On the Problem of Uniform References to Data Structures

CHARLES M. GESCHKE AND JAMES G. MITCHELL

Abstract—The cost of a change to a large software system is often primarily a function of the size of the system rather than the complexity of the change. One reason for this is that programs which access some given data structure must operate on it using notations which are determined by its exact representation. Thus, changing how it is implemented may necessitate changes to the programs which access it. This paper develops a programming language notation and semantic interpretations which allow a program to operate on a data object in a manner which is dependent only on its logical or abstract properties and independent of its underlying concrete representation.

Index Terms—Abstract data types, compilation, data description, extensible languages, generic functions, Simula 67, sparse arrays, structured values, systems programming language, uniform references.

INTRODUCTION

IF A programmer needs to represent a table which associates a sum of money with a name for all the checking accounts in some bank branch office, he must first decide what operations are germane to an account table. Adding an amount to someone's account (e.g., increase John Smith's account by $5.03), subtracting an amount from an account (possibly with a check against overdrawing), closing an account (deleting the entry in the table for John Smith), and opening a new account (e.g., making a new entry in the table for Jane Doe with an initial balance of $0.00) are one possible set. Once such a set of *logical* operations on the account table are known, the programmer can make decisions on how to represent one in the computer.

Any specific way of implementing an account table will have to provide storage and storage management for the table itself, as well as *concrete* operations corresponding to the logical operations on an account table. To distinguish between the notion of an account table and the specific way in which it is implemented in the computer, we will refer to the former as an *abstract data structure* (or simply an *abstraction*) and to the latter as a *representation* of it. For instance, representing an account table as two arrays, one containing strings (the names associated with the accounts), and the other containing numbers (the funds in the accounts), would mean that the

Manuscript received February 1, 1975.
The authors are with the Palo Alto Research Center, Xerox Corporation, Palo Alto, Calif. 94304.

concrete counterpart of making a withdrawal involves looking up an account name in the first array and then subtracting the amount from the corresponding entry in the numbers array. One would also have to specify what is to be done if the name cannot be found in the array of names or if there are not sufficient funds in the account to cover the withdrawal, but these considerations are issues about any representation of the abstract structure and not just about this implementation. Other considerations, however, may well be properties of a particular representation and not intrinsic to the abstract structure.

A good illustration of a constraint imposed by the concrete form of an abstraction is provided by the above representation for an account table. Since we need to be able to open new accounts, and since arrays typically have a fixed number of entries, we may be unable to open a new account if the names array is full. We will have more to say about this situation later; for the moment, it points out that we might want to change the representation for an account table because we found drawbacks in the one chosen. Such modifications and the situations which prompt them are a common occurrence in all large software systems.

In fact, a large software system may be viewed as a set of abstract data structures along with control programs which use them to some common end. The data represent the knowledge which is available for action, and the control programs are strategies which are driven by it. Compilers for programming languages, airline reservation systems, control systems for petroleum refining, and operating systems all have this flavor, if one is willing to accept people at computer terminals and chemical reactions as data structures with certain logical operations which can be invoked from the software system.

THE UNIFORM REFERENCE PROBLEM

Since the representations of a number of the abstract data structures in a large system can be expected to change over its lifetime, it would be desirable if programs in the system which access a given data structure could be decoupled as much as possible from the exact, concrete form which its implementor chose to give it. Then, changing the representation would not require altering all the programs which depend on the abstraction.

To be more precise, let us call all the logical operations on an abstract data structure its *attributes*. This term is intended to include those operations which we view as manipulating the data in some nontrivial way (such as withdrawing funds from a checking account), and those which we view as simpler properties (such as the current balance in someone's checking account). And, instead of speaking of a specific abstract data structure, we will talk about abstract (data) *types*, since it is often the case that there are a number of specific examples of the same abstraction serving different purposes in a single system (e.g., a table for a checking account and another for a savings account might be equivalent at the abstract level). Each such use of an abstract type will be called an *instance* of the type or an *object* of that type.

If a program using an object x of type T could always refer to attributes of x using a single, uniform notation, then it would be independent of exactly how the type T is represented. This is the *uniform reference* problem, and its solution is the central issue of this paper.

Currently, there is no reasonable solution for this problem: changing the representation of an abstract type will often precipitate widespread changes to the programs which use it, even though the semantics of the abstraction have not changed. This has (at least) three unfortunate side effects:

1) each unnecessary change to an already reliable program is an opportunity to introduce errors into it, and making global program alterations in order to accommodate a single change in data representation can be devastating to a system's reliability;

2) the high cost of diffusing a data change throughout a program inhibits experimentation with representations, and artificially increases the inertia of the system to good and necessary modifications;

3) the well-advertised advantages of abstraction [5] as an aid to producing good software are often lost as references to the abstract properties of an object are (manually) translated into the various syntactic forms demanded by the particular representation chosen.

The remainder of this paper is divided into five sections. The first (Historical Background) will review past attempts and partial solutions to the problem and give a reasonably explicit statement of the form of our solution for it. The second and third section (Syntax for Defining and Using Data Types, and The Forms of Type Extension) contain an assortment of programming language syntax necessary to the explanation and development of a solution. After that we will develop, by a series of examples, the semantic actions needed to support a solution in a compiler. That set of examples is believed to cover all the interesting cases of the problem. A statement of the general solution for the problem will follow that development.

HISTORICAL BACKGROUND

In this section we will show why past and current programming languages do not support solutions to the uniform reference problem, how Simula 67 [10] yields a partial solution which seems extendible to a complete solution, and what its form is.

The manipulation of a cloud of data structures which characterizes large software systems also exposes the main weakness of languages such as Fortran and Algol 60 when used for systems programming. If one wishes to represent an abstraction such as a stack in Algol, and wishes to separate programs using it from how it is represented, he must implement the logical operations such as "push an item onto the stack," "pop an item from the stack,"

or "is the stack empty?" as a set of procedures. Such procedural extensions are the most time-honored means for realizing abstractions, but they suffer from two main flaws:

1) programs which use procedural abstractions generally lose much of the clarity of simple Algol and begin to acquire the appearance of pure Lisp as functional embedding becomes the major form of control;

2) implementing all the attributes of a type as procedures can be intolerably inefficient of both space and time.

Simula's dot notation provides some assistance with these two problems. In Simula, the expression *x.Attr* is used to apply the attribute *Attr* to the object *x*. If *x* is of type *T* (actually *class T* in Simula), *Attr* may be either a procedure or a simple value declared within the definition of *T*. For example, let the class *Vector* define objects with attributes *x*, *y*, *rho*, and *theta*. This type is one possible way of capturing the semantics of two-dimensional vectors (there are undoubtedly others). The attributes are intended to have the following meanings: *x* is the *x*-coordinate in a rectangular coordinate system with basis vectors [1,0] and [0,1]; *y* is the corresponding *y*-coordinate; *rho* is the length of a *Vector* (= *Sqrt*[$x \uparrow 2 + y \uparrow 2$]); and *theta* is the angle between a *Vector* and the *x*-axis (= *ArcTan*[y/x]). In Simula, these attributes of a *Vector*, *v*, are referred to as *v.x*, *v.y*, *v.rho*, and *v.theta* regardless of whether the attributes are implemented as simple values or as procedures. A bonus from this uniform means of referring to attributes is that the representation of *Vector* objects could be rectangular or polar, without the necessity of changing the programs which refer to them. Of course, the programs might have to be recompiled if the representation of *Vector* were altered, but no alterations to the source text would be required.

Simula, however, imposes two (severe) limitations on this notion. First, attribute expressions may not appear on the left-hand side of assignment operators unless the attributes are simple values. Thus, if a program operating on *Vectors* contained a statement such as "*v.x* ← 5.0," the representation for *Vectors* could not be changed to implement *rho* and *theta* as values and *x* and *y* as procedures without altering this statement, perhaps to something of the form "*Setx*[*v*,5.0]." This, of course, is just procedural extension.

The second limitation is that it is not possible to augment the meanings of the built-in Simula operators such as "+" or "←" to include similar operations on programmer-defined types except by procedural extension. Such a capability has, however, been provided in a number of extensible languages such as ECL [2], PPL [3], and Algol 68 [4].

These two limitations to an otherwise satisfactory notation suggest the form and content of a more general solution to the uniform reference problem:

1) an attribute *Attr* of an object *x* should be accessible in both right- and left-hand contexts using either of the notations *x.Attr* or *Attr*[*x*] (function form);

2) the built-in language operators such as "←," "+," etc., should be extensible to user-defined data types. The second goal is closely connected with the first since a statement such as "*x.Attr* ← *y*" may require extension of "←" in order to allow *Attr* to have meaning in this context.

SYNTAX FOR DEFINING AND USING DATA TYPES

This section defines and explains some syntactic constructs which are useful for dealing with the uniform reference problem. This syntax has been drawn from the language definition of the Mesa programming system, in which these facilities are being implemented. This is not a complete language definition, and only those syntactic entities which are useful for explaining the solution of the problem are included. All syntax equations will be written in standard Backus–Naur form (BNF) except that nonterminal symbols are printed in italics and may contain a hyphen. A declaration of anything in the Mesa programming language (MPL) has the form

declaration ::= *list-of-identifiers* : *type*

A *type* will be more fully defined later, but it includes familiar forms such as INTEGER, REAL, and BOOLEAN: e.g.,

*x*1: REAL;
i,*j*: INTEGER;

One can also specify an initial value to be assigned to a variable being declared by appending "← *expression*" to a declaration. For example,

*x*1: REAL ← 7.5;
i,*j*: INTEGER ← *gcd*[23, 77];

The latter case will initialize both *i* and *j* to the value computed by the function call "*gcd*[23, 77]." Additionally, it is possible to replace the "←" in an initialization by "=," in which case the variable so initialized is not allowed to be changed after declaration (i.e., it cannot appear on the left of an assignment operator). In the special, but frequent case that the expression following "=" is a compile-time constant, the variable so declared and initialized will also be a compile-time constant. For example,

pi: REAL = 3.141592654;

declares pi to be a compile-time constant, and the compiler may take advantage of this constancy to produce more efficient code, or to save storage if either is feasible. This rule for constancy extends naturally to include cases such as

pisquare: REAL = *pi***pi*;

which makes pisquare a compile-time constant because *pi* is.

The built-in types of the language can be extended by declaring variables of type TYPE. Such a variable is

declared just like any other, with the additional constraint that its declaration must include an "=" initialization in which the expression is a compile-time constant. Some examples satisfying these demands are

```
int: TYPE = INTEGER;
intarray: TYPE = ARRAY [1..10] OF int;
```

This works because "INTEGER," and "ARRAY...," being built into the language, are valid examples of compile-time TYPE constants. Consequently, *int* and *intarray* are also TYPE constants, and may be used to declare other variables:

```
i,j: int ← gcd[23, 77];
ia: intarray;
```

This notion that there is no real distinction between TYPE variables and others in the language insofar as their declaration, scopes, etc., are concerned is due primarily to ECL [2].

THE FORMS OF TYPE EXTENSION

Besides built-in TYPE constants such as INTEGER and REAL, the programmer can build new types from others using one of the four following type *forms*.

1) An *array* type is formed by defining an interval for the indices which can be used to select individual components, and a range type defining the type of the components. For example,

```
ThreeD: TYPE = ARRAY [1..3] OF REAL;
```

defines a new type *ThreeD*, and any variable *td* declared to be of that type will consist of three REAL values: *td*[1], *td*[2], and *td*[3]. Arrays are not essential for explaining the uniform reference problem, and they will generally be omitted from further discussion, although the operations and language facilities provided for them are comparable to the other type forms to come.

2) A *record* type forms a new type from a list of others: the result describes objects each of which contains a set of components whose types correspond to those given in the record list. Records, and their creation, manipulation and destruction are central to the language's philosophy. Two-dimensional vectors could be represented as records like

```
Vector: TYPE = RECORD [x: REAL, y: REAL];
```

And, if one declared the variable *vv* as

```
vv: Vector;
```

then the two components of *vv* are accessible as *vv.x* and *vv.y*, respectively, as in Simula.

3) A *procedure* type is "similar" to a record type because all procedures are viewed as having record "values" as their results (the input parameter list to a procedure is also a record value). This apparently rigid constraint on parameters and return values will be explained below. Most procedure type forms are not used to declare *new* types, but are used directly in declaring procedure variables and real, live procedures as in Algol. For example, the declaration

```
hypotenuse: PROCEDURE [x,y: REAL]
  RETURNS [REAL];
```

uses a procedure type form (everything after the first ":") to declare the variable hypotenuse. Alternatively, we could have given two declarations as in previous examples:

```
Metric: TYPE = PROCEDURE [x,y: REAL]
  RETURNS [REAL];
hypotenuse: Metric;
```

This accomplishes much the same thing insofar as hypotenuse is concerned, and also gives a name to its type, *Metric*.

4) A *pointer* type describes objects which contain the addresses of other objects of some specified type. For example

```
VectorPtr: TYPE = POINTER TO Vector;
```

Any variable *vptr* declared to be of type *VectorPtr* will contain the addresses of *Vector* objects as possible values, and may be used to access those objects and their components. For instance, if *vptr* contained the address of *vv* [see 2) above], then *vptr.x* and *vptr.y* would have the same effect as *vv.x* and *vv.y*, respectively.

Hereafter, we will use the term *type* to mean any type form, the name of a type variable, or a type constant. The next sections describe record, procedure, and pointer types in more detail.

DETAILED SYNTAX FOR DECLARING EXTENDED TYPES

When a record type is defined, a list of types is given. Normally, each element of the list has an identifier, called a *selector* name associated with it, but there is also a form in which no selectors are supplied (it is primarily used in defining return records for procedures). The exact syntax for a record form is

```
record ::= RECORD [record-list]
record-list ::= declaration-list | type-list
declaration-list ::= declaration | declaration-list,
  declaration
type-list ::= type | type-list, type
```

If a component declaration includes an initialization part, it applies when any object of the record type comes into existence. If the initialization is a compile-time constant, then all instances of that type will "possess" the same value for that component—the compiler may, therefore, take advantage of this constancy just as for simple declarations. For instance, in the declaration

```
Polar: TYPE = RECORD [
  Angle: TYPE = REAL,
  ninety: Angle = pi/2,
  rho: REAL
  theta: Angle ];
```

the internal declaration of the type *Angle* can be used at compile time to declare the type of the component theta, and the component ninety need not even be allocated space in *Polar* objects since it is a compile-time constant. Also, any program using the *Polar* declaration may also state declarations such as

v: *Polar*;
mytheta: *Polar.Angle*;

thus making assignments such as

mytheta ← *v.theta*; *mytheta* ← *v.ninety*;

correct as to type. The use of the expression

"Polar. Angle"

(whose type is TYPE) in defining mytheta shows that a full definition of what a type form can be must also include general expressions.

As mentioned previously, a procedure is viewed as taking a record object as its argument and returning a record object as its result. The syntax for declaring a procedure type is

procedure ::= PROCEDURE *procedure-domain*
procedure-range
procedure-domain ::= [*record-list*] | *empty*
procedure-range ::= RETURNS [*record-list*]
| *empty*

Examples are

sqrt: PROCEDURE [*x*: REAL] RETURNS [REAL];
Create: PROCEDURE [*r,i*: REAL ← 0]
RETURNS [*c*: *Complex*];

A procedure object may be "assigned" a *procedure-body* as a value, and this supplies executable code for it. The form of such a body is

procedure-body ::= BEGIN *statement-list* END

Procedure variables which are assigned a body using an "=" initialization are the analogues of normal Algol or Pascal [8] procedures; for instance,

Create: PROCEDURE [*r,i*: REAL ← 0]
RETURNS [*c*: *Complex*] =
BEGIN
c ← [*r,i*]; —construct a
Complex and assign it to *c*
END

The value "[*r,i*]" assigned to *c* is an example of a structured value and is called a *constructor*. For each type form there is an associated constructor form for writing values of those types. In fact, the syntactic form *procedure-body* is the constructor form for procedure types. There are also constructor forms for arrays, records, and pointers.

CONSTRUCTORS

A record constructor is a list of values to be associated with the various components of an object of that type. Each value may be preceded by *"selector-name:"* in which case it will be associated with the component having that name. If no value is specified for some component, and there is an initialization expression associated with it in the record definition, then it will be assigned that value. For example, suppose that *r*1 is defined as

*r*1: RECORD [*f*1: REAL ← 0, *f*2: BOOLEAN,
*f*3: INTEGER];

Then the following assignments of various constructors to *r*1 are equivalent:

*r*1 ← [*f*2: TRUE, *f*3: 10];
*r*1 ← [*f*3: 10, *f*1: 0, *f*2: TRUE];

The result of each of these assignments is that *r*1.*f*1 = 0, *r*1.*f*2 = TRUE, and *r*1.*f*3 = 10. Notice that the order of the component values is not important because each of them is identified by selector name. Also, in the first assignment, the value of the *f*1 component is supplied by the default initialization in the record form.

It is often convenient to omit the selector names and simply write a list of values in a constructor. In this case, it is the position of a value which associates it with a component of the record (the order is determined by that given in the record definition). Using a positional constructor, the above effect could also be obtained by writing

*r*1 ← [0, TRUE, 10];
r1 ← [, TRUE, 10];

Here too, omitted values must correspond to components having an initialization expression in the definition.

These two forms can be intermixed, with some component/value associations being determined by name and others by position. The rules governing this are the following:

1) a *named* value is associated with the component having that selector name;
2) if the leftmost value in the constructor is positional, it is assigned to the leftmost component in the record;
3) a *positional* value is assigned to the component in the *record-list* which follows the one to which the preceding value (whether positional or named) was assigned;
4) components which are declared with "=" initialization are excluded from positional counts, and cannot be assigned values in a constructor.

The following "mixed" constructors perform exactly as the two sets of previous examples:

*r*1 ← [*f*2: TRUE, 10, *f*1: 0];
*r*1 ← [0, *f*3: 10, *f*2: TRUE];

The above rules for mixed constructors are adopted from the TSS/360 Command Language [7]. Of course, other rules are possible and have been used [6].

The dual of constructing a record object from separate values is disassembling a record into its component

values. The mechanism for assigning parts of a record value to a set of variables is called an *extractor*. Its form is syntactically similar to a constructor, but it is used only on the left-hand side of assignment operators. For example, if the variables r, b, and i are declared as

r: REAL; b: BOOLEAN; i: INTEGER;

then the following examples of extractors closely parallel the constructors above:

$[r, b, i] \leftarrow r1;$	—a positional extractor
$[f3: i, f1: r, f2: b] \leftarrow r1;$	—extraction by names only
$[r, f3: i, f2: b] \leftarrow r1;$	—mixed name/positional extraction

These all result in the assignments

$r \leftarrow r1.f1; b \leftarrow r1.f2; i \leftarrow r1.f3;$

Calling procedures and receiving results from them make heavy use of constructors and extractors because procedures accept and return only records. This has the following effects:

1) a parameter list to a procedure is a record constructor, and all the capabilities for specifying or defaulting values in constructors hold for parameters to procedures;

2) a procedure result, being a record value, may be disassembled using an extractor;

3) anything which can be transmitted to a procedure as an argument can also be received as a result, since both are record types.

The following procedure declarations and calls on them take advantage of these capabilities:

```
Create: PROCEDURE [r,i: REAL ← 0]
  RETURNS [c: Complex];
czero: Complex ← Create [];
                    —defaults r and i to 0.
imag: Complex ← Create [i:1];
                    —defaults only r to 0.
Root: PROCEDURE [r: REAL, n: INTEGER ← 2]
                RETURNS [REAL];
x: REAL ← Root [5];
                    —defaulting n gives square root
```

As we shall see, named values in constructors and the constructor/parameter-list identity are essential to our solution of the uniform reference problem.

Constructors for pointer values provide a good foil for the syntactic wizardry for records: a pointer constructor consists only of the symbol "@" (to be read as "address of") followed by a *referable-value*. And the only referable-values are simple identifiers, array selections (e.g., $td[i]$) or record component selections (e.g., $vv.x$). As further constraints on pointer values, one is not allowed to combine any pointer value with operators other than assignment and the various selection mechanisms (e.g., the "." notation). Thus, a statement such as

"$p \leftarrow (@x) + y;$"

is simply not allowed.

The main advantages and power of having pointer objects in the language are the ability to alter objects *in situ*, and the ability to build structures which are bound together by pointer links. They also have many disadvantages, including the problems of the relative lifetimes of pointers and the objects to which they refer, whether or not to build a system which assumes a garbage collection scheme, or whether using a pointer variable provides automatic access to the object to which it refers. This last difficulty is a special case of the critical role of *coercions* in the language, and we shall expand the problem first for pointers and then for procedures and records.

COERCIONS

In Simula, pointers are always automatically *dereferenced* (followed) on the left of ":=" and there is a special form of assignment ":—" for bypassing this automatic feature. If it did not exist, there would be no way to assign one pointer value to another. In Mesa, a pointer variable in a left-hand context is *never* automatically followed to obtain a value. If p is declared as

p: POINTER TO t;

then the statement "$p \leftarrow q$" has the following possible meanings, depending solely on the type of q, the right-hand side:

1) if q is declared as

q: POINTER TO t;

then the value of q is simply copied into p, and they will then both point to the same object;

2) if q is a *referable-value* of type t (as described previously), then a pointer to it is stored in p. In particular, if q is a simple variable, "$p \leftarrow q$" is equivalent to "$p \leftarrow @q$."

These interpretations are based on two principles:

1) never automatically coerce the left-hand side of an assignment: this simplifies the analysis of assignment and avoids the surprises to programmers (and language designers!) which can accompany automatic dereferencing of pointers;

2) associating actual parameters with formals in procedure calls should have the same semantics as normal assignment.

If, in the latter case, a procedure declares a parameter p as

p: POINTER TO t;

then the only reasonable meaning to attach to passing an actual parameter q of type t is that it be passed "by reference": i.e., the value of p should be the address of q. Hence, "$p \leftarrow q$" means "$p \leftarrow @q$" in this particular case—consequently, it has this same meaning in all normal assignments.

The restricted form of a *referable-value* has an interesting side effect for procedure parameters: one cannot, in general, write an arbitrary expression as an argument to

a procedure which expects a pointer. Such a restriction may at first appear somewhat overconstraining. However, it performs the valuable function of affirming that transmitting a pointer to an object gives the called procedure indirect access to it: if it does not need pointer access to a parameter, it should be passed by value. Such a rule would prevent the infamous Fortran problem which occurs when a subroutine alters a constant (such as "1") passed to it by reference, thereby creating an obscure sequence of events later in the calling program when references to the "constant" yield some other value.

The language also has a limited number of coercions which are applicable in right-hand contexts only. In the following descriptions, the notation TYPE [x] means "the type of x," and "$t1 \Rightarrow t2$" means "type $t1$ is coercible to type $t2$".

$c1$) If TYPE [x] = RECORD [t] (i.e., a single-component record), then TYPE [x] $\Rightarrow t$ by "unbundling" it. A procedure defined as

```
p: PROCEDURE...RETURNS [r: REAL];
```

may thus be used as if it returned a simple REAL.

$c2$) If TYPE [x] = POINTER TO t, then TYPE [x] $\Rightarrow t$ by following the pointer value. Thus, if y is declared as a t, "$y \leftarrow x$" means store the value to which x points into y.

$c3$) If TYPE [x] = PROCEDURE...RETURNS [t], then TYPE [x] $\Rightarrow$ RECORD [t] by calling x.

$c4$) Any combinations and levels of $c1$), $c2$), and $c3$) are allowed. For example, if

```
x: REAL;
f: PROCEDURE RETURNS [POINTER TO REAL];
```

then "$x \leftarrow f$" means "call f, unbundle the single-component return record, and assign the value to which it refers to x."

Another way of looking at these rules is that they describe a strategy for coercing types which corresponds to a "minimum energy" solution: if two types on the left and right of an assignment match, then absolutely no coercion takes place; if they do not, the right will be evaluated enough to make them the same; if no such sequence of evaluations on the right side is possible, the assignment is invalid.

Of course, sometimes one wants the right or the left to be evaluated even though these rules would not automatically cause it. In this case the programmer may write the operator "EVAL" preceding any expression whose evaluation is required. The compiler's automatic rules then apply to the assignment with the EVALed type of that expression replacing its normal type. The following contrived example shows this:

```
PtrToXproc: TYPE = POINTER TO Xproc;
Xproc: TYPE = PROCEDURE
  RETURNS [PtrToXproc];
p: PtrToXproc;
anXprox: Xproc;
```

Normally, the assignment "$p \leftarrow anXproc$" would mean "$p \leftarrow @anXproc$," according to the foregoing rules. However, if one wanted to call the procedure $anXprox$ and store the pointer value which it produces in p, it can be done by writing either

```
p ← EVAL anXproc;
```

or

```
p ← anXprox[ ];          —empty argument list
```

The type of the evaluated right-hand side in each case is *PtrToXproc* and will therefore be stored directly into p without further evaluation.

If the programmer knows what type he wants an expression to be evaluated to, he can append "*.type*" to it. This *casts* the expression in that type—provided, of course, that a reasonable sequence of evaluations can produce a result of the desired type. This is similar to the coercion of the same name in Algol 68 [4]. For example, if $v1$, $v2$, and pv are defined as

```
v1, v2: Vector;
pv: POINTER TO Vector ← @v1;
```

and one wants to assign $v2$ to whatever value pv currently points, then writing

```
pv.Vector ← v2;
```

accomplishes this by casting the left-hand side as a *Vector* (instead of a pointer to one). This notation seems to naturally extend the use of pointers for selecting record components to allow them to select entire records.

A SOLUTION FOR THE UNIFORM REFERENCE PROBLEM

The preceding language capabilities and a set of rules for assigning meanings to attribute references are sufficient to generate a solution for the uniform reference problem. We will develop the necessary rules by examining a set of examples which are believed to "cover" all the special cases of the problem. The basic problem is to make the notations $x.Attr$ and $Attr[x]$ equivalent in both right- and left-hand contexts. *Attr* may be a simple selector name, a procedure name, or an array name (the array case is very similar to the procedure case and will generally be given short shrift).

If *Attr* is a selector name in a record type, it is local (in scope) to that definition, and more than one record type may contain the same selector name. This is very handy for common attributes such as LENGTH, etc., which are often properties of different types of objects. If *Attr* is to be allowed to be a *procedural* attribute instead of a simple value, then procedures must have this same locality with respect to their parameters' types as selector names do.

Identically named procedures which are differentiated by the types of their arguments are called *generic* [9]: their common name implies a function which has meaning

over a number of different types. In the Mesa system, *all* procedures are potentially generic (i.e., any with the same name are distinguished by their parameter and return record types). This yields an immediate solution to a large number of the common cases of the uniform reference problem (viz., attribute references in right-hand contexts) and that solution has the same capabilities as Simula's, although built from quite a different semantic base.

Procedural attributes in Simula gain locality of naming by being declared within a class, just like value components. This is a solution which is also usable in Mesa. For example, if we were to implement the type *Vector* mentioned in the introduction, we might construct a definition such as the following (recall that its attributes are to be *x*, *y*, *rho*, and *theta* and that it is to represent two-dimensional vectors):

```
Vector: TYPE = RECORD [
  x,y: REAL,
  rho: PROCEDURE RETURNS [REAL] =
    BEGIN
    RETURN [SQRT[x ↑ 2 + y ↑ 2]]
    END,
  theta: PROCEDURE RETURNS [REAL] =
    BEGIN
    RETURN [ARCTAN [y/x]] END
  ];
```

The variables *x* and *y* which are referred to in rho and theta are, of course the *x* and *y* components of the record definition. Thus, the compiled code for evaluating an expression such as *v.rho* must provide a pointer to *v* as part of *rho's* execution environment. There is another way to define *rho* and *theta* which points out how generic procedures can provide procedural attributes which work in left-hand contexts. We will give another, equally valid definition of *Vector* which defines *rho* and *theta* generically and obtains the same effect as the above:

```
Vector: TYPE = RECORD [
  x,y: REAL,
  Handle: TYPE = POINTER TO Vector];
rho: PROCEDURE [v: Vector.Handle]
  RETURNS [REAL] =
    BEGIN
    RETURN [SQRT[(v.x) ↑ 2 + (v.y) ↑ 2]];
    END;
theta: PROCEDURE [v: Vector.Handle]
      RETURNS [REAL] =
    BEGIN
    RETURN [ARCTAN [v.y/v.x]];
    END;
```

In these definitions, the *Vector* to be operated on is passed explicitly (by reference since a *Vector.Handle* is a pointer to a *Vector*), and there could be many procedures named *rho*: the right one to use in a phrase such as *v.rho* (=*rho* [*v*]) is determined by *v*'s type. More importantly, however, we can use two other generic procedures, also called *rho* and *theta*, to provide left-hand meanings for the two attributes.

The issue of what interpretation to place on *v.rho* in left-hand contexts reduces to what meaning to assign to rho in a statement such as "*v.rho* ← *r*." We will interpret it as "*rho* [*v*,*r*]" if there is a (generic) procedure named *rho* which takes a *Vector.Handle* and a REAL as parameters (assuming the type of *r* is REAL). Such a definition might be

```
rho: PROCEDURE [v: Vector.Handle, r: REAL] =
  BEGIN
  v.Vector ← [x: r*COS[v.theta],
              y: r*SIN[v.theta]];
  END;
```

This definition allows us to alter the *rho* "value" of a *Vector* by mapping that change into suitable effects on the underlying [*x*,*y*] representation. It also demonstrates the first of the nonobvious interpretations which Mesa is prepared to apply to an attribute reference in support of uniform referencing.

There are two minor but interesting points about the body of the procedure itself:

1) it accesses *v.theta* as if *theta* were a simple value (it is not; look at the procedure definitions above);

2) although the expression *SIN* [*v.theta*] could have been written *v.theta.SIN*, it is sometimes nice to think of functions as functions and not as attributes of the object to which they are being applied.

Generic procedures also provide a natural way of extending built-in operators such as "+" to new types of operands. In Mesa, the meaning assigned to the expression "*a* + *b*" is "PLUS [*a*,*b*]," and if there is a suitably defined generic function PLUS, the meaning of the expression is that it be called and the value which it returns (whose type is determinable at compile time) will be the value of the expression. If no user-supplied PLUS is found, the system-defined meaning (which, of course, is defined in machine code terms) will be encountered at the outermost scope of the program. All the built-in operators including assignment, creating a record value using a constructor, etc., can be extended in this way.

Operator extension by generic procedures provides yet another programmer-augmentable interpretation for attribute references: normally, a simple variable *v* appearing in a right-hand context may be given the meaning SELECT [*v*] assuming that there is a suitable procedure SELECT which accepts *v*-like objects as its parameter. Similarly, the statement "*a* ← *b*" normally means ASSIGN [*a*,*b*]. Using this interpretation, one could define a stack type for which assignment means "push," and for which its appearance in a right-hand context can be interpreted to mean "pop and return the top value on the stack."

We will motivate a slightly generalized form of this

interpretation by another example. A sparse array is an object which is indexed by integer values and which maintains a collection of values for all those components of the array whose values are different from some distinguished value, c (c is often 0). If sa is a sparse array, and $sa[i] = c$, there is no storage allocated for $sa[i]$. Setting $sa[i]$ to c does nothing if $sa[i] = c$ already; if $sa[i]$ is not equal to c, then the storage for $sa[i]$ is reclaimed. Finally, accessing $sa[i]$ yields c if no component for $sa[i]$ is allocated. The hard part of this exercise centers around the meaning of the statement "$sa[i] \leftarrow x$," since the value of the right-hand side must be known in order to perform the correct action. The following type and procedure definitions accomplish this:

```
SparseArray: TYPE =
  RECORD [
    head: POINTER TO Node ← NIL,
    n: INTEGER ← 0,   —count of number
                          of things in the "array"
    Node: TYPE =
      RECORD [
        value: INTEGER,
        index: INTEGER,
        next: POINTER TO Node
      ],
    Handle: TYPE = POINTER TO
                          SparseArray
  ];
c: INTEGER = 0;   —the value for all
                      unallocated components
SELECT: PROCEDURE [i: INTEGER,
          sa: SparseArray.Handle]
    RETURNS [INTEGER] =
  BEGIN
  p: POINTER TO SparseArray.Node ←
          FindNode[sa.head, i];
  RETURN [IF p = NIL THEN c ELSE p.value];
  END;
ASSIGN: PROCEDURE [
    i: INTEGER,
    sa: Sparse Array.Handle.
    rhs: INTEGER] =
  BEGIN
  p: POINTER TO SparseArray.Node ←
          FindNode[sa.head, i];
  IF rhs = c
    THEN
      BEGIN
      IF p = NIL THEN ReleaseNode[p]
      END
    ELSE
      BEGIN
      IF p = NIL
        THEN p ← NewNode[sa,i];
      p.value ← v;
      END
  END;
```

The procedures *FindNode*, *ReleaseNode*, and *NewNode* perform the obvious tasks. The procedure SELECT will be called whenever an expression of the form $s[i]$ appears in a right-hand context. That is, one interpretation of the normal form $i.s$ in a right-hand context is SELECT$[i,s]$, for which the above example has provided a meaning. Similarly, the function ASSIGN$[i,s,v]$ is an allowable interpretation of the normal form assignment "$i.s \leftarrow v$" (or of "$s[i] \leftarrow v$").

Could the notion of a sparse vector have been implemented any other way? The explanation of left-hand meanings for *rho* and *theta* in the *Vector* example suggests that the answer is yes; if, instead of a general type *SparseArray* we had only needed to represent a single sparse array *sa* we could have done so by declaring a pair of generic procedures:

```
sa: PROCEDURE [i: INTEGER]
    RETURNS [REAL] =
  BEGIN
    —code for the right-hand meaning
  END;
sa: PROCEDURE [i: INTEGER, rhs: REAL]
    RETURNS [rhscopy: REAL] =
  BEGIN
    —code for the left-hand meaning
  END;
```

Using these procedures, the meaning of $sa[i]$ (right-hand context) is a call on the first procedure, and the meaning of $sa[i] \leftarrow r$ is $sa[i,r]$, a call on the second procedure. Of course, we would also have needed a set of suitable list management procedures similar to those for the type *SparseArray*. In this example it is worth noticing that the object named *sa* has no actual existence, but is constantly fabricated by procedures.

GENERALIZATIONS AND LIMITATIONS OF THE SOLUTION

References to attributes of objects are often more general than the simple ones illustrated above. This is a natural consequence of structuring data so that the components of an object are themselves composed of a set of attributes. If, for example, one had an array of records, each containing a string and a number (a possible implementation of the account table abstraction discussed in the introduction) defined as

```
AccountTable: TYPE =
  ARRAY [1..1000] OF
  RECORD [name: STRING, balance: REAL];
checking: AccountTable;
```

then a complex selection expression such as

"checking [i].name.LENGTH"

could easily occur. It would be nice if all such expressions could be placed into some standard form which involved only dots or brackets, but not both. Then our analysis of the meaning of the expression could be simpler.

We define the form "$x.a_n.a_{n-1}\cdots a_1$," where x is an expression and $a_1,\cdots,a_n$ are attribute references, as the *normal form* for cascading attribute accesses. An attribute reference string involving both dot notation and brackets can be unambiguously put into this normal form. There is no normal form which uses only brackets and not dot notation, so this is not a reversible process. The rules for doing this are the following:

1) the normal form for $a[i]$ is $i.a$;

2) the normal form for $a[x1,x2,\cdots,xn]$ is $[x1,x2,\cdots,xn].a$, where the $[x1,\cdots,xn]$ is a record value;

3) any mixed-attribute reference can be normalized in a piece-wise fashion: from left to right, once one knows that "[" binds more tightly than ".". For example: $a.r[i]$ becomes $a.(i.r)$, and $a[i].b[m].ic$ becomes $(i.a).(m.b).ic$.

The examples in the previous section and the interpretations developed to provide specific solutions to the problem form a basis from which a general solution can be developed. Of course, there is no proof that this is so. As with many issues in programming languages, we are only able to generate examples to test this base for completeness. So far, all such examples have fallen into one or more of the above interpretation categories. We will give here a final example which we believe is sufficiently "baroque" (although not implausible) to demonstrate this as well as show the direction in which we have generalized the simple interpretations.

If we extend the previous sparse array example to define a sparse array of record objects, it can still be represented using our schemes. We define the following record type as the type of the components of the sparse array:

```
RecordType: TYPE = RECORD [A1: STRING,
  A2: REAL];
```

As before, our strategy will be to define suitable generic procedures, each named *sa*, to handle the forms "$sa[i]$" (right-hand contexts) and "$sa[i] \leftarrow rhs$." In addition, since each component of *sa* is a record object, we need to supply procedures to handle "$sa[i].A1 \leftarrow rhs$" and "$sa[i].A2 \leftarrow rhs$."

```
sa: PROCEDURE [i: INTEGER]
      RETURNS [RecordType] =
  BEGIN
    —code for the right-hand
      meaning for sa[i]
  END;
sa: PROCEDURE [i: INTEGER, rhs: RecordType] =
  BEGIN
    —Code for the left-hand
      meaning for sa[i]
  END;
A1name: TYPE = IN {A1};  —like Pascal SET
                           (see below)
A2name: TYPE = IN {A2};
sa: PROCEDURE [i: INTEGER, comp: A1name,
          rhs: STRING] =
  BEGIN          —procedure which handles
                   sa[i].A1 ← rhs
  sa[i] ← [A1: rhs, A2: sa[i].A2];
      — the sa to the left of "←" is the second
        procedure above; the sa[i] appearing
        after "A2:" is a call on the first
        procedure above
  END;
sa: PROCEDURE [i: INTEGER, comp: A2name,
          rhs: REAL] =
  BEGIN          —procedure which handles
                   sa[i].A2 ← rhs
  sa[i] ← [A1: sa[i].A1, A2: rhs];
  END;
```

A solution containing four generic procedures seem somewhat complex, but the data structure being emulate is not a simple one. The main reason that this set procedures is an acceptable representation for an arra of records is that a statement such as

$sa[i].A1 \leftarrow rhs;$

(which, in normal form is "$i.sa.A1 \leftarrow rhs$") can b interpreted to mean

$sa[i,A1,rhs];$

if a suitable procedure *sa* exists. There is a progra "pun" used to achieve this effect in the above exampl The second argument to the last two procedures is special type (either *A1name* or *A2name*). Such a typ corresponds to the Pascal notion called a set, and variabl declared of such a type may assume one of a set of value The names of those values are the identifiers inside the in the type definition. Thus, in *A1name* we have introduce a set type for which only a single value, indicated by th identifier $A1$, is valid. The pun is that it is identical to th first selector name in the definition of *RecordType*. Thu when the programmer writes an assignment such as th one above, $A1$ will be taken to be a value of the typ *A1name*, and not a selector in *RecordType*. This tric enables the compiler to pick the appropriate procedu which can handle the type of the right-hand side (eith a STRING or a REAL).

One might think that a single procedure could be writte to replace the last two above. It would contain a CAS statement for deciding which field ($A1$ or $A2$) to alte such as the following:

```
CASE comp OF
  A1: sa[i] ← [rhs, sa[i].A2];
  A2: sa[i] ← [sa[i].A1, rhs];
END;
```

The only problem with this solution is that the type the parameter *rhs* must be able to be either STRING REAL, and run-time type checking would be needed in th CASE statement. The two-procedure solution avoids bo of these issues.

The real point of the example is that one interpretatio of the general form

$x.Attr1\cdots Attrn \leftarrow y;$

is

*Attr*1[*x*,*Attr*2,···,*Attrn*,*y*];

as we will show in the next section. It is also possible to develop a more general representation which parallels the *SparseArray* type developed earlier. That case generalizes in a fashion similar to the "*sa*" version: SELECT and ASSIGN will receive the extra attribute names as parameters which they may use to decide which action is required.

The language syntax and semantic rules which we have presented appear to satisfactorily solve at least the obvious cases of the uniform reference problem. No guarantee is made that all Mesa programs will be free of the problem, but it should be possible for someone defining a new abstract type to "enforce" the virtual access to its attributes and thereby free programs using it from representation dependencies. Specifically, this means that when a data type is redefined and the programs which use objects of that type are recompiled, one of the two following conditions must hold:

1) the recompiled programs now access the altered attributes and require no change; or

2) some construction in a program is not suitable for the altered attributes, will be flagged as an error, and must be changed.

The latter possibility correctly suggests that our solution to the problem is not foolproof. It also illustrates one reason for naming the values in a constructor.

If we were using the previous definition for a *Vector* (with x and y value components), it would seem entirely reasonable to write a statement such as

ybasis: *Vector* ← [0,1];

which, besides declaring *ybasis*, is also intended to have the effect

ybasis.x ← 0; *ybasis*.y ← 1;

However, if we decided to alter the type *Vector* so that *rho* and *theta* were the simple value components, and x and y were implemented as procedural attributes, the above assignment statement, although still valid, would mean

ybasis.*rho* ← 0; *ybasis*. *theta* ← 1;

which is *not* what was originally intended (it is interesting to note that simple procedural extension of data types would also have this problem).

One way out of the difficulty is to permit only named values in constructors and eliminate positional notation altogether. Then the original assignment would have been

ybasis ← [x:0, y:1];

and, even after the definition of Vector has been altered, would still mean

ybasis.x ← 0; *ybasis*.y ← 1;

which would correctly invoke the new procedural meanings for x and y.

Another way of avoiding the problem without discarding the venerable positional notation requires that an expression in a constructor which is to be assigned to an *extended* type either be cast as that type or be a named value. In that case, the implementer for *Vector* could have described its form as

Vector: TYPE = RECORD [
  Length, *Angle*: TYPE = REAL,
  rho: *Length*,
  theta: *Angle*];

In this case, the statement "*ybasis* ← [0,1];" would cause an error when compiled because "0" and "1" are neither named nor cast as a *Length* or an *Angle*. The user could then alter the constructor to something like "[x:0, y:1]" as above or to "[*rho*:1, *theta*:*pi*/2]" or to "[1.*Length*, (*pi*/2).*Angle*]" (casting) to avoid future change.

We are now in a position to state the complete set of rules for interpreting normal forms in both right- and left-hand contexts in the general case.

GENERAL RULES OF INTERPRETATION

The general forms under consideration are

(R_n) $x.a_n.a_{n-1}\cdots a_1$ (right-hand contexts only)

(L_n) $x.a_n.a_{n-1}\cdots a_1 \leftarrow y$.

In general, the following rules can assign zero, one or more meanings to a single expression in a program. If zero, the expression is meaningless (probably because some identifier was not declared), if more than one, the form is ambiguous and is in error. If exactly one meaning is found, the form is *semantically valid* (whether or not it is pragmatically correct and performs the actions the programmer intended is a separate and much more difficult question!).

(R_0) x

The case of a simple variable used in a right-hand context has been described previously in the section Coercions. The form "x" always means "SELECT [x]"; if there is a procedure

SELECT: PROCEDURE [TYPE [x]] RETURNS [t],

then this will be used as the meaning of x and overrides any of the default coercion/evaluation rules. One could use this interpretation to define a type "stack" which would pop the value from a stack variable s whenever it appeared in a right-hand context.

(L_0) $x \leftarrow y$

The possible interpretations of this form are

Case 1): ASSIGN [x,y]

Case 2): x[y].

The built in meanings are associated with Case 1) and are replaceable by suitable user-defined functions named "ASSIGN." To define the stack type mentioned in R_0, one would also write a function

ASSIGN: PROCEDURE [s: *Stack.Handle*, v: *Value*]

to push *Values* onto the stack object for which s is a *Handle*. Case 2) is a trivial version of the previous sparse array examples.

(R_1) $x.a_1$

The possible interpretations of this form are
1) If there is a procedure

SELECT: PROCEDURE [TYPE [x], TYPE [a_1]]
RETURNS [t],

then SELECT [x,a_1] is a possible interpretation.
2) If there is a procedure

a_1: PROCEDURE [TYPE [x]] RETURNS [t],

this is possibly a procedure call, and TYPE [$x.a_1$] = t. This form was used in defining the right-hand value for the procedural attribute *rho* in the *Vector* example.
3) If x is a record type r, and a_1 is one of the selectors for r, then this may be a simple component selection, and TYPE [$x.a_1$] is the type of that component.
4) If there is an array

a_1: ARRAY [$m..n$] OF t,

then this may be an array selection, with TYPE [$a_1[x]$] = t.

(L_1) $x.a_1 \leftarrow y$

1) If there is a procedure

ASSIGN: PROCEDURE [TYPE [x],
TYPE [a_1], TYPE [y]],

then ASSIGN [x,a_1,y] is a possible interpretation.
2) If there is a procedure

a_1: PROCEDURE [TYPE [x], TYPE [y]],

then $a_1[x,y]$ is a possible interpretation. This form was used to define the left-hand procedural attribute *rho* in the *Vector* example.
3) If TYPE [x] $\Rightarrow r$ (a record type), a_1 is a selector name in r, and TYPE [y] $\Rightarrow$ TYPE [$x.a_1$], then a possible interpretation is an assignment to a record component.
4) If there is an array

a_1: ARRAY [$m..n$] OF t,

and TYPE [y] $\Rightarrow t$, then this may be an assignment to an array component.

The general cases R_n and L_n are generalizations of R_0, R_1, L_0, and L_1 above. A large number of extra cases are introduced by allowing the normal form $x.a_n \cdots a_1$ to associate to the left: for instance, a reference string $x.a.b.c$ could be taken to mean $(x.a).b.c$ or $(x.a.b).c$ if the parenthesized portions had meanings which coupled with the remainder of the string. The expression

"*checking* [*i*].*name*.LENGTH"

given previously has this property.

(R_n) $x.a_n.a_{n-1} \cdots a_1$

1) If there is a procedure

SELECT: PROCEDURE [TYPE [x], TYPE [a_n],
$\cdots$,TYPE [a_1]]
RETURNS [t],

then SELECT [x,a_n,a_{n-1},$\cdots$,a_1] is a possible interpretation.
2) If there is a procedure

a_n: PROCEDURE [TYPE [x], TYPE [a_{n-1}],
$\cdots$, TYPE [a_1]],

then $a_n[x,a_{n-1},\cdots,a_1]$ is a possible interpretation. (Left-associativity) for all k IN [0..n), let

$w = x.a_n \cdots a_{n-k}$

be meaningful using interpretation R_k ($k < n$); then

$w.a_{n-k-1} \cdots a_1 \quad (R_{n-k})$ ——

is a possible interpretation of R_n.

(L_n) $x.a_n.a_{n-1} \cdots a_1 \leftarrow y$

1) If there is a procedure

ASSIGN: PROCEDURE [TYPE [x], TYPE [a_n],
$\cdots$, TYPE [a_1], TYPE [y]],

then ASSIGN [x,a_n,$\cdots$,a_1,y] is a possible interpretation of L_n. This possibility is the most general interpretation for the form since it simply passes all the relevant information to the procedure ASSIGN.
2) If there is a procedure

a_n: PROCEDURE [TYPE [x], TYPE [a_{n-1}],
$\cdots$, TYPE [a_1], TYPE [y]],

then $a_n[x,a_{n-1},\cdots,a_1,y]$ is a possible interpretation. (Left-associativity) for all k IN [0..n), let

$w = x.a_n \cdots a_{n-k}$

be meaningful as an R_k interpretation ($k < m$); then,

$w.a_{n-k-1} \cdots a_1 \leftarrow y \quad (L_{n-k})$

is a possible interpretation of L_n.

Although these rules define how meanings are attached to normal forms, that does not mean that there is an implementation of suitable efficiency for a compiler. There is, however, a conceptually easy method: generate all the interpretations of a normal form and then attempt to match each with the possible types for the identifiers in it. If only one such matching is correct, we are done; if none match, or if there is more than one, it is an error. Unhappily, the number of potential interpretations of a normal form is too large to make this reasonable (because of all the subcases introduced by left association). However, a slight modification of the naive approach appears to have much better properties. By keeping all the possible semantics of a given identifier in the compiler's symbol table on a single thread (i.e., associated in some easily accessible way), one can drive interpretation generation in a bottom-up rather than a top-down fashion, discarding an interpretation as soon as a mismatch occurs. The bulk

of the possible interpretations will normally not even be attempted because of the absence of suitable identifier meanings; others will be discarded as soon as they fail to match. Lastly, because just one interpretation fits in the "normal" case, the amount of work done is close to the amount for type checking in any strongly typed language.

CONCLUSIONS

We have attempted to knead the myriad notations for accessing and operating on data in programming languages into a common form. The gains from doing this are twofold:

1) changing the way in which an abstract data type is implemented can be localized to the portion of the program where it is implemented;

2) the user of a data type can concentrate on its logical properties since he cannot take advantage of particular representational characteristics which it might have.

The solution to the uniform reference problem which we developed in this paper depends strongly on two features of programming languages. The first, data type extension, has been developed over the past decade and required little augmentation to support our solution. The second, allowing multiple meanings for an expression depending on the availability of named objects of appropriate types, has been a feature of only a few languages used in artificial intelligence (cf. Bobrow and Raphael [13]). Our human linguistic abilities for context-sensitive communications will probably move future languages even further from the use of syntactic precision as the means of ensuring semantic precision.

Most of our concern with uniform references as well as the initial direction of the solution were inspired by Simula 67 [10], AED [14], and LIS [15]. Simula has also pointed out the way in a number of other areas, most notably with respect to the idea of *classes*, and with the idea that certain manipulations for a data type are the purview of the program which implements it and are too important to allow random programs to operate on it in that way. This is a form of the pervasive protection problem. We have purposely ignored both these issues in this report, but others (cf. Liskov and Zilles [12] and Wulf [11]) are working to generalize the class notion and to employ protection domains to constrain the access to data objects from different parts of a software system. Both of these areas could have strong effects on the reliability of future software, our methodologies for producing it, and our ability to reuse already developed program components instead of rewriting them in almost every new system.

ACKNOWLEDGMENT

The research reported here is the result of contributions from a number of individuals, including C. Dornbush, C. Irby, B. Lampson, E. Satterthwaite, and B. Wegbreit.

REFERENCES

[1] D. B. Wortman, Ed., in *Proc. Workshop Attainment of Reliable Software*, Univ. Toronto, Toronto, Ont., Canada, June 1974.
[2] B. Wegbreit, "The treatment of data types in EL1," *Commun. Ass. Comput. Mach.*, vol. 17, pp. 251–264, May 1974.
[3] E. A. Taft and T. A. Standish, *PPL Users Manual*, Center for Res. in Computing Technol., Harvard Univ., Cambridge, Mass., Tech. Rep. 21–74, Sept. 1974.
[4] J. Branquart, J. Lewi, M. Sintzoff, and P. L. Wodon, "The composition of semantics in Algol 68," *Commun. Ass. Comput. Mach.*, vol. 14, pp. 697–708, Nov. 1971.
[5] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming*. New York: Academic, 1972.
[6] *IBM System/360 Operating System, Assembler Language*, Poughkeepsie, N. Y., Form C28-6514-4, 1964.
[7] *IBM System/360 Time Sharing System: Command Language User's Guide*, Yorktown Heights, N. Y., Form C28-2001, 1966.
[8] N. Wirth, "The programming language Pascal," *Acta Informatica*, vol. 1, pp. 35–63, 1971.
[9] *IBM System/360 Operating System, PL/1 Language Specifications*, New York, N. Y., Form C28-6571, 1965.
[10] O.-J. Dahl, B. Myhrhaug, and K. Nygaard, "The Simula 67 common base language," Norwegian Computing Centre, Oslo, Norway, 1968.
[11] W. A. Wulf, "Alphard: Toward a language to support structured programs," Dep. Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, Pa., Internal Rep., Apr. 1974.
[12] B. Liskov and S. Zilles, "Programming with abstract data types," in *Proc. Symp. Very High Level Languages, SIGPLAN Notices*, vol. 9, Apr. 1974, pp. 50–59.
[13] D. G. Bobrow and B. Raphael, "New programming languages for artificial intelligence," *Ass. Comput. Mach. Computing Surveys*, vol. 6, pp. 155–174, Sept. 1974.
[14] D. T. Ross, "Uniform referents: An essential property for a software engineering language," in *Software Engineering*, J. T. Tou, Ed., vol. 1. New York: Academic, 1969, pp. 91–101.
[15] J. D. Ichbiah, J. P. Rissen, and J. C. Heliard, "The two-level approach to data independent programming in the LIS system implementation language," in *Machine Oriented Higher Level Languages*, Van der Poel and Maarssen, Ed. Amsterdam: North-Holland, 1974, pp. 161–174.

Charles M. Geschke was born in Cleveland, Ohio, on September 11, 1939. He received the A.B. degree in Latin and the M.S. degree in mathematics from Xavier University, Cincinnati, Ohio, in 1962 and 1963, respectively, and the Ph.D. degree in computer science from Carnegie-Mellon University, Pittsburgh, Pa., in 1972.

He is a member of the Research Staff at Xerox Palo Alto Research Center, Palo Alto, Calif. Prior to his graduate studies at Carnegie-Mellon University, he taught mathematics at John Carroll University, Cleveland, Ohio. His current research interests are in programming languages, optimizing compilers, and the design of instruction sets for efficient execution of higher-level languages.

Dr. Geschke is a member of the Association for Computing Machinery, the Mathematical Association of America, and Pi Mu Epsilon.

James G. Mitchell was born in Waterloo, Ont., Canada, on April 25, 1943. He received the B.Sc.(Honors) degree in mathematics and physics from the University of Waterloo, Waterloo, Ont., Canada, in 1966, and the Ph.D. degree in computer science from Carnegie-Mellon University, Pittsburgh, Pa., in 1970.

While at the University of Waterloo, he was a member of the group which produced the first WATFOR compiler. His graduate work at Carnegie-Mellon University was concerned with the design and construction of interactive programming systems. Since 1971, he has been a member of the Research Staff at Xerox Palo Alto Research Center, Palo Alto, Calif. His primary interests are programming languages, interactive systems, and distributed computing environments.

Dr. Mitchell is a member of the Association for Computing Machinery.