w,z:complex):complex =
        begin
        declare sum:complex;

        sum.r←w.r+z.r;
        sum.i←w.i+z.i;
        return sum
        end;
```

```
op mul(w,z:complex):complex =
      begin
      declare prod:complex;

      prod.r←w.r*z.r-w.i*z.i;
      prod.i←w.r*z.i+w.i*z.r;
      return prod
      end;

op create(a,b:real):complex =
      begin
      declare c:complex;

      c.r←a;  c.i←b
      return c
      end
end;
```

The body of the definition contains some variable declarations (r and i) and some operation definitions (add, mul, and create). When a variable of type **complex** is declared, e.g.

```
declare x:complex;
```

or dynamically allocated, e.g.

```
complex.create(1,2)
```

the variables inside the definition of complex are allocated, just as when an **integer** variable is declared the bit string by which it is represented is allocated. From outside the definition it appears that x is an atomic entity to be used with the operations provided, e.g.

```
complex.add(x,y).
```

From inside the type definition, as within, say, the microcode which implements integer arithmetic, the structure of x is visible. In this case the structure consists of the variables r and i. To access a particular structure, it is necessary to specify which structure by prefixing the variables with the name of a **complex** variable. For example, the use of w.r in the body of the add operation refers to the r part of the **complex** object w which was passed in as a parameter. The use of sum.i refers to the i part of the locally declared **complex** object sum. An implementation of this type in a language with dynamic storage allocation will be straightforward, i.e. it will be possible to return a pointer to a newly allocated structure representing the result of an operation. In other languages it may be necessary to write the operations as subroutines, passing the destination variable as an output parameter. In FORTRAN, storage allocation could be simulated by declaring an array of pairs, each index into the array representing a single **complex** value. The informal language in which these examples are presented assumes that functions can return values of arbitrary types.

An example of the use of type **complex** might be:

```
declare x:complex;
x←complex.add( complex.create(1,-2), complex.create(2,3) );
```

which would result in x acquiring the value 3+i.

Note that we were not forced to represent **complex** values in terms of their real and imaginary parts. We are free to re-design the type definition as long as we conform to the stated specifications of the operations. For example, we might decide to use polar coordinates to represent a **complex** value, if that made the operations more efficient. Although the create operation is still specified to produce the value

a+bi from parameters a and b, there is nothing to prevent it from internally converting to any other representation, and there is certainly nothing to prevent us from writing a different create operation which expects arguments with a new significance. e.g.

```
type complex =
    declare r,theta:real;
          .
          .
          .
    op create(a,b:real):complex =
          begin
          declare c:complex;

          c.r←sqrt(a*a+b*b);
          c.theta←arcsin(b/c.r);
          return c
          end

    end;
```

In addition to the allocation of the structure variables, when a variable of type **complex** (or any type) is declared, the create operation is implicitly invoked. This is important in order to allow the data structure invariant relation (if there is one) to be established. Although in this simple example this is unnecessary, it will become necessary later on.

4) Data Structuring

A comprehensive presentation of data structuring methods in relation to types may be found in [2]. We will discuss briefly some structuring mechanisms in order to have some syntactic base upon which to build.

4.1) Arrays

In the past, an *array* has generally been thought of as a table indexed by integers, as is the case with a matrix. A matrix, however, is a type, having certain operations such as multiplication and inversion defined for it. An array is only a structuring mechanism and might as well be generalized to a more powerful mapping than is possible with just integer domains. Think of the array as a mapping from any finite scalar type to any type at all. For example,

```
declare A:array[color] of integer;
```

declares a vector which is indexed by objects of type color and whose individual elements are integers. We could use such an array to represent the frequencies of the various colors of light, e.g. A[red] = 6485.

A two dimensional array can be modeled as a vector of vectors, e.g.

```
declare A:array [1:10] of array[1:5] of real;
```

so that A[i] denotes a row (or column, depending upon the interpretation).

4.2) Records

Another means of grouping values is the *record* structure. The local data of any type declaration can be thought of as a record. The variable names of the local data denote the *fields* of the record. These fields are accessed as previously noted, by prefixing the field name by the name of the complete object. While arrays are used to

group values of the same type, records are used to group values of potentially differing type. Often records are used to structure data in a hierarchical manner, as in PL/I or COBOL. For this facility we will use the notation

declare r:**record** (f_1:t_1,f_2:t_2,...,f_n:t_n);

to declare a variable r which has n subfields ($r.f_1$,$r.f_2$,...,$r.f_n$) of types (t_1,t_2,...,t_n) respectively. The t_i can be arbitrary, including other records. For example,

```
declare person:record   (name:string,
                         age:integer,
                         salary:record   (regular:real,
                                          overtime:real));
```

The subfields of this variable are person.name, person.age, person.salary.regular, and person.salary.overtime. It is possible to declare a table as an array of records, e.g.

declare T:**array**[1:100] **of record** (height,weight:**integer**);

in which case the fields would be accessed as T[i].height and T[i].weight. In general, however, if a record is to be used in any non-trivial way, it should be embedded in a type declaration, e.g.

declare T:**array**[1:100] **of medicalrec**;

where **medicalrec** defines the fields of the record as local data and provides operations on that data. Since it is not unlikely that the structure of **medicalrec** will change at

some point during system development, in this way we anticipate modifications by restricting their scope.

4.3) References

A reference value is a pointer to an object, with the restriction that reference variables can only reference objects of a single type. Typeless pointer variables are the cause of many program bugs since they are free to point to arbitrary addresses. References are much safer to use and make programs less devious in their logic. The reference concept is important with regard to the use of a dynamic storage allocation mechanism. We postulate the operations new and free for every data type, but make them available for use only inside of a type definition. This is because of the need for initialization of the data structure invariant relation discussed in section 1. From outside of a type definition, the create operation must be used.

If c is a reference to **complex**, i.e.

```
declare c:ref complex;
```

then, inside of type **complex**, the statement

```
c←new complex;
```

causes the dynamic allocation of a **complex** object, and assigns a reference to that new object to c. The statement

```
free c;
```

results in the storage referenced by c being released. All that we can do with the value of c is to assign it to other **ref complex** variables or pass it as a procedure argument, since as yet we have not defined any operations for reference types. In particular, we will not allow statements such as

```
c←c+2;
```

which are likely to introduce errors. More importantly, these statements violate the notion of type, since if the address contained in c is incremented by 2 it probably no longer is the address of a **complex** variable. We will define an operator for references called *dereferencing*, which converts a reference into the object it references. If c contains a valid reference, then c↑ is of type **complex**. For example, c↑.r is the r field of the **complex** object referenced by c. It is not legal to write c.r, since c is not of type **complex**. This imposes more structure on the program specifications and prevents many possible bugs. Our examples assume that ↑ applies to the reference immediately to its left. We will use the reference value **null** to indicate that no object is currently being referenced.

5) A Small Abstraction

With the tools just presented, we can begin to describe some simple abstract objects. Suppose we would like to model the concept of a "bag" full of data, in the sense that once a datum is inserted in the bag, it can only be removed by trial and error, since "reaching your hand" into the bag is done blindly. Assume that the bag is to hold integers.

We need to provide certain key operations, namely:

```
empty(b:bag):boolean
put(b:bag,i:integer)
take(b:bag):integer
```

and we would like to guarantee to the external world that a "taken" element is chosen at random. There are several ways to specify this type. One way is to use a vector to represent the state of the bag, although we have to place an upper limit on the size of the bag. Consider:

```
type bag =
    declare   A:array[1:100] of integer,
              top:integer;

    op create:bag =
         begin
         declare b:ref bag;

         b←new bag;
         b↑.top←0;
         return b↑
         end;
```

So far we have specified that a bag will consist of a vector of length 100 (A), and an integer (top) which we will use to indicate the current size. We have specified the create operation, since here it is necessary to perform initialization. The invariant relation of a bag is "the bag contains (top) elements" so we must set top to zero whenever a bag is created. If we had allowed the new operation to be used outside of the type definition, top could remain unitialized.

We can specify the empty and put operations easily:

```
op empty(b:bag):boolean = return (b.top=0);

op put(b:bag,integer) =
     if b.top=100 then error     comment overflow;
     else
          begin
          b.top←b.top+1;
          b.A[top]←i
          end;
```

Assuming a uniform random number generator, take can be specified as:

```
op take(b:bag):integer =
     if b.top=0 then error     comment underflow;
     else
          begin
          declare k,i:integer;

          k←randint(1,b.top);
          i←b.A[k];
          b.A[k]←b.A[top];     comment fill in gap;
          b.top←b.top-1;
          return i
          end

end of type bag;
```

If we desire to remove the restriction on the size of a bag, we can store the bag contents in a list. To define a list of integers, we need to define each node in the list and the rules for connecting them. We will first specify type **node**, so that we can use its general purpose operations to manage a list:

```
type node =
    declare    val:integer,
               next:ref node;

    op create(i:integer):node =
         begin
         declare n:ref node;

         n←new node;
         n↑.val←i;
         n↑.next←null;
         return n↑
         end;

    op destroy(n:node) = free n;
```

The operation called follow causes node1 to point at node2:

```
op follow(node1,node2:node) = node1.next←node2;
```

Operation successor returns the current next value of node1:

```
op successor(node1:node):node = return node1.next↑;
```

and operation value returns the current integer value of node1:

```
op value(node1:node):integer = return node1.val

end of type node;
```

A useful type of list for the purpose of defining type **bag** is a circular list, so that a random choice from a bag is accomplished by "spinning" the list before deletion. A circular list of integers defined using type **node** could be:

```
type clist =
    declare    current:ref node,
               size:integer;

    op create:clist =
         begin
         declare c:ref clist;

         c←new clist;
         c↑.current←null;
         c↑.size←0;
         return c↑
         end;
```

The insert operation maintains the proper circularity:

```
    op insert(i:integer,c:clist) =
         begin
         declare t,n:ref node;

         c.size←c.size+1;
         n←node.create(i);
         if c.size=0 then
              begin
              c.current←n;
              node.follow(c.current↑,c.current↑)
              end
         else
              begin
              t←node.successor(c.current↑);
              node.follow(c.current↑,n↑);
              node.follow(n↑,t↑)
              end
         end;
```

The circular list spin operation can be defined as:

```
op spin(c:clist) =
     begin
     declare n:integer;

     n←randint(0,c.size);
     for i from 1 to n do
          c.current←node.successor(c.current↑)
     end;
```

and the remove operation (which deletes the element <u>after</u> current) as:

```
op remove(c:clist):integer =
     begin
     declare   t:ref node,
               k:integer;

     if c.size=0 then error     comment underflow;
     else if c.size=1 then
          begin
          k←node.value(c.current↑);
          node.destroy(c.current↑);
          c.current←null;
          return k
          end
     else
          begin
          t←node.successor(c.current↑);
          node.follow(c.current↑,node.successor(t↑));
          k←node.value(t↑);
          node.destroy(t↑);
          return k
          end
     end;
```

One more useful operation will tell us if the list is empty:

```
op empty(c:clist):boolean = return (c.size=0)

end of type clist;
```

Now type bag can be simply defined as:

```
type bag =
    declare c:clist;

    comment the create operation for a clist performs the
          necessary initialization for a bag;

    op empty(b:bag):boolean) = return clist.empty(b.c);

    op put(b:bag,i:integer) = clist.insert(i,b.c);

    op take(b:bag):integer =
          begin
          clist.spin(b.c);
          return clist.remove(b.c)
          end

    end of type bag;
```

Note that we have completely re-designed the algorithmic specification of type **bag**, but left the functional specifications unchanged. Any user of type **bag** would not have had his programs affected (except for the size restriction having been removed). Note also that the intermediate types **node** and **clist** can be re-used in the specification of other list structures or more abstract types such as **bag** without the need for duplication of code or effort. In addition, the three types make up three modules suitable for independent programmer work assignments.

6) Parameterization

As we mentioned in the introduction, abstraction involves the noticing of similarities. This requires that things which may be potentially dissimilar among a group of objects have a wide range of freedom of variation. The use of parameters to procedures is necessary to convey the notion of abstract operation. The "square" operation gains its usefulness because it will square any number. When types are given parameters, they acquire much wider applicability in a similar way.

A parameterized type is called a *type constructor*, since the types resulting from different parameter values may be widely differing. Consider the definition of type **string** in terms of the primitive type **character**:

```
type string(n:integer) =
    declare    A:array[1:n] of character,
               len:integer initially n;

    op concat(s,t:string):string =
         begin
         declare r:string(s.len+t.len);

         for i from 1 to s.len do r.A[i]←s.A[i];
         for i from 1 to t.len do r.A[s.len+i]←t.A[i];
         return r
         end

    etc.

    end;
```

A type constructor may be thought of as a macro. When its actual parameters are specified, it completely defines a new type. In the declaration of r in **string.concat**, the parameter to its type is given as (s.len+t.len). A vector of this length (A) is allocated, along with a word (len) containing this value. If not for the ability to parameterize a type definition, we would have to define a new type **string** for every possible string length!

Even more important is the ability to use type names as parameters to type definitions. In our previous example, we made the unfortunate (and irrelevant) committment that the objects contained in a **bag** must be integers. If, however, we had written the definition of **bag** as:

```
type bag =
    declare c:clist(integer);

    etc.

    end;
```

we would be using a much more general notion of list. In fact, there is no reason to restrict a bag to only integers:

```
type bag(t:type) =
    declare c:clist(t);

    etc.

    end;
```

with t replacing all occurrences of **integer**. Type **clist** would be defined in the same way as before, except that the type of the variable called current would be determined from a parameter, as would the type of the variable called val in type **node**.

It would not be hard to implement the notions of **bag** and **clist** in a language with dynamic allocation and pointer facilities. In other languages it might be necessary to re-implement them for every different parameter value (e.g. using arrays), but at least the specifications do not have to be re-done.

In general, parameterization gives us the ability to precisely describe such concepts as stack, queue, tree, etc., without tying down the types of values which are not important to the structure of the data.

7) A Larger Design Example

A data type having many applications in systems programming is the hash table. For this discussion, we characterize a hash table by the type of object used to index the table, by the type of object stored in the table, and by the number of buckets provided:

```
type hashtable(buckets:integer,index,result:type) =
```

Each bucket contains a list of elements with the same hash value. Each element is an (index,result) value pair. Assuming a type pair which we can use to group two values of arbitrary type:

```
type pair(left,right:type) =
    declare    lval:left,
               rval:right;

    op leftval(p:pair):left = return p.lval;

    op rightval(p:pair):right = return p.rval;

    op create(l:left,r:right):pair(left,right) =
         begin
         declare p:ref pair;

         p←new pair;
         p↑.lval←l;
         p↑.rval←r;
         return p↑
         end

    end;
```

and assuming a definition for type list similar to LISP:

```
type list(t:type) =
    declare p:ref pair(t,list(t));

    op head(l:list(t)):t = return pair.leftval(l.p↑);
            comment LISP car;

    op tail(l:list(t)):list(t) = return pair.rightval(l.p↑);
            comment LISP cdr;

    op cons(tv:t,lv:list(t)):list(t) =
        begin
        declare q:ref list(t);

        q←list.create;
        q.p←pair.create(tv,lv);
        return q↑
        end;

    op create:list(t) =
        begin
        declare l:ref list(t);

        l←new list;
        l↑.p←null;
        return l↑
        end;

    op empty(l:list):boolean = return (l.p=null)

    end;
```

we can specify the structure of a **hashtable** as:

```
    declare A:array[1:buckets] of list(pair(index,result));
```

In order to find a result value given an index value, the appropriate array element must be found using a hash function. Then the list must be searched for the index value. Since the hash table module knows nothing about the representation of values of type index, it is up to index itself to provide that hash function. For example,

in the case of indexing by strings, we would expect to find the operation **string**.hash defined. We can then define the **hashtable**.find operation as:

```
op find(h:hashtable,i:index):result =
    return search(i,h.A[1+(index.hash(i) mod upperbound(h.A))]);
```

where search is a procedure (local to type **hashtable** only) to scan the list:

```
procedure search(i:index,l:list(pair(index,result))):result =
    begin
    declare p:list(pair(index,result));

    p←l;
    while not list.empty(p) do
        if pair.leftval(list.head(p))=i then
            return pair.rightval(list.head(p))
        else p←list.tail(p);

    return result.undefined
    end;
```

Note that if the search procedure does not find the index on the list, it returns the undefined value of type **result** - a convenient way to indicate an error to the caller. This of course assumes an operation called "undefined" for type **result**.

It now remains to be able to insert values into the hash table. The insert function must search the appropriate list and only add to the list if the value is not already there:

```
op insert(h:hashtable,i:index,r:result):boolean =
      begin
      declare n:integer;

      n←1+(index.hash(i) mod upperbound(h.A));

      if search(i,h.A[n])=result.undefined then
            begin
            A[n]←list.cons(pair.create(i,r),A[n]);
            return true
            end
      else return false
      end;
```

Finally, the creation of a new **hashtable** requires some initialization:

```
op create:hashtable =
      begin
      declare h:ref hashtable(buckets,index,result);

      h←new hashtable;
      for i from 1 to upperbound(h↑.A) do
            h↑.A[i]←null;

      return h↑
      end

end of type hashtable;
```

8) A Directory System

To illustrate the power of systems design using data types, we have chosen as our final example a hypothetical directory system as might be found in an operating system.

The directory system to be designed will be a hierarchical one, in which a node name consists of a path through the directory structure to the appropriate descriptor.

For example, the name A.B.C refers to directory A, subdirectory B, descriptor C. There should be no a priori bound on the number of directories, nor on the depth of the hierarchy. We will not restrict the definition to that of file directory, since the notion of file is not relevant to the structure. The purpose of the directory is to associate an object of type **descriptor** (maybe **filedescriptor**) with one of type **path**. The operations provided for directories should include lookup and insertion functions. If we think of a path name as a list of strings, we can define type **path** as:

```
type path =
    declare l:list(string);

    op first(p:path):string = return list.head(p.l);

    op rest(p:path):path = return list.tail(p.l);

    op empty(p:path):boolean = return list.empty(p.l);

    op terminal(p:path):boolean = return path.empty(path.rest(p))

    end;
```

The directory itself will contain objects which will either be descriptors or sub-directories. We will call this type **direntry** for directory entry. The structure of a **directory**, then, will be a hashtable (as previously described) which is indexed by a **string** and which contains **direntry** objects:

```
type directory(descriptor:type) =
    declare H:hashtable(100,string,direntry(descriptor));
```

The hashtable will have 100 buckets, but we could have left that as a parameter to **directory**.

In order to look up a **path** in a **directory**, we must first find the **direntry** corresponding to the **path** header (via **hashtable**.find) and then look up the rest of the **path** through the **direntry**:

```
op lookup(p:path,d:directory:descriptor) =
      begin
      declare t:ref direntry;

      t←hashtable.find(d.H,path.first(p));

      if t↑=direntry.undefined then
            return descriptor.undefined
      else return direntry.lookup(path.rest(p),t↑)
      end;
```

Insertion in a **directory** may require the creation of a **direntry** and its insertion in the hashtable, or simply insertion in an existing **direntry**:

```
op insert(p:path,d:directory,r:descriptor):boolean =
      begin
      declare t:ref direntry;

      t←hashtable.find(d.H,path.first(p));

      if t↑=direntry.undefined then
            begin
            t←direntry.create(p,r);
            return hashtable.insert(d.H,path.first(p),r)
            end
      else return direntry.insert(path.rest(p),t↑,r)
      end;
```

Finally, the creation of a **directory** is accomplished by:

```
op create:directory(descriptor) =
     (new directory(descriptor))↑

end of type directory;
```

Having finished the definition of type **directory**, we must now tackle the definition of type **direntry**. A direntry has a name (part of a path) and either indicates a descriptor or a sub-directory:

```
type direntry(descriptor:type) =
    declare   name:string,
              node:boolean,
              value:ref descriptor,
              subdir:ref directory(descriptor);
```

To look up a **path** in a **direntry** which is a node requires checking that the **path** terminates at that point. If the **direntry** refers to a sub-directory, then the **directory**.lookup operation is recursively applied:

```
op lookup(p:path,d:direntry):descriptor =
     if d.node then
          if path.empty(p) then return d.value↑
          else return descriptor.undefined
     else return descriptor.lookup(p,d.subdir↑);
```

To insert a **descriptor** in an existing **direntry**, we must do the insertion in the sub-directory (if there is one):

```
op insert(p:path,d:direntry,r:descriptor):boolean =
     if d.node or path.empty(p) then return false
     else return directory.insert(p,d.subdir↑,r);
```

Creation of a new **direntry** (as required by the **directory.insert** operation) is straightforward when the **path** is a terminal. When it is not, creation of the **direntry** requires creation of a sub-directory:

```
op create(p:path,r:descriptor):direntry(descriptor) =
     begin
     declare d:ref direntry(descriptor);

     d←new direntry;
     if path.terminal(p) then
          begin
          d↑.name←path.first(p);
          d↑.node←true;
          d↑.subdir←null;
          d↑.value←r
          end
     else
          begin
          d↑.name←path.first(p);
          d↑.node←false;
          d↑.value←null;
          d↑.subdir←directory(descriptor).create;
          directory.insert(path.rest(p),d↑.subdir↑,r)
          end;

     return d↑
end;
```

In the end we have a precise description of the concept of directory. This directory system may be used in conjunction with a file system (note that there is a difference between a file system and the directory structure it uses, as we have just shown). It may also be used to store objects in a protection system, or for any of a number of other applications. Each of the types **directory, direntry, hashtable,** etc. may be viewed as a module suitable for a programmer work assignment.

Summary

There is much confusion in the programming community today over such concepts as modularity, structured programming, specifications, etc. What we have presented here is a technique for employing all of these tools in unified form - i.e. data types. We have, of course, omitted discussion of many topics associated with data types, but have done so on the grounds that we are concerned with design and specification, and not coding. The freedom to use a specification language without constraints of particular syntax is very important.

The use of data types as a modularization principle views a module as being responsible for the maintenance of some invariance. The explicit advantage of this is that system verification may be done selectively by module. An implication of this is that most changes to the system will be restricted to a small number of modules, if not a single one.

Acknowledgement

I would like to thank Prof. A. N. Habermann for reading and commenting on the various drafts and, more importantly, for encouraging my interest in this area.

REFERENCES

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

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

3 Liskov, B. and Zilles, S., "Programming With Abstract Data Types," **SIGPLAN Notices** (April 1974) 50-59.

4 Parnas, D. L., "On the Criteria to be Used in Decomposing Systems into Modules," **CACM** 15,12 (Dec. 1972) 1053-1058.

5 Tennent, R. D., "PASQUAL: A Proposed Generalization of PASCAL," Department of Computing and Information Science, Queens University, Kingston, Ont. (Feb. 1975).

6 Wegbreit, B., "The Treatment of Data Types in EL1," **CACM** 17,5 (May 1974) 251-264.

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

8 Wirth, N., "The Programming Language PASCAL (Revised Report)," Berichte der Fachgruppe Computer-Wissenschaften, EidgenossischeTechnische Hochschule, Zurich (1972).

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

Contents