A Survey of Data Type Specification Methods

by

SHIOWJY FAN

B.S., Fu-Jen Catholic University(Taiwan, R.O.C.), 1979

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements of the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1982

Approved by:

Major Professor

ACKNOWLEDGEMENTS

I wish to express my sincere thanks to my advisor, Dr. David A. Schmidt, for his valuable help, guidance, and suggestions. Thanks are also given to the other members of my committee, Dr. Paul S. Fisher and Dr. Rodney M. Bates.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Most all programming languages provide constructs for structuring the control flow of algorithms, but few languages provide the capability for the users to define their own data types, which can make the program shorter and easier to understand. In particular, we are concerned with abstract data types, so named since they prevent a user from using the representation information, or something in connection with representation, in a nonintended way.

The key concept in this form of data type specification is abstraction (26). Abstraction provides a mechanism for separating those attributes of an object or event that are relevant in a given context from those that are not. Abstraction serves to reduce the amount of detail that must be comprehended at any one time. One of the most significant aids to abstraction in programming is the self-contained subroutine (27). When one decides to invoke a subroutine, it can be treated as a 'black box', the details of its representation unimportant (and hidden) from its user. It is the same idea that we wish to examine in data type

definition and use.

Since abstract data types are useful to programmers, there are many computer scientists who are trying to extend existing languages or design new languages in which programmers can define their own data types. One view (35) of the basic requirements for abstract data types in programming languages is:

1. access to an abstract data type is allowed only through the operator set for the data type.
2. language constructs in the base language should be extendable to abstract data types.
3. definitions of the abstract data types should allow for formal parameters in their definitions. Invocations through declarations of abstract data types should allow for corresponding actual parameters.
4. the definition of an abstract data type should be implemented by the compiler as a truly new data type.
5. operations on abstract data types should be implemented efficiently.

These ideas are held by many researchers in the area, and we will see these five concepts realized again and again in the examples to follow.

This report is structured in four parts. The first part is this introduction. Chapter 2 contains the five different approaches to data type definitions. Chapter 3 reviews three different methods used for construct data type definitions and gives examples to explain the construction. Type recursion, constructs for type conversion, parameterization, and exception handling are discussed in this chapter too. Some conclusions and future trends are given in chapter 4.

CHAPTER II

WHAT IS A DATA TYPE

Different people have different concepts about data types, so many definitions of data types are distinctive. With two ways, we can specify the semantics of abstract data type. One is operational specification. It begins with some well-understood language or discipline and builds a model for the type in terms of that discipline. The other is definitional, axiomatic, or algebraic specification, which includes two parts. The first part is a syntactic specification, which provides the syntactic information that many programming languages already require: names of operators and the domains and range of the operation associated with them. The second part gives a list of relations which define the meanings of the operations by stating their relationship to one another.

First, we state five different approaches to defining a data type, and then we use the approaches to organize the different definitions.

2.1 Approaches to define a type

Generally, there are five approaches to define a data type (47):

1. Syntactic: A data type is described by the information that one gives about a variable in a declaration, such as VARIABLE X IS ***. The *** represents implementation-oriented attributes, such as number and size of storage cells, methods of allocation and deallocation. Sometimes built-in names associated with specific attribute sets are used.
2. Value space: A type is defined by a set of possible semantic values. For example, the set of integers is a value space and thus a data type. Given some existing types, set-theoretic union, cartesian products, etc., can be used to form new types.
3. Behavior: A type is defined by a value space and a set of operations on elements of the space. This is an extension of 2, and is the widely accepted semantic definition of data type. For example, the value space of integers has the operations addition, subtraction, etc.
4. Representation: A type is determined by the way that it has been represented in terms of more primitive computer-oriented types. Pascal's INTEGER is an

example. Hardware (or compilers) must have implemented these primitive types. In this way, the definition of complex types can be understood in terms of computer implementation.

5. Representation plus behavior: A type is determined by a representation plus the set of operators that define its behavior; these operators are defined in terms of a set of procedures operating on the representation.

2.2 Examples of the approaches to data type definition

The following sections introduce the various research ideas which support the different approaches of data type definition.

2.2.1 Syntactic

PL/I and FORTRAN are standard programming languages which approach the data types as 'syntactic'; type is the definition that one gives about a variable in a declaration. In PL/1, we can define:

```
DCL SEMESTER DECIMAL FIXED (1) STATIC REAL;
DCL FLDA BINARY FIXED (3,2) EXTERNAL;
```

In the declaration, we (should) give all the information

that machine needs for allocation. Often default attributes are applied when some of this information is omitted.
In Fortran, we can define:

```
INTEGER*2 JACK,BILL
REAL*8 B,DSQRT
```

In this declaration, we define storage size in bytes for the variables (*8 means 8 Bytes for allocation) to control actual allocation. These two languages emphasize the use of machine based information. The types defined by them are concrete, not abstract at all.

2.2.2 Value space

Hoare (31) defines data types as a value space, that is a type is defined by a set of possible values. Some axioms of this version of type are:

1. A type determines the class of values which may be assumed by a variable or expression.
2. Every value belongs to one and only one type.
3. The type of a value denoted by any constant, variable, or expression may be deduced from its form or context, without any knowledge of its value as computed at run time.

4. Each operator expects operands of some fixed type, and delivers a result of some fixed type (usually the same).
5. The properties of the values of a type and of the primitive operations defined over them are specified by means of a set of axioms.
6. Type information is used in a high-level language both to prevent or detect meaningless constructions in a program and to determine the method of representing and manipulating data on a computer.

As mentioned earlier, based on sets of primitive value spaces we can define compound value spaces using the set-theoretic operations of union, discriminated union, powerset construction, cartesian product, and function space.
Some examples are:

Cartesian Products:

```
complex = real * real
```

Discriminated Union:

```
exceptions = Parity-faulty | empty | manual
```

Powerset:

```
type primary-color = (red,yellow,blue)
type color = powerset of primary-color
```

By first definition, each value of the type complex is a

structure with exactly two components. The first component is real and second component is also real-- in other words, an ordered pair of reals. The second definition states that type exception consists of exactly three values with the names indicated and that every exception variable has one of these three values. In the third example, we can get the different mixed colors from the primary-color.

2.2.3 Behavior

In the behavior approach, a type is defined by a value space and a set of operations on elements of that space. This format is neatly supported by the algebraic specification and algebraic theory specification. These representation forms usually consist of two parts: 'interface specification' and 'behavioral specification'. Interface specification consists of the name of the type, and the names and types of the associated operations. Behavioral specifications use the 'axiomatic specification' and 'abstract model approach'. A convenient way of syntactically describing the semantic propertities of a data type algebra is by using an axiomatization (equation specification). The idea is similiar to the use of a first order theory in mathematical logic to describe the inherent propertities of a model. (15)

Axiomatic specifications define the behavior of an

abstract data type by giving axioms describing the results of applying operations to arguments. A simple example of axiomatic description of stack operations is:

```
pop(push(a,s)) = s
empty?(push(a,s)) = false
empty?(nullstack) = true
```

However, a valid criticism of the approach is that only properties of the operations are described the operations themselves are left undefined. In the abstract model approach, the objects of the data type are represented in terms of other data abstractions with known properties established by formal (probably axiomatic) specifications given in advance. These 'other abstractions' form a programming language for data type definition. A good example of this method is using Scott's LAMBDA to define the denotational semantics of a data type (54).

A more powerful method for converting semantic properties is through theory representation (8); a signature together with a set of equations using the operators of the signature and respecting their input and output sorts is used. The signature is a set of sort names (names of semantic data sets) and a set of operator symbols, each with a given sequence of sorts for its arguments and a sequence of sorts

for its results.

In both the axiomatic and theory forms of representation, universal algebra (24) is applied to determine that the model represented by the specification is the initial algebra (23) of the class of algebras satisfying the specification.

2.2.4 Representation

When using the representation format, a type is determined by the way that it has been represented in terms of more primitive types. This approach is supported by Pascal-like languages, such as PASCAL, and ALGOL 68. For example PASCAL (25) has several notable kinds of data types:

1. the scalar data type:
   an ordered set of values, such as
   scalar = (red,yellow,blue,green)
2. the subrange type:
   a subrange of any scalar type such as
   digits = 1..9
   yg = yellow..green
3. the record type:
   a structure consisting of a fixed number of components, possibly of different types:

```
complex = record
            rpart: integer
            ipart: integer
            end;
```

2.2.5 Representation plus behavior

A type is determined by a representation plus the set of operators that define its behavior. Most newly developed programming languages support this definition. The idea originally stems from SIMULA (1), and most of the versions follow a similar syntax:

```
type <identifier> =
     <definition of structures used internally
      to hold data >
     <subroutine-like definition for operation 1>
     ..........
     .........
     <subroutine-like definition for operation n >
end type
```

Most of the definitions of this form of data type (8, 10, 26, 27, 28, 30, 40, 42, 53) use a representation-independent specification and a set of values and a set of operators: together they introduce a new typ of data object that is deemed useful in the domain of the problem being solved. At the level of use, the programmer is concerned with the behavior of these data objects, what kinds of information

can be stored in them and obtained from them. The programmer is not concerned with how the data objects are represented in storage nor with the algorithms used to store and access information in them. The programming languages, such as CLU, ALPHARD, EUCLID, MODEL, etc., support the approach of defining data type as representation plus behavior. Concurrent Pascal and Simula 67 call this structure a class, but many other names are used as well-- those will be explored in the next chapter.

Since this format is so widely used, we have a wide number of opinions about its exact purpose. Flon (17) describes that data type is an arbitrarily complex form which characterizes a certain kind of behavior. Type definition is a representation for objects of the type and the representation is known only to the definition itself. (It is an algorithmic specification, the means by which that behavior is accomplished.)

Parnas (47) describes types as classes of modes. Each mode defines a simple class of variables such that variables with the same type can be substituted for each other in any context and not cause a compile-time error. A language should allow programmers to define types as below:

1. Specification-type (spec-type): a type consisting of modes with identical externally visible behavior. For any mode defined by a representation and a set of permissible operators, one can describe the characteristics of this mode by using the operators provided. When a group of modes all have the same characteristics with respect to these operators, this group constitutes a spec-type. Modes having the same spec-type can share operations originally defined for the specific modes in the spec-type. Algol 68 has this feature:

```
mode i = integer
var a:i; b:integer;
```

   In this example, we can assign the value of b to a. But this feature is not allowed in Pascal.

2. Representation-type (rep-type): a type consisting of a collection of modes with identical machine representations. A user declares a rep-type as a group of modes that have the same representation and to define a set of operations on variables of that type in terms of the representation.
   We see in PL/1:

```
dcl nine bit (9);
dcl nine binary fixed (9,0);
```

These two variables in the example have the same rep-type.

3. Parameterized type (parm-type): a type consisting of modes that are invocations of parameterized mode description. Based on earlier defined 'mode-builder' descriptions in which key information has been abstracted as parameters, specific 'actual parameter' modes are supplied to form an invocation and produce a parm-type. This type should allow the user to declare as members of the same type, modes that can be generated by assigning values to the parameters of a mode description. For example, programmers can convert INTEGER ARRAY [m:n] to a mode-builder TYP ARRAY [m:n], where TYP is the abstract mode.
4. Variant-types: a type consisting of modes with some common properties. This type allows programmers to exploit the same properties in different modes which do not have identical specification, such as when the programmer wants to extract the same attribute from the different record structures in the data base. This abstract data types should allow the programmers to define the type includes the all needed attributes. This type is also defined by a specification, and the

operators specified to be common to all variables of the mode must be implemented for the new type in accordance with those specifications.

CHAPTER III

CONSTRUCTS FOR DEFINING DATA TYPES

In this chapter we consider the available formats for defining new data types. A large number of examples are provided. Each one has its own format for definition, and among similar methods are different mechanisms.

3.1 Existing methods for defining data types

In general, there are three methods used by computer scientists to define data types(40). One is in extensible languages, in which an existing language contains a construct for extending its features in order to define a new data type. This method is defining representations rather than abstract data types, and it is impossible to define all the operations characterizing an abstract data type. This method uses the representation approach to type definition.

The second approach is Simula-like. Inside the class, there are subroutines to define the operations of this new data type. Every attribute and function in a class is

accessible in the block in which the class definition is embedded. Therefore the actual form of the representation is always known to the user but can not be changed from outside.

The final approach uses standard abstract operations which define a set of abstract operators to create, access, modify and destroy abstract data collections. A data collection is a map from a set of selectors to a set of values, and that operations on data collections are either transformations on the map or use the map to access elements. OBJ and CLEAR are languages which use this method to express and execute algebraic specifications of programs.

Each language (58) has a different emphasis in its constructs for defining data types. Details of the basic notion of type, objects vs. variable model, identification with encapsulation instance, separation of specification and implementation, the role of generics and parameterization, closed vs. open scopes, generic instantiation, concurrency, mapping to/from the underlying representation of a type, etc., form a number of variations. The following sections discuss three methods for defining data types and their supporting languages or theories. The later part of this chapter will discuss type recursion, constructs for type conversion, parameterization and exception handling.

3.2 Representation approaches to data type definition

A Pascal-like language is a standard example using data structures to define a new data type. A language like Pascal normally has a small number of built-in primitive types. These can be scalar types, such as INTEGER and BOOLEAN, or nonscalar, such as REAL. The methods (31) for constructing new data types are:

1. arrays: A mapping from any finite scalar type to any type at all. An example is declare A: array[n..m] of integer; in Pascal the bounds are both fixed and part of the type, but scalar types are allowed as subscripts as well as integers. For example, var A: array [color] of integer is legal. If we have:

```
suit = (clubs,diamonds,hearts,spades);
card = record s:suit; value=1..13 end;
var c:card
```

We can use these two notations (25):

(1) for<variable> in ordered <ordered set of values> do s

(2) for <variable> in unordered <set of values> do s

to manipulate the iteration of FOR loops. From the

above declarations, we can write:

```
for c.value in ordered 1..13 do
    for c.s in ordered suit do
    ....
    ....
for c in unordered card do s
  ( does not care about the order in which
    values from card are used)
```

2. records: A record is a structure consisting of a fixed number of components, called fields. Records are used to group values of potentially differing types:

```
declare r : record (f1:t1,f2:t2,....fn:tn);
```

Where f1,f2,..fn are variables and t1,t2,...tn are types. The figure 1 contains the example of record in PASCAL and ALGOL 68 to define the variant fields of male and female.

PASCAL:

```
type person = record
          name: string;
          age: integer;
          case sex: (male,female) of
          male: (height,weight: integer);
          female: (size: array [1..3]of integer);
          end
        end;
```

ALGOL 68:

```
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);
```

Figure 1. Two examples for records

3. Enumeration types: A new data type is formed by exhibiting the set of constants which comprise its value. For example:

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

In order to define a new enumeration type (16) consistent with the rest of the language, we must specify the set of constants of that type and the associated primitive operators. In PASCAL, we can use pred, succ, etc. in enumeration type. Using above

example, pred(blue) is yellow, succ(blue) is green, and pred(blue) = succ(red).

4. Pointers: A reference value is a 'pointer' to an object, with the restriction (in Pascal) that reference variables can only reference objects of a single type:

```
declare c: ref integer;
        c ← new integer; free c;
```

New allocates a new cell for integer, and free deallocates the cell where c points to.

5. Procedures: When treating procedure as a data type, one can do some arithmetic operations in the body and return a value from the function (as in PASCAL). Programmers can often overload the new data type with (25):

(1) operator overloading: Assume data type complex is a record containing two reals:

```
function + (x,y:complex):complex;
  begin result.rpart:= x.rpart + y.rpart;
        result.ipart:= x.ipart + y.ipart;
  end
```

Now + denotes both real and complex arithmetic operations.

(2) procedure and function overloading: The returned type of a function can be one of a set of parameter types.

```
function sum (a:array of T,
              n: integer):T;
  var i: integer; s: T;
  begin s:= 0;
  for i := 1 to n do s:= s+a[i];
   sum := s
  end
```

Assuming T to be a data type which can change from call to call, we can use function sum to total arrays of type complex or real.

(3) Defining subtypes: Subroutines can be used to output subtypes of an existing type:

```
function subtype(x:real):complex;
        subtype := complex(x,0);
```

We get the type of complex from real.

6. Files: These are understood in the usual COBOL or

Pascal sense.

7. Sets: Sets are unordered compound objects, similiar in spirit to their mathematical counterparts. Of course, sets can contain members from ordered types, but the types are usually restricted to be scalars. For example:

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

8. Subrange: Given a primitive type, it is possible to define a type to be a subrange of the other type. For example:

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

Subranges of nonscalar and compound types are rarely found.

Gries(25) describes two notions that can be used in Pascal-like languages. The first one is array-like. An array object is declared with an index set (subscript values), and the set of legal subscript values is a data type. One should be able to declare variables of that type.

For example:

```
var a: array [1..n] of integer;
var i: domain(a); s:integer;
    s:= 0;
    for i in unordered domain(a) do s:= s+ a[i]
```

handles the summing of array a.

If we define:

```
var suit = (club,diamonds,hearts,spades);
var c = array [1..10,1..10,suit] of ...;
var k : domain(c);
var l :1..10;
var m : record comp1:1..10;comp2:1..10 end;
var n : suit;
```

then c[k],c[l,l,spades],c[m,n]are all legal expressions.

The second notion generalizes procedures; the type of a formal parameter of a procedure can depend on a particular call of that procedure. For examples

```
(1) procedure x (var a:<z>);...
(2) procedure x (var a:array² of <z>);....
(3) procedure x (var a: array^<n> of integer);....
(4) procedure x (var m: record c1:integer;c2:<z> end);
          var n: <z>;....
```

<> indicates the type will depend on the actual parameters of call. Z stands for a type and n is an integer. With this feature, we can define one procedure to handle variously-typed arguments. For example:

```
procedure swap (var a,b :<t>);
        var x: t
    begin x:= a;a:= b; b:= x end
```

The example above is able to exchange values of two variables, regardless what the type of the formal parameters are.

3.3 Representation plus behavior forms of data type definition

Simula (1) and Concurrent Pascal (7) are the standard examples of representation plus behavior method to define new data types. They use the class construct to define new data types. A class contains an internal data structure plus subroutines defining operations on the structure. Programmers can only access a class through the declared procedures, and have no other names of accessing the class internal structure.

Concurrent Pascal (7) is an extension of Pascal, having

features to define new data types and to perform Parallel processing. The classes are used to define new data types. The structure of a class is somewhat similar to a small Pascal program. Operations can only be accessed through the procedure entries and the local variables of the class can not be accessed by outside. Figure 2 contains the example of Concurrent Pascal to define FIFO as a new data type.

```
type fifo = class (n:integer);
   var inp, oup,ct: integer;

   function entry empty:boolean;
      begin empty:= ct=0 end;

   function entry full:boolean;
      begin full:= ct=n end;

   function entry arrival:integer;
      begin arrival:= inp;
            inp:= (inp +1) mod n;
            ct:= succ(ct);
      end;

   function entry departure:integer;
      begin departure:= out;
            out:= (out +1) mod n;
            ct:= pred(ct)
      end;

   function entry count:integer;
      begin count:= ct end;

   procedure initial;
      begin inp := 0;
            out := 0;
            ct := 0;
      end

   begin initial end;
```

Figure 2. Example of Concurrent Pascal to define a data type

The operations of FIFO are function entries, such as empty, full, and departure. These operations defined by function (procedure) entries are accessible by external processes. Local procedures without entry, such as procedure initial, can not be accessed by the outside environment.

Simula (1) is an other language which uses data structures and subroutines to define data types. Simula is based on Algol 60 ; the basic concepts are extended in Simula by:

1. concatenation: propertities of two or more classes may be fused together, minimizing textual description. In general, if C1,C2,...Cn are classes such that C1 has no prefix and Ck has the prefix Ck-1 (k= 2,3,...n) then the suffix k is said to be the prefix level of Ck. Cp is a sub-class of Cq if p>q. An object of a class has a main part (the class declared at the highest prefix level) and a prefix part (the remainder of the chain). A statement at prefix level k has direct access to all the attributes declared at the prefix levels equal to or less than k, except those hidden by conflicting definitions. A statement at prefix level k has access to attributes at higher level only through the virtual mechanism.
2. extended binding rule: the binding rules of Algol are extended enabling a semantic redefinition of quantities valid at all levels in concatenated objects by the virtual concept. Connection statements allow the temporary shifting of an environment.
3. block prefixing: classes may be used to prefix program blocks, thus providing an environment in which to

operate. The occurrences of a class name in a prefix makes all the attributes of that class accessible within the prefixed class. When an attribute wants to reference a variable, which is at the higher prefix level. We can use qua mechanism to get the qualification of a reference variable and have the compile time checks it is valid or not.

The example of Simula's class is illustrated in figure 3.

```
class matrix;
   begin class rectangular(a,m,n);....;
         class column(a,m);.....;
         class row(a,m);real array a;integer m;

         virtual:real procedure norm;
            begin real t; integer i
               for i=1 step 1 until m do
                  t := t+a[i]↑2;
                  norm := sqrt(t)
            end of norm;

         procedure normalise;
            begin real t; integer i;
                  t := norm;
                  if t ≠ 0 then
                  begin t := 1.0/t;
                     for i := 1 step 1 until m do
                        a[i] = t*a[i]
                  end

            end of normalise;
            ............
         end of row;
         ...........
   end of matrix
```

Figure 3. Example of Simula to define a data type

In this example, class rectangular, column, row are defined local to class matrix. The procedure norm is virtual, so it can be redefined by the sub-class of class row and can have external compilation. This sub-class can have all the variables and procedures of class row plus those procedures and variables declared in this sub-class. We can redefine the body of procedure norm:

```
begin external class matrix;
      matrix begin row class row1;
         begin real procedure norm;
            begin integer i; real t;
               for i := 1 step 1 until m do
                t := t + abs(a[i]);
                norm := t
            end of norm;
            ........
         end of prefixed block
      end
```

Row1 is a sub-class of row. After this redefinition of procedure norm, row1 will get a different value of norm from the other rows. This provides a feature for the user to redefine certain segments. Through the class structure, programmers can have more protection in the data processing.

Mesa (21) expands on the class concept; it uses modules to provide a capability for partitioning a large system into manageable units. The modules can be used to encapsulate abstractions and to provide a degree of protection. The language includes definition module and program module

constructs. Definition modules define the interface to an abstraction, and program modules, called implementers, provide the concrete implementation of an abstraction. Figure 4 gives an example of how the two consructs specify and implement a data type.

```
Abstraction DEFINITIONS =
 BEGIN....
 it:TYPE = ...; rt:TYPE = ...;...
 P: PROCEDURE ;
 P1: PROCEDURE INTEGER
 .....
 Pi = PROCEDURE [ it ]RETURNS[rt];
 .....
 END

Implementer: PROGRAM IMPLEMENTING Abstraction =
 BEGIN
 OPEN Abstraction;
 X :INTEGER;...
 P: PUBLIC PROCEDURE = <code for p>;
 P1: PUBLIC PROCEDURE = <code for p1>;
 ......
 Pi: PUBLIC PROCEDURE[x:it] RETURNS[y:rt]=
    <code for pi>;
 ..........
 END
```

Figure 4. Mesa's definition and program modules

There are two kinds of special variables in Mesa. One is a private variable, visible only in the module in which it is declared and in any module claiming to implement that module. The other is a public variable, which is visible in

any module that includes and opens the module in which it is declared. These specifications of attributes can be used to control intermodular access to identifiers. In figure 4, Abstraction contains definitions of shared types and enumerates the elements of a procedure interface. Implementer uses those type definitions and provides the bodies of the procedures; the compiler will check that an actual procedure with the same name and type is supplied for each public procedure declared in Abstraction. The Mesa modules are also used in ADA (32).

Alphard uses a 'form' to define a new data type. The generator construct is a special form which performs a sequence of bindings to the control variable of a loop in Alphard (52). It gives a convenient mechanism for constructing for loops operating on new types. The basic form to define a new data type is:

```
form istack (n:integer)=
beginform
specifications......;
representations......;
implementations......;
endform
```

Inside a form definition, programmers can provide

verification information, such as pre and post logical assertions. The example of form definition is illustrated in figure 5. This example defines the data type 'upto' as an interval [lb..ub] for use in looping.

```
form upto(lb,ub:integer) extends k:integer=

 beginform

 specification
     requires true;
     inherits <allbut←>;
     let upto = [lb..ub] where lb≤ub ⊂ upto =
         [lb..k-1][k][k+1..ub];
     invariant true;
     initially true;
     function
        &init (u:upto) returns b:boolean
           post (b≡lb≤ub) ∧ (b⊃lb=k≤ub);
        &next (u:upto) returns b:boolean
           pre lb≤k≤ub
           post (b≡k'≤ub) ∧ (b⊃k=k'+1 ∧ lb≤k≤ub);

 representation
     rep(k) = if lb≤ub then [lb..k+1]k[k+1..ub]
                       else □;
     invariant true;

 implementation
     body &init out(b≡lb≤ub)∧(b⊃lb=k≤ub)=
          (u.k←u.lb;b←u.lb≤u.ub);
     body &next in(lb≤k≤ub)out (b≡k'≤ub)
               ∧ (b⊃k=k'+1 ∧ lb≤k≤ub) =
              (u.k←u.k+1; b←u.k≤u.ub);
endform
```

Figure 5. Example of Alphard to define a data type

(k' denotes the value of k upon entry to &next)

The phrase <all but←> means that all integer functions except ← are applicable to the upto. This abstract specifications describe an 'upto' as an interval [lb..ub]. Function &init gives the initial value and function &next gives the next value of the current counter. Pre and post assertions guarantee the conditions before and after the statements are true. The rep function in representation shows how an interval is represented by its two endpoints and the loop variables. Body &init and body &next perform the actual implementation. In and out must match pre and post assertions, respectively.

Another language widely studied for its structuring ideas is CLU (42). The basic elements in CLU are objects, which have a particular type, and variables, the name used in a program to refer to objects. The two types of objects are:

1. mutable objects: may exhibit time-varying behavior, may be modified by certain operators without changing the identity of the object. A record is an example.
2. immutable objects: do not exhibit time-varying behavior, such as booleans, integers, characters, and strings.

A cluster (36) is used to define a new data type in CLU; it has:

1. object representation of form:

   rep {(<rep-parameter>)} = <type-definition>

   where rep is accessable only with the cluster. The braces {} make it possible to delay specifying some aspects of the <type definition> until an instance of the rep is created.
2. object creation: use 'create' to get the initial state of cluster.
3. operation: are always specified as part of a cluster. Operations always have at least one parameter.... of type rep.

An example which defines stack as an abstract data type is illustrated in figure 6.

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

     rep (type-parm:type) = (tp:integer;
                             e-type:type;
                             stk:array [1..] of type-parm);

     create
          s:rep(element-type);
          s.tp := 0;
          s.e-type:= element-type;
          return s ;
          end

     push: operation(s:rep,v:s.e-type);
          s.tp := s.tp+1;
          s.stk[s.tp] := v;
          return;
          end

     pop: operation (s:rep) returns s.e-type;
          if s.tp=0 then error;
          s.tp := s.tp-1;
          return s.stk[s.tp+1];
          end

     top: operation (s:rep) returns s.e-type;
          if s.tp=0 then error;
          return s.stk[s.tp];
          end

     erasetop: operation (s:rep);
               if s.tp=0 then error;
               s.tp := s.tp-1 ;
               return;
               end

     empty: operation (s:rep) returns boolean;
          return s.tp=0 ;
          end

end stack
```

Figure 6. Example of Clu to define a data type

Stack is defined by the operations of push, pop, top, etc. Create sets the initial state of stack. Rep specifies the elements of stack. Each operation is specified to operate the items defined by rep. In this example, the value which the stack performs is depended on the type of the actual parameters. The maximum size of this stack has not been defined.

New data types special to EUCLID (3) are defined as 'modules'. A feature special to Euclid is the exports class, which provides communication between module and outside environment. A programmer may use the reserved word parameter to indicate elements which already have been declared. Figure 7 contains the example of Euclid's module to define new data types.

```
type stack = module

    exports (stk, pop, push)

    type stk (stacksize:unsignedint) = record
           var stackptr:0..stacksize := 0
           var body:array(1..stacksize)of signedint
         end stk

    procedure push (var istk:stk(parameter),
                    var x:signedint) =
       begin
          procedure overflow = ...end overflow
          if istk.stackptr = istkstacksize then
             overflow
          else
             istk.stackptr := istk.stackptr+1
             istk.body (istk.stackptr) := x
          end if
       end push

    procedure pop (var istk:stk(parameter),
                   var x:signedint) =
       begin
          procedure underflow = ....end underflow
          if istk.stackptr = 0 then
             underflow
          else
             x:= istk.body(istk.stackptr)
             istk.stackptr := istk.stackptr-1
          end if
       end pop

end stack
```

Figure 7. Example of Euclid to define a data type

This stack example defines the operations of pop and push. Stk is used to define the elements that this stack has. After stk's definition, we can use reserved word parameter to indicate the elements of stk. Stackptr is

initiated to be zero in the beginning. This stack can only store values of signed integer. Operation push has internal procedure overflow, and operation pop has internal procedure underflow to handle the error conditions.

A 'space' is the basic form in Model (35) to define data types. In Model, access to the representation of a new data type is restricted to the operations and procedures declared in the form. This can guarantee that the operations defined for the space are not subverted. The example is given in figure 8.

```
space intset {r} def concrset;

   type concrset is record {size:sizerng;
                            iset:array r,arraybnd }
   type sizerng is [0...ubnd r - lbnd r +1];
   type arraybnd is [1...ubnd r - lbnd r +1];
   type srchbnd is [1...ubnd r - lbnd r +2];

   insert is <<
      formal inset {r} a (varies); r i noresult;
      $ add element i to set a
      if not i in a then
         *a.size := *a.size +1;
         *a.iset[*a.size] := i ;
      fi;
   >>;

   has is <<
      formal r i ; inset {r} a (varies) result boolean;
      $ set membership predicate
      srchbnd j;
      boolean res;
      res := false; j := 1 ;
      repeat while j <= *a.size and not res do
         if *a.size[j] = i then res := true fi;
         j := j+1;
      od;
      return res
   >>;

   in def <<
      formal r i ; intset {r} a result boolean;
      return has (i,a)
   >>;

   opunion is <<
      formal intset {r} a (copied), b (copied)
             result intset {r};
      $union
      r k;
      intset {r} res;
      if empty(a) then res := b
      else
         k := select(a);
         res := opunion(a-k,b+k);
      fi
      return res
   >>;
```

Figure 8. Example of Model to define a data type

This example is part of definition for showing the structure. 'intset {r}' represents an abstract data type 'set of integers'. The identifier r represents a formal parameter which will take on a data type 'value' where intset is used in a declaration. For a declaration such as

intset{ 1...100 } a ;

the formal parameter r takes on the actual parameter value '1...100'. 'Concrset' is the shorthand notation for the concrete representation of an 'intset'. This representation involves a 'record' of two items, a 'size' representing the number of elements in the set and an 'iset' representing the actual elements of the set as an array of integers. The notation ⌜ p...q ⌟ indicates the lower-bound is p and the upper-bound is q. The operator '#' appearing in the expression #a.size is termed a 'concretion' operator. The operation followed by 'is', is defined as local to the space definition. The operation followed by 'def', is accessible outside the space as an operation. Varies states that the operation will change the content of the formal parameter, but copies will not. The operations defined inside the space can reference to each other and can be recursive too.

Scheme (44), a notation and semantics for parameterized implementation mechanisms, is built upon 5 basic notions:

1. extendable data type
2. the ability to refer to the attributes of an object in

an uniform way

3. the ability to override the system-defined meaning of attributes
4. the ability to encapsulate a set of definitions, and
5. the ability to control external access to the attributes of an encapsulate set.

An example using Scheme is shown in Figure 9.

```
scheme queue (type itemtype) = record

   type head = record var front, rear = node.ref end;
   type node = item :itemtype extended by private var
        next = node.ref end;

   procedure init (shared q:head);
   {This will be invoked whenever a new head
    is created}
     begin q.front := new node;
           q.rear := q.front
     end;

   function empty (q:head) returns boolean;
   {Test for the empty queue}
     begin return (q.front = q.rear) end;

   procedure add (shared q.head;item:itemtype);
   {Add item to q, modifying q as a side effect}
     begin q.rear↑.item := item;
           q.rear↑.next := new node;
           q.rear := q.rear↑.next
     end;

   function remove (shared q:head) returns item:itemtype;
   {Remove (and return) the first item from q,
    modifying q as a side effect}
     begin
          if empty(q) then error;
          item := q.rear↑.item;
          q.front := q.front↑.next
     end;

end queue
```

Figure 9. Example of scheme to define a data type

The operations of queue are init, empty, add and remove. The value stored in queue is depended on the actual parameter of call. New is used to create a new node. Functions can return results, but not with procedures. This

queue is implemented by link list of nodes which hold queue elements. If values of the parameters are changed by assignments, then they must be specified by shared. Scheme has not been implemented.

3.4 Behavior approach to data type definition

Algebraic specification (36), is a standard example of using the behavior approach to define new data types. It consists of a set of object declarations and a set of axioms describing the behavior of these objects. Figure 10 is an example of an algebraic specification to define stack.

```
type stack elementtype:type

   syntax
      newstack→stack,
      push(stack,elementtype)→ stack,
      pop(stack)→stack,
      top(stack)→elementtype ∪ {UNDEFINED},
      isnew(stack)→ boolean,
      replace(stack,elementtype)→ stack,

   semantics
      declare stk:stack,elm:elementtype;
      pop(newstack) = newstack,
      pop(push(stk,elm)) = stk,
      top(newstack) = UNDEFINED,
      top(push(stk,elm)) = elm,
      isnew(newstack) = true,
      isnew(push(stk,elm)) = false,
      replace(stk,elm) = push(pop(stk),elm),
```

Figure 10. Example of algebraic specification to define a data type

This stack has newstack, push, pop, isnew, etc., as its operations. The type of the formal parameter is not fixed. Syntax defines the syntactic structure of stack and semantics defines the meaning of each operation.

One implementation language using algebric specification is OBJ. OBJ (22) is a language for writing and executing abstract formal specification of programs. It is based on an algebraic model of computation. OBJ can also be seen as a rather inefficient but very high level programming language,

in which the programmer can define and then use abstractions. There are two major syntactic units in OBJ. One is declaration, which enters definition into a data base. The other is execution, which requires expressions, using information from that data base.

The expression in OBJ has a very flexible syntax allowing not only the usual prefix, postfix and infix notations, but also what is called distributed fix notation, which permits an operator to have any desired distribution of key-words and arguments.

if B then I else J

can be an expression. The equations of OBJ provide an abstract semantics, through the initial algebra approach. It uses rewrite rules, which cause a substitution instance of left-hand sides by a corresponding substitution instance of right-hand sides. Given axiom

if T then I else J = I

and expression

2* (if T then (1+1) else 3)

the latter can be simplified to 2*(1+1) (T stands for true). Figure 11 shows an example in OBJ to define a 'data type' of

Greatest Common Divisor.

```
object gcd

   sorts / int

   ok-ops
      gcd = int, int→ int

   error-ops
      neg-arg: int

   vars i,j:int

   ok-eqns
      (gcd(i,i) = i)
      (gcd(i,0) = i)
      (gcd(0,i) = i)
      (gcd(i,j) = gcd(i-j,j) if i>j)
      (gcd(i,j) = gcd(i,j-i) if i<j)

   error-eqns
      (gcd(i,j) = neg-arg if (i<0 + j<0))

tcejbo
```

Figure 11. Example of OBJ to define a data type

'+' is boolean disjunction (or)

Sorts define the type, in this example, is integer, to be manipulated. Ok-ops and ok-eqns define the normal operations and equations, respectively. error-ops and error-eqns define the error operations and error equations.

Another example of behavior approach to data type definition is the use of algebraic theories (8), which is

implemented by CLEAR. To build an algebraic theory we need:

1. the ability to write explicit theories; the basic structure is

```
theory sorts.....
       opns.........
       eqns.........endth
```

Sorts specify the data sets used in the theory. Opns are the representation of operations and eqns list the axioms describing properties of the operations. For example:

```
The theory Nat0
   theory sort nat
          opns 0:→nat
               succ: nat→nat
          eqns                    endth

The theory Boolo
   theory sort bool

          opns true:→bool
               false:→bool
               ¬: bool→bool
               ∧: bool,bool→ bool

          eqns  ¬true = false
                ¬false = true
                false ∧ p = false
                true ∧ p = p endth
```

2. operations on theories to combine, enrich, induce, and derive, which enable us to build up theory expressions denoting complex theories. Combine is similar to set

union; it combines two theories to be one. One can combine boolean theory and natural number 0 theory to form a bool+nat0 theory:

```
The theory Boolo+Nat0
   sorts bool,nat

   opns true:→bool
        false:→bool
        ¬: bool→bool
        ∧: bool,bool→bool
        0:→nat
        succ: nat→nat

   eqns ¬true = false
        ¬false = true
        false ∧ p = false
        true ∧ p = p endth
```

Enrich is used to introduce more axioms to a theory:

```
The theory Nat1
enrich Boolo+Nat0 by

   opns ≤: nat,nat→bool
        eq: nat,nat→bool

   eqns 0 ≤ n = true
        succ(m) ≤ 0 = false
        succ(m) ≤ succ(n) = m ≤ n
        eq(m,n) = m ≤ n ∧ n ≤ m enden
```

Induce performs the transitive closure on the axiom set of a theory. This is useful for developing inductive properties of objects in a theory. For example, Induce Nat1 adds these eqns to Nat1, allowing us to explicitly reason about general propertities of members of Nat1:

eq(n,n) = true
eq(m,n) = eq(n,m)
eq(l,m) ∧ eq(m,n) ∧ ¬eq(l,n) =false

Derive can be used to create a subtheory from some existing theory. A subset of the sorts, operators, and axioms are extracted and used to build the new theory. For example

The theory Natequal
  sorts element,bool
  opns equal, true, false

from Nat by
  element is nat
  bool is bool
  equal is eq
  true is true
  false is false endde

3. procedures for theory-building:

   (1) theory constants enable us to give a name to a theory. The words with double underlining are constants.

   (2) theory procedures can take other theories as their parameters and producing a theory as a result. Their bodies use the primitive operations already defined and may call other theory procedures. For example:

```
constant pi =22/7
procedure f(x:number,b:boolean)=
        if b then pi*x else 0
procedure g(y:number)=
            let z = f(y*y,true) in z*z*z
```

We call g(2), then we can get g(2) = ((22/7)*(2*2))^3

(3) local theory definitions, permitted in the bodies of theory procedures:

```
let i = ..... in.....
```

The example of a theory specification is shown in Figure 12.

```
const triv - theory sorts element endth
proc stack (value:triv) =
induce enrich value+bool by
sorts stack

opns nilstack:→stack
     push: value,stack→stack
     empty: stack→bool
     pop: stack→stack
     top: stack→value

erroropns underflow:→stack
          undef:→value

var v:value,s:stack

eqns empty(nilstack)=true
     empty(push(v,s))=false
     pop(push(v,s))=s
     top(push(v,s))=v

erroreqns pop(empty)=underflow
          top(empty)=undef
          pop(underflow)=underflow
enden
```

Figure 12. Example of algebraic theory definition

The value handled by stack is sort element defined by triv with a sorts statement. Stack is induced and enriched from constants of value and boolean. Opns and eqns define the operations and equations, respectivelly. erroropns and erroreqns define the error operations and error equations.

3.5 Type recursion

Hoare (26, 27, 28) describes a recursive data type definition as the occurrence of a type name inside its own definition, denoting an occurrence of a (smaller) instance of a value of that type as a component. A good example of recursively typed objects is found in Lisp

```
type list = atom | list*list
```

Where a list is either an atom (defined elsewhere) or an ordered pair, whose first and second components are themselves lists. This recursive data type definition may be used to present another data type. For example, Pascal can be extended to allow recursive type definitions (29)

```
type tree-node = record
          data: t;
          left-subtree: tree-node;
          right-subtree: tree-node
          end
```

in place of the usual pointer-oriented declarations:

```
type tree-pointer =  tree-node;
     tree-node = record
          data: t;
          left-subtree: tree-pointer;
          right-subtree: tree-pointer
          end
```

The latter is a special case which standard Pascal allows, but it contains recursion, too. It would be much more convenient to allow general recursion. Then we can abbreviate 'infinite enumerations' like

```
type color = set of (red, orange,yellow, green,
                     blue, indigo, purple,....)
```

to

```
type color = red | yellow | blue | color*color
```

We can get the different colors without listing all the colors in a set. But unfortunately, there is no well-known programming language permits this simple use of recursion in data type definition.

3.6 Constructs for type conversion

Flon (16) describes the term coercion as 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. Generally the coercions are widening, as in changing integers to reals, and narrowing, as truncating reals into integers. Algol 68 has a large number of built-in type conversions, such as 'voiding':

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

In this example, x gets the integer value of i, but i's value and type will not change. Most commonly used programming languages have a set of conversion rules for the arithmetic and assignment operations. The rules are built-in, and conversion occurs automatically.

In a lower-level view of coercion, Venema (57) states that type converters allow breaches of the type system so that a value of one type can be used as a value of some

other type. To convert an object of one type to another, both objects must have the same implementation word size and be represented in the same internal format (e.g. twos-complement, binary float).

Gries (25) believes that type conversions should allow only explicit conversion from a specific type to another type. This achieves the necessary flexiblity without endangering transparency and understanding. Thus we should provide procedures to do type conversion explicitly.

```
type complex = record
               re:integer
               im:integer
               end;

function converfrominteger (x:integer)
         : complex;
   begin converfrominteger.re := x;
         converfrominteger.im := o
   end;
```

Still another viewpoint is that the language's compiler should handle all type conversions, regardless of the effort. Figure 13 is an example of a type conversion which illustrates what a compiler should do in an implementation if it allows implicit type conversion.

```
 .......
var a,b: integer; c:real;
procedure p (var x:<type>);
   var y:type; begin...end;
 .......
p(a);p(c);p(b);
 .....
```

compiler should translate above code to be

```
var a,b: integer; c:real;
procedure p'(var x:integer);
   var y: integer; begin...end;
procedure p (var x:real);
   var y: real; begin...end;
 ......
p'(a);p(c);p'(b);
 .....
```

Figure 13. Example of type conversion

3.7 Parameterization (generic procedures)

In most all languages, programmers have to declare a formal parameter with a fixed type in a procedure. It will be very convenient if this restriction could be relaxed. Parameterized types (48), which provide this feature, can eliminate the need for writing repetitive code for similar functions for different data types. Clusters in CLU, formsin ALPHARD, Macro in BLISS have some parameterization in their language designs:

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

In this case, a programmer can use any data type as the parameter.

The following example illustrates the feature of generic procedures. <> indicates the types are dependent on the actual parameters of the call, and n is an integer.

```
function subtype (x:<type>): array^<n> of <type>;
   var i:domain(subtype);
   begin for i in domain(subtype) do
            subtype(i):= x
   end;
```

The point we want to make in this example is the dimension of return type is varied too, which provides a more flexible feature in parameterization.

3.8 Exception handling

When people design a programming language, they often consider the normal conditions and forget the error (or exceptions) conditions. It is assumed that a program should only handle the normal, expected conditions, but error conditions arise nonetheless. The properties of a language (5) should have the facility to handle exceptions.

ADA has a feature to aid in exception handling. In ADA (32), we may define

```
function fromint(i:integer)
         return intorunderflow is
   begin return(isint⇒true, int⇒i)
   end;

function fromunderflow(u:underflow)
         return intorunderflow is
   begin return(isint⇒false)
   end;
```

to return an integer and boolean pair to handle exceptions. The function fromint returns an integer-boolean pair indicating that the argument is type correct and its value is i. The second, however, sets the boolean to false to indicate a run-time error.

Gaudel (20) discusses two ways of handling the errors in algebraic specification. One is restriction specification, defined by Guttag. This specification is illustrated in figure 14, which defines a queue of maximun size 100.

```
type queue,bool,int,item;

operations
    ()→queue:emptyq;
    (queue,item)→queue:append;
    (queue)→queue:remove;
    (queue)→item: first;
    (queue)→bool: empty?;
    (queue)→int: length;

axioms
    declare q∈queue, i∈item;
    remove(append(q,i)) = if empty?(q) then emptyq
                  else append(remove(q,i));
    first(append(q,i)) = if empty?(q) then i
                 else first(q);
    empty?(emptyq) = true;
    empty?(append(q,i)) = false;
    length(emptyq) = 0;
    length(append(q,i)) = length(q)+1;

restrictions
    pre(remove,q) = ¬ empty?(q);
    pre(first,q) = ¬ empty?(q);
    length(q) ≥ 100 = failure(append(q,i));
end
```

Figure 14. Example of restriction specification

The other one is error algebras, which is defined by Goguen. In this specification, error-operations and error-axioms are specified. Figure 15 shows the same example of restriction specification with error algebras.

```
type queue,bool,int,item;

operations
    ()→queue: emptyq;
    (queue,item)→queue: append;
    (queue)→item: first;
    (queue)→boolean: emptyq?;
    (queue)→int: length;

error-operations
    ()→queue: underflow,overflow;
    ()→item: missing;

axioms
    declare q∈queue, i∈item;
      the same ones as in figure 14

error-axioms
    remove(emptyq) = underflow;
    remove(underflow)= underflow;
    append(overflow,i)= overflow;
    first(emptyq)= missing;

others
    append(q,i)= if length(q) ≥ 100
            then overflow
            else append(q,i);
end.
```

Figure 15. Example of error algebras

Restriction specification sets the restriction rules to prevent where errors which might happen. But error algebras set the error-axioms which provide the error messages when an error occurs. These two methods are different, but the result is the same.

CHAPTER IV

CONCLUSION

The basic requirements for a programming language are simplicity, ease of understanding, and ease of implementation. These are good requirements for the constructs for building new data types, too. It should be easy for programmers to learn how to build data types and to understand the constructs which already have been built. The programmers should be able to ignore details and work with simplified rules to design their own data types. Some languages or concepts are too complicated for constructing a data type. Space in Model and form in Alphard are difficult for programmers to understand. Other programming languages such as Concurrent Pascal, Clear, Euclid are easier to understand and to use. Controlling overall complexity can aid in designing and implementing. Programmers should not have to spend a lot of time in figuring out how data types can be built, and they should be able to easily learn how these type constructs work. Then they can use them correctly.

Protection of the operations defined in the abstract data

type is another important consideration. Those operations defined for the data type must not be changed by outside environment, or the integrity of the type is lost. Some programming languages have explicit protection, but some do not. Module in Euclid has 'exports' and cluster in CLU have 'is' to specify the operatios which can be accessed by outside. In Concurrent Pascal, class has procedure (or function) entries which can be accessed but can not be changed by outside of the class, and have internal procedure (or function) which can not be accessed by outside of the class at all. This is a good mechanism to protect the operations of the abstract data types.

Most abstract data types can handle different sizes (passed from the parameters of each call lower and upper bounds can be varied). Some languages, such as Concurrent Pascal and Euclid have them fixed. Some languages, such as CLEAR, OBJ and CLU have them parameterized, so that the types of formal parameters of a procedure can depend on a particula call of the procedure. In the latter case, the programmer may define one abstract data type which can be used by all the similar calls with different variable types. The program size will be much smaller and the program will easier to understand. After setting up these generic procedures, the compiler must be able to check the type to see whether variables match with the operations which

accompany the type definition.

Type conversion is a very convenient feature for programmers, if it is implicitly defined in the languages. But some restrictions for type conversion must exist. Clearly we should not convert types boolean to integer, especially when there are arithmetic operations on these integers.

For a programmer, to choose a language which is simple and easy to be understood is the first consideration. But how to solve a number of technical problems affecting the efficiency of generics and tasking, and how to develop the technology of program specification and verification into a usable abstract data type model, are the next problems for the language designers. In the future, there will be more programming languages providing abstract data type constructs, and more programmers will get used to this feature, too. So to develop a library of representations and assistance with selection of which ones to use, and to mix representations, and to have automatic conversion of representations in a running system are the future trends for abstract data type developers.

REFERENCES

1. Babcicky, K., and Birtwistle, G. M. Class Distinction in Simula - Some Aspects of A General Programming Language, Norwegian Computer Center, Oslo, Norway.

2. Bang, S. Y., and Yeh, R. T. Notes on Relational Data Structures, in Current Trends in Programming Methodology, volume IV : Data structuring, Yeh, R.T., ed., Prentice-Hall, Inc., Englewood Cliffs, N.J., 1978, pp. 241-262.

3. Barnard, T. D., Elliott, W. D., and Thompson, D. H. EUCLID and MODULA, Sigplan Notices, 12-3 (1973) 70-84.

4. Bergstra, J. What is An Abstract Data Type?, Information Processing Letters, 7-1 (1978) 42-43.

5. Black, A. P. Exception Handling and Data Abstraction, Report RC8059, IBM, Thomas J. Watson Research Center, Yorktown Heights, New York.

6. Brand, D. A Note on Data Abstractions, Sigplan Notices, 13-1 (1978) 21-24.

7. Brinch Hansen, P. The Architecture of Concurrent Programs, Prentice-Hall, Inc., 1977.

8. Burstall, R., and Goguen, J. Putting Theories Together to Make Specifications, Proceedings of The 1977 International Joint Conference on Artifical Intelligence, 1977, pp. 1045-1058.

9. Cassel, D. Programming Languages One, Reston Publishing Company, Inc., Reston, Va, 1972.

10. Chang E., Kaden, N. E., and Elliott W. D. Abstract Data Types in EUCLID, Sigplan Notices, 12-3 (1978) 34-42.

11. Chaudhary, B. D.,and Sahasrabuddle, H. V. Suggestions about A Specification Technique, Sigplan Notices, 13-12 (1978) 25-28.

12. Cress, P., Dirksen, P.,and Graham, J. W. FORTRAN IV with WATFOR And WATFIV, Prentice-Hall Inc., Englewood Cliffs, N.J., 1970.

13. Dahl, O. J.,and Hoare, C.A.R. Hierarchical Program Structures, in Structured Programming, Dahl, O.J., Dijkstra, E. W.,and Hoare, C.A.R., eds., Academic Press,

New York, 1972, pp. 175-220.

14. Demers, A., Donahue, J.,and Skinner, F. G. Data Types As Values: Polymorphism, Type-Checking, Encapsulation, Proceeding of 5th ACM Conference on Principles of Programming Languages, Tucson, 1978, pp. 23-30.

15. Enderton, H. A Mathematical Introduction to Logic, Addison Wesley, Reading, Mass., 1974.

16. Flon L. A Survey of Some Issues Concerning Abstract Data Types, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa 15213, 1974.

17. Flon, L. Program Design with Abstract Data Types, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa. 1975.

18. Gannon, J. D., and Horning, J. J. Language Design for Programming Reliability, IEEE Transaction on Software Engineering, 1-2 (1975) 179-191.

19. Gannon, J. D. An Experimental Evaluation of Data Type Conventions, Communications of ACM, 20-8 (1977) 584-595.

20. Gaudel Marie-Claude, Algebraic Specification of Abstract Data Types, Rapport de Recherche No. 360, INRIA, Le Chesnay, France, 1979.

21. Geschke, C. M., Morris, J. H. Jr., and Satterthwaite E. H. Early Experience with MESA, Communications of ACM, 20-8 (1977) 540-552.

22. Goguen J. A. Some Design Principles and Theory for OBJ-0, A Language to Express and Execute Algebraic Specificaions of Programs, Computer Science Department, UCLA, L.A., Cal.

23. Goguen, J., Thaitcher, J., and Wagner, E. G. An Initial Algebra Approach to the Specification, Correctness, And Implementation of Abstract Data Types, in Current Trends in Programming Methodology, Volume IV: Data Structuring, Yeh, R. T., ed., Prentice-Hall, Inc., Englewood Cliffs, N.J., 1978, pp. 80-149.

24. Gratzer, G. Universal Algebra, Van Nostrand, New York, 1967.

25. Gries, D., and Gehani, N. Some Ideas on Data Types in High-Level Languages, Communications of ACM, 20-6 (1977) 414-420.

26. Guttag, J. V., and Horowitz, E., and Musser, D. R. Abstract Data Types and Software Validation, Communications of ACM, 21-12 (1978) 1048-1064.

27. Guttag, J. Abstract Data Types And The Development of Data Structures, Communications of ACM, 20-6 (1977) 396-404.

28. Guttag, J. V., Horowitz, E., and Musser, D. R. The Design of Data Type Specifications, in Current Trends in Programming Methodology, Volume IV: Data Structureing, Yeh, R. T., ed., Prentice-Hall, Inc., Englewood Cliffs, N. J., 1978, pp. 60-79.

29. Hoare, C.A.R. Recursive Data Structures, International Journal of Computer And Information Science, 4-2 (1975) 105-132.

30. Hoare, C.A.R. Data Structures, Current Trends in Programming Methodology, Volume IV: Data Structuring, Yeh, R. T., ed., Prentice-Hall Inc., Englewood Cliffs, N.J., 1978, pp. 1-11.

31. Hoare, C.A.R. Notes on Data Structuring, in Structured Programming, Dahl, O. J., Dijkstra, E. W., and Hoare, C.A.R., eds., Academic Press, 1972, pp. 83-174.

32. Ichbiah, J. D., Heliard, J. C., Roubine, O., Barnes, J. G. P., Krieg-Brueckner, B., and Wichmann, B. A. Rationale for The Design of The ADA programming Language, Sigplan Notices, 14-6, (1979).

33. Iglewski, M., Madey, J., and Matwin, S. A Contribution to An Improvement of PASCAL, Sigplan Notices, 13-1 (1978) 48-58.

34. Jensen, K., and Wirth, N. PASCAL User Manual and Report, Springer-Verlag, 1974

35. Johnson, R. T., and Morris, J. B. Abstract Data Types in The MODEL Programming Language, Sigplan Notices, 8-2 (1976 special issue) 36-46.

36. Jones, D. W. A Note on Some Limits of The Algebraic Specification Method, Sigplan Notices, 13-4 (1978) 64-67.

37. Kamin, S. Some Definitions for Algebraic Data Type Specifications, Sigplan Notices, 14-3 (1979) 28-37.

38. Lampson, B., et al. Report on The Programming Language EUCLID, Sigplan Notices, 12-2 (1977).

39. Ledgard, H. F., and Taylor R. W. Two Views of Data Abstraction, Communications of ACM, 20-6 (1977) 382-384.

40. Liskov, B., and Zilles, S. Progrmming with Abstract Data Types, Sigplan Notices, 9-4, (1974), 50-59.

41. Liskov, B. H., and Berzins V. An Appraisal of Program Specifications, in Research Directions in Software Technology, Wagner, P., ed., MIT Press, Cambridge, Mass., 1979, pp. 276-301.

42. Liskov, B., Snjder A., Atkinson, R., and Schaffert, C. Abstract Mechanisms in CLU, Communications of ACM, 20-8 (1977) 564-576.

43. Mealy, G. H. Notions , in Current Trends in Programming Methodology, Volume IV: Data Structuring, Yeh, R. T., ed., Prentice-Hall Inc., Englewood Cliffs, N.J., 1978, pp. 12-29.

44. Mitchell, J. G., and Wegbreit, B. Schemes: A High-Level Data Structuring Concept, in Current Trends in Progeramming Methodology, Volume IV: Data Structuring, Yeh, R. T., ed., Prentice-Hall Inc., Englewood Cliffs, N.J., 1978, pp. 150-184.

45. Morris, J. H. Jr. Types Are Not Sets, Proceeding 1st ACM Symp on Principles of Programming Languages, Boston, 1973, pp. 120-124.

46. Morris J. H. Jr. Protection in Programming Languages, Communications of ACM, 16-1 (1973) 15-21.

47. Parnes, D. L., Shore, J. E., and Weiss, D. Abstract Types Defined As Classes of Variables, Sigplan Notices, 8-2 (1976 special issue) 149-154.

48. Rivers, J. D., and Spencer, H. Readability and Writability in EUCLID, Sigplan Notices, 12-3 (1978) 49-56.

49. Ross D. T. Toward Foundations for The Understanding of Type, Sigplan Notices, 8-2 (1976 special issue) 63-65.

50. Schwartz J. T. Program Genesis And The Design of Programming, in Current Trends in Programming Methodology, Volume IV: Data Structuring, Yeh, R. T., ed., Prentice-Hall Inc., Englewood Cliffs, N.J., 1978, pp. 185-215.

51. Shaw, M. Research Directions in Abstract Data Structures, Sigplan Notices, 8-2 (1976 special issue)

66-68.

52. Shaw, M., Wulf, W A., and London. R. L. Abstraction And Verification in ALPHARD: Defining And Specifying Iteration And Generations, Communications of ACM, 20-8 (1977) 553-564.

53. Standish, T. A. Data Structures - An Axiomatic Approach, in Current Trends in Programming Methodology, Volume IV: Data Structuring, Yeh, R. T., ed., Prentice-Hall Inc., Englewood Cliffs, N.J. 1978, pp. 30-59.

54. Stoy, J. Denotational Semantics, MIT Press, Cambridge, Mass., 1977.

55. Tonenbaun, A. S. A Comparison of PASCAL and ALGOL 68, The Computer Journal, 21-4 (1978) 316-323.

56. Tennent R. D. On A New Approach to Representation Independent Data Classes, Acta Information, 8-4 (1977) 315-324.

57. Venema T., and Rivers, J. D. EUCLID And PASCAL, Sigplan Notices, 12-3 (1978) 57-69.

58. Wulf W. A. Abstract Data Types: A Retrospective And Prospective View, Computer Science Department, Carnegie Mellon University, Pittsburgh, Pa, 1980.

59. Wulf, W. A., London, R. L.,and Shaw, M. An Introduction to The Construction and Verification of Alphard Programs, IEEE Transactions on Software Engineering, 2-4 (1976) 253-264.

A Survey of Data Type Specification Methods

by

SHIOWJY FAN

B.S., Fu-Jen Catholic University(Taiwan, R.O.C.), 1979

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements of the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1982

ABSTRACT

Different programming languages provide different features for defining abstract data types. Some use the existing language's structure to extend its features to define a type, some use a data structures plus subroutines package to define a type, and others provide mechanisms using operators and equational axioms to define a new data type.

This paper surveys five basic approaches to data type definition and the programming languages and theories which support these methods.

A large portion of this paper deals with the representations of the methods for data type definition. Examples are given to support these constructions. Relevant topics such as type recursion, constructs for type conversion, paramaterization, and exception handling are discussed as well.

In the last section some conclusions and future trends are given.