D. B. Lomet

Objects and Values: The Basis of a Storage Model for Procedural Languages

Abstract: A model for storage in procedural languages is presented. Its fundamental notion is to strictly distinguish values from storage objects. Several difficulties in current languages are resolved in this model, e.g., the problem of flexible locations [1] and the meaning of the term type [2, 3, 4]. In the light of the storage object/value dichotomy, several notions are found to be covered by the term type. The implications of the model are explored with respect to the more conventional data constructs of procedural languages as well as to sets and how they might be provided. Finally, data extension mechanisms are considered. Whereas the treatment here is not complete, the template concept introduced in the model does suggest a useful framework for providing the much talked of but seldom realized benefits of data extensibility.

Introduction

Much scientific development is a process of going from the concrete, special case to interesting generalizations. Such has been the history of procedural programming languages. In particular, this progression can be seen in the area of data and storage. Table 1 presents some of the important languages together with the data and storage notions given wide exposure by that language.

In examining Table 1, it becomes clear that a rather remarkable amount of progress has occurred since FORTRAN first appeared. Some confusion and controversy has also arisen. The word type was introduced in FORTRAN to describe variables and to make efficient compilation possible. Although ALGOL 68 substitutes the word mode for type, the historical situation is that, by and large, a single term has been used to cover the multitude of concepts listed in Table 1. The thesis presented here is that the word type currently includes several distinct notions and that the situation would be clarified if these notions were distinguished by unique terms.

The basis for much of the following discussion lies in strictly separating the notion of *value* from that of storage *object*. Most language designers and users probably believe that their language maintains exactly that separation. Our contention is that this distinction has broken down during the process of generalization that has accompanied language evolution. Resurecting this distinction in a rigorous way exposes several of the concepts embedded in the word type as well as resolving some other difficulties. First it is necessary to establish what we mean by these terms. Table 2 enumerates the properties that characterize values and objects.

The relation between values and objects is twofold. First, references are values that are used to manipulate objects. Second, values may be stored in and retrieved from some objects. We call an object that can contain a single value a cell. The term variable has sometimes been used for a cell, but it also can mean the symbolic name used to identify the cell. It should be emphasized

Table 1 Languages and the data and storage features given exposure by them.

Year	*Language*	*Feature*
1957	FORTRAN [5]	scalars and fixed size arrays
1961	COBOL [6]	structures
1963	ALGOL 60 [7]	local (stack) variables, settable array bounds
1965	PL/I [8]	dynamic allocation and freeing, pointers, strings
1966	ALGOL W [9]	constrained references
1969	ALGOL 68 [10]	unions, flexible arrays, recursive types
1970	SIMULA 67 [11]	class definitions
1970	Pascal [12]	ranges, sets
1971	EL1 [13]	type variables, user defined types

Table 2 Properties of values and objects.

	Property	*Discussion*
Values	atomic	No references to components.
	immutable	Values may be replaced by other values, but it is meaningless to talk of changing values. If 1 is added to 3, the effect is not to change 3 to 4 but to replace 3 by 4.
	storable	Values can become the contents of (parts of) storage objects.
	returnable	Operators may produce values as results. Further, any result returned is always a value.
Objects	constructible	Whereas operators may return only values, a side effect of certain operators is the creation of storage objects that persist over time.
	referable	Reference values are used to specify the storage objects or their components that are to be manipulated.
	changeable	Storage objects contain the state of the computation. If an object is changed, subsequent operations involving references to it reflect the change.
	deletable	Some languages permit storage objects to be deleted. Subsequent references to the deleted object are erroneous.

that there may be other forms of objects, e.g., processes, locks, modules, etc. We consider in what follows only cells, aggregates (i.e., groupings of objects that can be accessed via a common reference and that have a common lifetime), and sets (i.e., objects that can contain several values).

Cells and values

Associated with each cell is a predicate that must be satisfied by any value that is to be stored in or retrieved from the cell. The predicate is the first concept introduced that has been covered by the word type. We use *type* to denote this predicate. Note, however, that here we are taking a term that is currently imprecise and giving it a very specific meaning; we give other names to the other notions covered imprecisely by the word type. The view of types as sets, which has been discussed by Reynolds [4], is equivalent to our use of the term type.

Several recent languages [10, 12, 13, 14] have types as values, and further they provide a type calculus by supplying operations whose results are types. The methodology for this is as follows:

1. As a basis, one is given a collection of primitive types, e.g., integer, real, boolean, character, and the values that satisfy them. There is no requirement that the primitive types be satisfied by disjoint sets of values. It is entirely reasonable for values satisfying the type integer to also satisfy the type real. This would permit the so-called "natural conversions" from integer to real. In the other direction, an entier function [7] would be required to convert real values to integers because there are real values that are not integer values.
2. Type operators are drawn naturally from set theory. Some potential operators follow:
   a. union The disjunction of two type predicates specifies the union of the sets of values satisfying either type.
   b. range Sometimes called "subrange," the range operator defines the subset of a type whose values are ordered by specifying an upper bound and a lower bound that are imposed on members of the resulting type [12].
   c. enumeration A type predicate can be specified by enumerating the values that satisfy the type. This is sometimes present in only a restricted form, e.g., only identifiers may be listed [12, 14]. However, a general capability can be provided by supplying a constant operator, which, given any value, yields a type that is satisfied by precisely that one value. Enumeration is achieved by applying union to several of the results produced by the constant operator.
3. At times, one does not wish to specify any constraints on the values that can be stored in a particular cell. For this reason, we introduce the type general, which is satisfied by all values. This type provides the means of supporting so-called typeless languages.

More types are needed as new values are introduced, (e.g., reference types), but we defer consideration of these until the values themselves are introduced.

Many other set theoretic operations are not generally included in a type calculus. An operator embodying the axiom of separation, i.e., designating a subset of a given type by means of a more or less arbitrary predicate applied to members of the original type, is not included. Neither are such simpler operators as intersection, difference, and Cartesian product. There are two reasons for excluding these operators: Implementation problems can be formidable; undecidability can become a problem.

Both of these difficulties arise when the problem of the equivalence of types must be faced. When types can be recursively defined, as in ALGOL 68 [10], determining type equivalence can be very difficult. One must, in fact, be careful to ensure that the recursive types are indeed well defined. Lewis and Rosen discuss these issues in a recent paper [15]. Thus, the type operators are very carefully chosen. Whereas all types are value predicates, all value predicates are decidedly not types.

The operations involving cells, the cells being identified by means of references to them, consist of the storing and retrieving of values from them. The storing operator is generally called assign, and it has two operands, a target cell and a source that is either a value or a means of generating such a value. The effect of executing assign is to replace whatever is currently stored in the cell with the value produced by the source. The retrieving operation is frequently called val and, given a cell reference as an operand, returns the current contents of the cell as its result.

Templates and aggregates

Cells and aggregates are storage objects. As such they can be created and destroyed. When creating a cell, a description of the cell must be provided to the create operator. Because a type is associated with each cell, it is natural to use a type as the description when a cell of the given type is desired. There is no harm in this. However, the extension of this idea to aggregates creates difficulties.

Typical aggregates in existing languages are arrays, structures (records), and files. These aggregates are usually treated as storage objects, not as values. How do we tell the difference? In IBM PL/I [16], although not in the ANSI standard [17], and in ALGOL 60 [7] and Pascal [12], the following properties of storage objects also hold for aggregates:

1. Array operators, if any, are simply shorthand notations for iterations in which a scalar operation is performed on each component of the aggregate(s).
2. Functions never return aggregates as results.
3. Components of aggregates can be updated.
4. References to aggregates and to their components can be acquired in those languages that provide reference types.

Unlike the above languages, ALGOL 68 attempts to treat aggregates as both objects and values. This results in the notion of flexible locations, which is the cause of some ad hoc restrictions. We explore this more fully later.

An aggregate is a collection of objects, all sharing a common lifetime, i.e., the objects are all created by a single execution of the create operator and are all destroyed by a single execution of the free operator. In addition, given a reference to the aggregate, it is possible, via a selection operation, to acquire a reference to any of the components of the aggregate. Each component is uniquely named within the aggregate. Hence, the aggregate reference together with one of these names, called selectors, identifies the desired component in a system-unique way.

Using types to describe aggregates is clearly inadequate. Each cell of an aggregate must be described, via its type, but, in addition, these cells must be given selectors and must be "aggregated" in whatever way is desired. For this reason, a second term is required. We call the specification of an aggregate a *template*, following the terminology of Wegner [18]. Template denotes the second meaning that is sometimes attributed to the intuitive notion of type. Given that types can be used as templates for cells, together with the confusion about whether aggregates are objects or values, it becomes understandable why types and templates are not universally distinguished.

Once types and templates are distinguished, it becomes necessary to introduce template operators as well as type operators. Most languages provide notation for expressing the results of these "operations" even when they do not treat templates as values. Typically, array and structure construction is provided. File construction is also frequently provided but is not usually well integrated with the other mechanisms. We wish to treat these constructions as explicit operators in a template calculus that augments our previous type calculus.

The template operators introduced are, of course, sensitive to the forms of aggregates that are desired. A rather large number of aggregate forms have been used, e.g., structures, arrays, files, tuples, records (similar to structures), strings, etc. Instead of trying to define operators for all these different forms, we suggest the following alternative. Most aggregates are characterized by the following set of properties associated with each component:

1. A selector that names the component.
2. A template that describes the component. The component is itself, of course, a storage object.
3. The current value of the component, when the component is a cell.

The operators we describe permit 1) the association of selectors with templates that describe the components that the selectors identify and 2) the composition of previously constructed templates. All aggregate forms are thus unified. These operators are

1. row (*selector set*,*template*). The resulting template specifies an aggregate, all of whose elements are simi-

lar. For example, an array with a range of integers as a selector set can be specified. Further, using the constant operator, a single element aggregate can be specified with any desired selector name. This is useful in conjunction with the compose operator (below) in specifying structures.
2. compose(*template*1,*template*2). The resulting template specifies an aggregate with all the components specified by both *template*1 and *template*2. In particular, nonhomogeneous aggregates can be specified via this operator, e.g., structures.

This view is elaborated more fully in [19]. Although this view is attractive to us, it is by no means essential to the discussion of fundamental concepts that we have given.

Let us now illustrate one of the practical advantages of the fundamental concepts we have been presenting. To do this, we examine the roles played by cells and aggregates from a slightly different perspective.

The fundamental purpose of aggregates in higher level languages is to support a restricted form of address computation. Thus, given a reference to an aggregate and a selector value, a selection operation computes a new address, i.e., the address of the component specified by the selector. Such address computation is controlled in high level languages by the requirements that

1. References to aggregates originate as a result of a create operation, either explicit or implicit. Arbitrary addresses cannot be computed.
2. Selection operations have a valid address as an argument, perform a defined selection, and produce an address to a component of the aggregate originally referenced.

The purpose of cells is to provide repositories for values, i.e., to maintain state information. To make this possible, the assign operation is provided. In our methodology, the only function of assignment is to store a value in a cell. In no way does it interact with address computation.

ALGOL 68 does not strictly segregate the functions of assignment and address computation. These functions intersect in the case of flexible locations, which arise, e.g., as a result of the union operation. We have distinguished types from templates, based on our distinction between values and objects. Further, we have restricted union to apply only to types, not to templates. ALGOL 68 does not make these distinctions, and hence it does not prevent unions of what in our view would be templates. Thus, it permits the declaration of a mode (template?):

```
mode flexloc = union(struc(real a,real b),struc
                                    (real b,real c))
```

If a variable X is declared to be of the mode *flexloc*, it becomes possible to change the selector set of X from (a, b) to (b, c) by means of an assignment to X. If taking a reference to a of X were permitted, every use of this reference would require checking to determine that the alternative for X that contained a of X was the current "value" of X. This checking would also be required if a of X were passed by reference as an argument. Because of the uncertainty about the source of a parameter, checking would frequently be required even if the parameter were not of this form. Distinguishing templates from types permits unions to be supported for types while preventing unions involving templates, thus preventing flexible locations. The explanation for the restriction arises in a completely natural way from the value/object distinctions being made. Because ALGOL 68 has no way of preventing flexible locations, it must cope with them. It does this by imposing a set of ad hoc restrictions that prevent a user from acquiring a reference to a component of a flexible location. Most of the function of flexible locations is provided via aggregate values, introduced later, while avoiding these difficulties.

Reference values

Storage objects themselves cannot be assigned from one variable to another, incorporated directly into programs, or passed as parameters. Rather, a storage object is manipulated by means of a value that uniquely denotes it. This value is called a reference in ALGOL 68 and a pointer in PL/I. We use the term *reference*. The concept of reference occurs in most high level languages, even when references are not treated as values, i.e., are not assignable to variables. For example, FORTRAN passes arguments "by reference." Further, if one wishes to describe the assignment operator in terms of values that the system can manipulate, then the notion of reference is unavoidable.

If references are values in a language, it becomes necessary to specify cells that can contain them, i.e., types that are satisfied by references are needed. In PL/I the type POINTER is provided, which is satisfied by all references (pointers). Because PL/I only offers POINTER cells, the type checking that involves the variables pointed at can only be done at run time. In most implementations, this checking is not done at all because of the serious performance penalty. In order to perform at least some of this type checking at compile time, as well as providing more comprehensive syntax checking, reference constraints are utilized by such languages as ALGOL 68 and Pascal.

In our methodology, where types are themselves values, what is required is an operator that, when given a constraint, produces a type that is satisfied only by references to objects that satisfy the constraint. We call this

operator the ref operator. An operand of ref is a value that we call a *constraint*, and constraints denote the third concept included in the intuitive notion of type. This is a distinct concept, because a constraint is, in fact, a predicate over storage objects, and none of the previous terms play this role.

With Pascal, constraints uniquely specify the form of object required. That is, a constraint must be satisfied by objects that were all created by the same template. Thus, in Pascal it is possible to use templates as constraints and thus not introduce special constraint values. However, ALGOL 68 permits constraints that are satisfied by any of several forms of object. One example of this occurs in the reference type

ref ( ) int

which is satisfied by references to any vector of integers. ALGOL 68 partially distinguishes template from constraint by means of the distinction between actual and formal declarers.

The need for constraints arises in other contexts than as operands of the ref operator. In any language that passes arguments by reference, e.g., FORTRAN and PL/I, the description associated with a parameter must be considered as a constraint. With FORTRAN, a parameter description may specify only a single form of object, which must be a cell or array of cells. Therefore, templates can be used as parameter descriptions. It is, of course, special cases like this that contribute to the confusion concerning types, etc. Although PL/I calls parameter descriptions types, it in fact permits some flexibility in these descriptions, i.e., a parameter can take one of several forms. For example, the declaration

DCL *X* CHAR(*)

requires that the argument passed to *X* be a character string of some fixed length. However, the length of the string is unspecified. One cannot declare such a "type" to be BASED, and hence be allocated, because CHAR(*) is not a template, but rather describes many templates.

Having demonstrated the need for constraints, we now show that it is not necessary to introduce a new set of values to serve this purpose. Rather, we can make use of values that, given our previous type and template operators, we can already generate. The effect of doing this does not alter the fact that a constraint is fundamentally distinct from type and template concepts. Rather, the impact is purely pragmatic, i.e., the using of previous operators to manipulate and generate constraints.

Although constraints are predicates over storage objects, they may be expressed in terms of predicates over the templates that describe storage objects. Because we are treating templates as values, predicates over templates are, therefore, types. Thus, it becomes possible to apply the type operators to the construction of constraints. A constraint satisfied by objects that can all be described using a single template can be formed using the constant operator with the template as its operand. Further, the union operator can be used to form a disjunction of constraints. Types of the above form can serve as operands of the ref operator, yielding a type that is satisfied only by references to objects that satisfy the constraint, i.e., are described by templates that satisfy the type that is the operand of ref.

Additional constraints (and hence types satisfied by templates) are very useful. For example, the type satisfied by any template, denoted any, when used in ref(any), produces a type that is satisfied by all references. Thus, it is equivalent to the PL/I POINTER type. Predicates over ranges are useful in forming constraints that are satisfied, for example, by vectors of unspecified bounds. Operators for producing such predicates are explored in [19].

Aggregate values

The preceding sections explored the distinction between values and objects. During this exploration, aggregates were treated exclusively as storage objects. What we wish to discuss here is the treatment of aggregates as values. Languages such as APL [20], ALGOL 68 [10, 21], and ANSI PL/I [17] support operators and procedures that can return aggregates as results and permit aggregate assignments. The problem that we wish to address is how to reconcile these aggregate values with object aggregates. In particular, how are aggregate values introduced so as to avoid the problems that were discussed in the introduction?

We propose to introduce aggregate values in such a way that the distinction between values and objects is rigorously maintained. Thus, whereas aggregate values have components, it is not possible to either construct a reference to a component or to alter the value of a component. In APL, ALGOL 68, and PL/I (strings), this distinction is compromised, either through overlays, elaborate rationalizations such as flexible locations, or by treating assignment as an operator that is different in kind from other operators and procedures. In fact, in none of the above languages is it possible to reproduce the effect of the assignment operator by means of a procedure. This is the case for all APL variables, because APL does not pass arguments by reference. In PL/I, substring assignment cannot be handled, because when SUBSTR appears anywhere except to the left of the assignment operator, it designates the SUBSTR function that produces a string value rather than the SUBSTR pseudovariable that designates the location of a substring. In ALGOL 68, one cannot acquire a reference to a component of a flexible location, and hence a procedure cannot update these components.

There is one aspect of aggregate values in most languages that we want to be sure to preserve. An aggregate value should be characterized in the same way as a cell aggregate, i.e., by

1. A selector set,
2. A descriptor for each component (i.e., a template),
3. A value for each component.

For example, consider two aggregate values, which we denote using angle brackets in what we hope is an obvious notation.

a. $\langle a$:int := 1, b:real := 1.0$\rangle$
b. $\langle a$:int := 1, b:union(real, char) := 1.0$\rangle$

Despite the fact that both of these aggregates have components with equal values, the aggregates themselves are not equal because their second components are described differently. Property 2) above is a very useful one for aggregate values for several reasons: It suggests a way of forming aggregate values, it suggests a way of providing types satisfied by aggregate values, and it increases the amount of type checking that is possible at compile time. It is this characterization of aggregate values, particularly property 2), that has created much of the confusion between aggregate values and object aggregates. The approach taken here carefully avoids this object/value confusion while providing almost all the capability that ALGOL 68 provides by means of flexible locations.

We form aggregate values from references to object aggregates by means of an operation we call enclose. Enclosing a reference to an object aggregate yields an aggregate value: whose selector set equals the selector set of the object aggregate, whose components are described by types that are enclosed forms of the templates that describe the corresponding components of the object aggregate, and whose values consist of the enclosed forms of the components of the object aggregate. These enclosed values include the current values contained by the components of the object aggregate.

Aggregate values can be subjected to selection operations, but the result of a selection is not a reference to the selected component but to the value of the component. The enclose operator is implemented by merely copying the cell aggregate and returning an implicit indirection to the copy. All cells to which the aggregate value is subsequently assigned can share, via the indirection, this copy of the original aggregate. Because it is not possible to update aggregate values, no unexpected side effects can occur because of this sharing. Reclaiming the storage of the aggregate can be accomplished by maintaining reference counts, by general purpose garbage collection, or by normal stack storage reclamation.

Cells with types satisfied by aggregate values can be provided in a similarly straightforward manner. Constraints that are satisfied by various forms of object aggregates can already be constructed. What is required is an operator, enclose-type, that converts such a constraint into a type that is satisfied precisely by aggregate values formed from references to object aggregates that satisfy the constraint. The type calculus for aggregate values is thus exactly as powerful as the constraint calculus that is supported for object aggregates.

Having provided aggregate values in addition to object aggregates, we now wish to consider the advantages derived from this view. In particular, we consider how aggregate values can be used to support certain data forms of existing languages while avoiding complications that give rise to ad hoc restrictions. Strings in PL/I, both character and bit, can very naturally be treated as enclosed vectors of characters and bits, respectively. Thus, they can be treated as "scalars" with respect to assignment and comparison operators, while being susceptible to selection operations, i.e., SUBSTR operations. Selection operations on aggregate values yield, not a reference to the component specified, but the value of the component. This is exactly the manner in which the SUBSTR function works. Only the SUBSTR pseudo-variable, which permits the assignment operator to alter a specified substring, is not supported with this characterization. A cell that can contain varying length strings can be described by a type that is a union of types, each one satisfied by a different length enclosed vector. Similarly, ALGOL 68 flexible locations, e.g., those described by unions of aggregate modes, can by characterized as unions of enclosed aggregate types. This characterization provides a natural explanation for the prohibition on taking references to components of flexible locations. Currently, this is merely an ad hoc limitation.

Assignments to components of flexible locations, as to substrings of PL/I strings, can be regarded as follows. The aggregate value in the target of such an assignment is accessed in its entirety. A new aggregate is formed from this value and the value from the source, the selected components being set to the source value and the resultant aggregate value assigned to the target. Whether such assignments should be supported at all is a question that the reader will have to answer for himself.

Sets

Several languages support some form of set data and operations on sets. Among these languages are Pascal [12], MADCAP [22, 23], and SETL [22, 24]. Further, the word set is often used to describe collections of data in data base systems, e.g., "data set" is a term commonly used in IBM data management systems, and relations in relational data base systems are usually considered as

sets of tuples [25]. While the intuitive notion of set provides a reasonable justification for calling any of the preceding entities sets, careful definition in a programming language context requires that some rigorous distinctions be made. Thus we consider two kinds of sets. These two forms of sets reflect the distinction we have been making throughout the paper, i.e., the distinction between object and value. Thus sets as objects and sets as values are considered separately.

• *Sets as objects*

We have discussed cells as objects that can each contain a single value. A set object is an object that can contain zero or more values. Note that it is not a cell into which a set value can be stored. Set values are discussed below. Because a set object is not a value, it cannot be passed around via assignment or returned as a result of an operation or procedure. However, references to set objects are, as expected, values and thus can be treated in these ways. Hence, whereas sets only contain multiple values, the effect of having sets of set objects, or sets of objects of any form, can be realized by specifying a set of references to the objects desired. Further, as a natural consequence of this view, an object can be "contained" simultaneously in several such sets because its reference can freely belong to the several sets.

Programming languages use the assign operator to store a value in a cell, thereby destroying the existing value in the cell. With set objects, however, we wish to be able to add new values or to remove some of the existing values without affecting the other values in it. Thus we require a new group of operators for set objects. Whereas several possible variants for the operators exist, we specify the following ones for illustrative purposes.

1. insert(*ref to set object*, *value*) This operation includes the value of operand two in the set object referenced by operand one. If the set is "full" (see below), an error condition occurs.
2. delete(*ref to set object*, *value*) This operation removes the value of operand two from the set referenced by operand one. If the set does not contain the given value, no change is made.
3. empty(*ref to set object*) This operation removes all values from the set referenced.

The above operators do not produce values as results. Rather, they are the set analogs of cell assignment. Thus their execution results in side effects on storage. We need to retrieve values from sets and to test the contents of sets. For this purpose, the additional operators below are introduced.

4. member (*ref to set object*) → *value* The value produced by this operation is some member of the set referenced by its operand. In a particular language, it might be desirable to specify an ordering on elements as a set attribute and thus define the member chosen. We do not pursue this possibility here. In order to achieve the effect of sequencing through the members of a set, the result of this operation must be removed from the set before requesting a member again. If there are no values in the set, an error condition occurs.
5. in (*ref to set object*,*value*) → *boolean value* The result is true if the value specified by the second operand is in the set referenced by the first operand. Otherwise, the result is false.
6. null (*ref to set object*) → *boolean*: The result is true if the set referenced by the operand contains no values. Otherwise it is false.

The normal set theoretic operations, e.g., union, intersection, have not been provided as primitives for set objects. These operations must produce new sets as results. With set objects, this requires the allocation of a new set object and the returning of its reference. We believe it is a bad practice to provide primitives that implicitly allocate new objects during the course of their execution. The set theoretic operations can be programmed, however, in terms of the primitives already provided.

We have yet to discuss how set objects are specified and allocated. We can, of course, already specify sets of values. These specifications are the types. A type describes the set of values, one member of which can be contained in a cell. Because types have been designed so as to permit their equivalence and satisfiability to be decidable, they form a natural basis for the description of set objects. Thus, to construct a template for a set object, we introduce the set operator such that

set(*type*) → *template*

where *template* describes a set object that can contain zero or more of the values that satisfy the type.

Sets are enormously useful and a natural construct for specifying a broad range of algorithms. Given this, the question arises as to why they are not provided in more languages. The reason is primarily that it is very difficult to choose a representation for sets that uses storage efficiently and simultaneously permits an efficient implementation of the desired set operations. This is an area of ongoing research [26]. Given our model of storage, this problem can be eased considerably. We use a technique analogous to that used in PL/I to ease the problems with VARYING strings [16], i.e., we permit the user to specify the maximum number of elements that a set is

to contain, thus bounding the storage requirements and suggesting implementation strategies for the set operations. For this purpose, the cardinality operator is introduced as follows:

cardinality(*template for set, integer*) → *template*

where the resulting template specifies a set object that can contain at most the number of values specified by operand two.

Whereas the provision of a maximum number of elements for a set object is an extra burden in some cases, this information is frequently available to the user but he has no way of supplying it to the language system. Further, such information can be a valuable additional constraint on the program that is useful for error detection.

Set objects correspond approximately to the sets of MADCAP, where the set variables must be regarded as containing references to the set objects. Operations on one set variable can cause changes to another set variable if the two variables reference the same set object. MADCAP has thus been described by Low [26] as a "pointer language."

• *Sets as values*

Values, as discussed previously, are distinguished from storage objects in that they are atomic and immutable, can be passed from one object to another, and can be returned as results of operators and procedures. In particular, a value may be assigned to and hence become the contents of a cell. In the case of sets, then, we can distinguish a cell that contains a set value from a set object that contains a number of values.

To support set values requires the introduction of operations that permit their construction and manipulation and the construction of types that are satisfied by set values. By introducing set values after set objects, we can make use of the same device we have used before for producing values from objects, i.e., we can use the enclose operator to produce a set value from a set object. As with aggregates, not only do the set values reflect the current contents of the set object. The characteristics of the set object itself also characterize the enclosed set value produced from it. Similarly, types that are satisfied by set values can be formed by means of the enclose-type operator. Given a constraint satisfied by a set object, enclose-type produces a type that is satisfied by the analogous set value, thus permitting the construction of cells that can contain set values.

Some of the operations on set objects can be carried over to work on set values. These are in, member, and null. The remaining operators have to be alterred so as to return a modified set value rather than modifying a set object in place. We prefix the original object operators with a v and hence introduce the operators v-insert, v-delete, and v-empty to perform these functions.

Set values provide the natural programming language analog of the mathematical concept of a set. The usual set theoretic operations, e.g., union, intersection, relative complement, etc., can all be provided. These can either be programmed in terms of the preceding primitives or be supplied directly as primitives themselves. Which alternative is selected depends greatly on the representation(s) chosen for sets. We do not pursue this further. Set values are the form of set supported by SETL and Pascal. These sets possess what J. Schwartz [27] has called "pure value semantics" in that no set operation involving one group of sets can produce side effects on any other set. Further, assignment of the usual form for cells carries over to cells that can contain sets.

Extension mechanisms

The term type as we have defined it is but one of several concepts included in the intuitive notion of type. Morris argues that types are not sets or predicates [3]. What is generally meant by this is that a data type is defined not only by a set of values but also by a set of operations that manipulate the values. SIMULA [11], with its CLASS concept, was the language that first introduced a mechanism in which new objects and their operators could be defined together. The essentials required are a description of how the objects of the class are to be represented and the definition, as procedures, of the operators that are to manipulate the objects. This technique has been further refined by Brinch Hansen [28], who emphasizes the importance of protecting the underlying representation of the object from access and modification by procedures other than those that were defined as its operators. In SIMULA, the operators are designated as if they were components of the object. Liskov and Zilles [29], in addition to providing a clean syntactic treatment of class definitions, which they call clusters, change the operator naming scheme so that an operator is identified by its own name qualified by the name of the class, not the object. This seems a small change, but we believe it represents an important philosophical distinction that we exploit in what follows.

Data object descriptions (templates) have already been introduced. We wish to build on the template mechanism to provide a methodology for constructing new classes of objects. Thus *class* is the last term that we introduce that is included in the intuitive notion of type. Because we base classes on the notion of template, a class value is not needed. Rather, the approach we take is to specify class definitions as generalizations of template definitions. Hence template values serve as classes. As will be apparent, not all the problems associated with this view have been solved. Rather, we put

forth the view in an attempt to capture the essential concepts and directions of what is an ongoing research effort [15, 19, 29, 30].

In order to provide a coherent framework for class definitions, we need to consider templates in a new light. Templates for aggregates provide a description of the aggregate, but we need to understand the nature of that description. The view we consider here is as follows. The values that serve to index an aggregate are not to be considered as the selectors of the aggregate. Rather, these index values can be used to acquire selectors. The selectors themselves are functions that take a reference to an aggregate as an operand and return a reference to the specified component. Thus, the selector function already incorporates the index value in its definition, and this need not be provided as an operand. It is the template that serves to relate the index values to the selector functions. Thus we may regard a template as consisting of the following aspects:

1. A specification of how the aggregate is to be represented.
2. A dictionary of <index, function> pairs that provides a way of naming the operators that can manipulate the aggregate.
3. Perhaps a specification of how the aggregate is to be constructed and initialized, although this can be provided in 2) above.

Notice that this functional approach to selectors is fundamentally different from the functional data objects discussed by Reynolds [31]. In Reynolds' scheme, it is the allocated object that is the function that, when applied to a selector value (index), yields a reference to the component.

Our conception of templates and selectors has a significant consequence. It permits the addressing computation that selects a component of an aggregate to be decomposed into two parts, one of which can be executed prior to actually having a reference to the aggregate available, i.e., the selector function can be chosen by knowing only the index and the template. This part of the address computation, now exposed in the language itself, can be precomputed via constant propagation or subjected to common subexpression elimination by a compiler. Further, this view gives us a rationale for knowing the characteristics of the component selected. Note that with a general purpose select function that is given the aggregate reference plus an index value, or with Reynolds' functional data objects, all we know syntactically when dealing with heterogeneous aggregates is that the function will return some reference, but not its precise characteristics. This is not so when we have a selector function into which the index has already been incorporated. In this case we know that the type of its operand is a reference to a particular form of aggregate and that its return value is a reference to a particular form of object, i.e., the desired component. In order to acquire the selector functions from the indices of an aggregate, we introduce the index operation. The index operation takes a template and a value used to index the aggregate described by the template and returns the selector function.

Having established the basic framework, it is now a straightforward matter to augment it by permitting operations other than selector functions to be included in a template. This is the capability that Liskov and Zilles supply with function clusters. For example, to define a stack, they include push and pop operations in the cluster describing stacks and then refer to these operations in the body of the program by "indexing" into the template describing the stack, i.e., the stack cluster. Thus the pop operation is specified as stack $ pop.

A major virtue of the above view of templates is its ability to unify the data definition function. The same mechanism that explains basic aggregate objects also can provide 1) a way of providing differing representations for aggregates, i.e., use the same indices but change the selector functions; and 2) a way of defining new objects with their associated operations. However, there are some unresolved problems. Among them are the following:

1. If selector functions are to be provided by a user, when are they executed and how are references represented? One would like a given selector function, when applied to an object, to always produce the same reference value. Suppose we are realizing a large array using sparse array techniques [32]. Then the real storage locations of the elements change dynamically, and immediate execution of the selector function will not always produce the same result. It appears to be necessary to delay evaluation of the selector function, recording in an "extended" reference the selector to be used at access time. This causes references to become complex very rapidly, especially when multilevel selection is required.
2. How do we provide mechanisms by which a generic object specification can be defined? An example of this is to provide a generic specification for stacks and then permit subsequent specification of, e.g., integer, real, character, etc., stacks. SIMULA provides this via its subclass mechanism. Liskov and Zilles permit procedure declarations in which the return type is dependent on a parameter. Another alternative is to provide a way of constructing procedures with the required properties when needed. In any case, in the stack example, the push and pop operations must be tailored to the kind of stack that is specified. What-

ever method is chosen, an exceedingly careful assessment of the implications of the solution must be made to assure that the mechanism fits well into other language contexts.

3. How can flexible constraints be provided? Given the constant operator, one can always construct a constraint, one template at a time. However, this is not satisfactory in any practical sense. Constraints such as those provided in ALGOL 68 in which ( ) int is satisfied by any integer vector are highly desirable and cannot be provided by finite unions. Further, equivalence for templates must be defined so as to be decidable. The equivalence of templates depends partially on how we choose to regard the equivalence of procedures in the view of templates as including functions. A solution that requires every template that is constructed to be unique, i.e., not equivalent to any other, even those formed in the same way, is not satisfactory for the primitive objects.
4. How does the compose operation interact with the functions and the representing storage descriptions contained in its argument templates? When compose is executed, resulting in a new template, the aggregate specified has all the components specified in both arguments. Clearly then the selection functions associated with the index values must be modified to reflect the change in the aggregate from which the component is to be selected. The result is easy to describe when primitive aggregates are being treated. However, what are the effects when user-provided procedures are involved?
5. How is the concept of enclosing objects to form values applied to the more general situation involving user-defined extensions? During the enclosing of aggregates and sets, the operations that manipulated them were re-interpreted (actually new but analogous operators were provided) so as to work on the enclosed values. Is there a systematic way in which the operations on objects can be transformed to operate on values? Short of this, how are new operators for values specified that can access the values' representations?

Let us now state a general criterion that we feel an extension mechanism should possess. We feel it is the goal toward which efforts in the field should be directed. We also feel that there is still considerable progress that must be made before the goal is realized. The criterion is this: An extension mechanism should be capable of defining the primitive storage objects supplied with the language in terms of an even more primitive undifferentiated storage and some very simple operations on this storage. Further, the primitive values of the language should also be definable by extension. With our methodology, this would be done by enclosing objects that were constructed using the extension mechanism.

Summary

We have attempted to make a rigorous distinction between objects and values and have explored the consequences of this distinction. In so doing, we have identified four separate concepts that have been lumped under the intuitive notion of type. These are:

1. *type* We use the term type to mean a predicate that is satisfied by values and that characterizes a cell.
2. *template* A template is a description and specification of storage objects. Among the objects are cells whose templates may be types. Other objects, e.g., aggregates, must have templates that are not types.
3. *constraint* A constraint is a predicate that is satisfied by objects. One may, however, use types that are satisfied by the templates that describe the desired objects for this purpose. Hence, separate constraint values are not required.
4. *class* A class is a description and specification of storage objects that includes the operations that are to be used to manipulate the objects. In our view, a class can be considered as a generalization of the template concept.

The features of current languages have been examined in the light of the value/object distinction, which led us to introduce the enclose operation as a means of generating values that are based on objects. This view enabled us to have a full complement of aggregate values, paralleling the aggregate objects, without the difficulties involved with flexible locations. Sets were studied as they appear in some existing languages to demonstrate that the value/object distinction and the notion of enclosing can provide sets in object and value forms, both of which have proved useful.

Lastly, extension mechanisms were discussed. This area is the most incomplete despite many efforts extending over a period of at least ten years [33]. We have discussed them in terms that integrate them with the basic concepts already provided, but we have sketched only the framework in which such a mechanism might be defined. Some of the problems with extension mechanisms have been discussed in a series of questions. It is our hope that future work involving extension mechanisms will result in the design of one that satisfies the criterion that was expressed at the end of that section.

Acknowledgments

Several associates have been helpful in the development of the ideas presented in this paper, including R. Goldberg, W. H. Harrison, C. Lewis, and P. H. Oden. A spe-

cial debt is owed to M. A. Auslander, who, by pointing out shortcomings in their precursors, greatly assisted the emergence of these ideas.

References

1. K. Walk, "Modeling of Storage Properties of Higher Level Languages," *SIGPLAN Notices* (ACM) **6**, 146 (February 1973).
2. A. N. Habermann, "Critical Comments on the Programming Language Pascal," *Acta Informatica* **3**, 47 (1973).
3. J. H. Morris, "Types Are Not Sets," *Conf. Record of ACM Symposium on Principles of Programming Languages*, October 1973, p. 120.
4. J. C. Reynolds, "A Set-theoretic Approach to the Concept of Type," NATO Conf. on Techniques in Software Engineering, Rome, October 1969.
5. J. W. Backus, et al., "The FORTRAN Automatic Coding System," *AFIPS Conf. Proc., Fall Joint Computer Conference* **11**, 1957, p. 188.
6. "COBOL-1961: Revised Specifications for a Common Business Oriented Language," Dept. of Defense, U.S. Government Printing Office, Washington, D.C., 1961.
7. P. Naur, et al., "Revised Report on the Algorithmic Language ALGOL 60," *Commun. ACM* **6**, 1 (January 1963).
8. G. Radin and H. P. Rogoway, "NPL: Highlights of a New Programming Language," *Common. ACM* **8**, 9 (January 1963).
9. N. Wirth and C. A. R. Hoare, "A Contribution to the Development of ALGOL," *Commun. ACM* **9**, 413 (September 1966).
10. A. van Wijngaarden, B. J. Mailoux, J. E. L. Peck, and C. H. A. Koster, "Report on the Algorithmic Language ALGOL 68," MR101, Mathematisch Centrum, Amsterdam, October 1969.
11. O. J. Dahl, B. Myhrhaug, and K. Nygaard, *The SIMULA 67 Common Base Language*, Publication S-22, Norwegian Computing Center, Oslo, 1970.
12. N. Wirth, "The Programming Language Pascal," *Acta Informatica* **1**, 35 (1971).
13. B. Wegbreit, "The Treatment of Data Types in EL1," *Commun. ACM* **17**, 251 (May 1974).
14. B. L. Clark and J. J. Horning, "The System Language for Project SUE," *SIGPLAN Notices* **6**, 79 (October 1971).
15. C. H. Lewis and B. K. Rosen, "Recursively Defined Data Types," *Conf. Record of ACM Symp. on Principles of Programming Languages*, October 1973, p. 125.
16. *PL/1 Language Specifications*, Order no. GY33-6003-2, IBM United Kingdom Laboratories, Winchester, Hampshire, England, 1970.
17. *PL/1 Basis/1-11*, ECMA.TC10/ANSI.X3J1, European Computer Manufacturers Association, 1974.
18. P. Wegner, *Programming Languages, Information Structures, and Machine Organization*, McGraw-Hill Book Company, Inc., New York, 1968.
19. D. B. Lomet, "A Cellular Storage Model for Programming Languages," *Research Report RC4360*, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, 1973.
20. *APL/360: User's Manual*, GH20-0683, IBM Corporation, White Plains, New York, 1968.
21. C. H. Lindsey and S. G. van der Meulen, *Informal Introduction to ALGOL 68*, North Holland Publishing Company, Amsterdam 1971.
22. J. B. Morris, *A Comparison of MADCAP and SETL*, Los Alamos Scientific Laboratory, University of California, Los Alamos, Ca., 1973.
23. M. B. Wells, "Aspects of Language Design for Combinatorial Computing," *IEEE Trans. Comput.* **13**, 431 (August 1964).
24. J. Schwartz, "Automatic and Semi-automatic Optimization of SETL," *SIGPLAN Notices* **9**, 43 (April 1974).
25. E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Commun. ACM* **13**, 377 (1970).
26. J. R. Low, *Automatic Coding: Choice of Data Structures*, TR1, University of Rochester, Rochester, New York 1974.
27. Private communication.
28. P. Brinch Hansen, *Operating System Principles*, Prentice Hall, Inc., Englewood Cliffs, New Jersey 1973.
29. B. Liskov and S. Zilles, "Programming with Abstract Data Types," *SIGPLAN Notices* **9**, 50 (April 1974).
30. R. Goldberg and P. H. Oden, "Data Types and Data Type Extensions in Programming Languages," *Research Report RC4651*, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, 1973.
31. J. C. Reynolds, "GEDANKEN – A Simple Typeless Language Based on the Principle of Completeness and the Reference Concept," *Commun. ACM* **23**, 308 (May 1970).
32. F. Gustavson, W. Liniger, and R. Willoughby, "Symbolic Generation of an Optimal Crout Algorithm for Sparse Systems of Linear Equations," *Journal Assoc. Comput. Mach.* **17**, 87 (April 1970).
33. T. E. Cheatam, "The Introduction of Definital Facilities into Higher Level Languages," *AFIPS Conf. Proc., Fall Joint Computer Conference* **29**, 23 (1966).

Received June 26, 1975

The author, who is assigned to the IBM Thomas J. Watson Research Center, Yorktown Heights, N.Y. 10598, is presently on sabbatical at the University of Newcastle upon Tyne.